1_Collections - Standard types_fr

May 2, 2023

1 Collections - Types standard

La bibliothèque standard de Rust contient un certain nombre de structures de données très utiles appelées **collections**. Contrairement aux tableaux et aux tuples, les données vers lesquelles ces collections pointent sont stockées sur le tas, ce qui signifie que la quantité de données n'a pas besoin d'être connue au moment de la compilation et peut augmenter ou diminuer au fur et à mesure de l'exécution du programme.

Collections disponibles dans Rust :

- Un vecteur **vector** permet de stocker un nombre variable de valeurs.
- Un **string** est une collection de caractères.
- Un **hash map** permet d'associer une valeur à une clé particulière. Il s'gait d'une implémentation particulière d'une structure plus générale appélée **map**.

2 Vecteurs

Les vecteurs permettent de stocker dans une seule structure de données plusieurs valeurs placées en mémoire les unes à côté des autres. Les vecteurs ne peuvent stocker que des valeurs de même type.

[ ]:
```
let v: Vec<i32> = Vec::new(); // crée un vecteur vide
```

Rust peut déduire le type des valeurs une fois qu'on les insère dans le vecteur, donc il n'est pas nécessaire d'utiliser des annotations. En pratique, il est plus courant de créer un `Vec< T >` qui contient des valeurs initiales en utilisant le macro **`vec!`**. Dans l'exemple suivant on a fourni des valeurs `i32` à l'initialisation, donc Rust va déduire que le type du vecteur `v` est `Vec< i32 >`.

[ ]:
```
let v = vec![1, 2, 3];
```

2.1 Mise à jours des vecteurs

[ ]:
```rust
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
println!("Vector after update: {:?}",v);
```

Comme pour les autres structures, la mémoire sera libérée quand le vecteur sort du contexte.

2.2 Lire les éléments d'un vecteur

[ ]:
```rust
fn main() {
    let v = vec![1, 2, 3, 4, 5];

    let third: &i32 = &v[2]; // première méthode; l'index commence à 0
    println!("The third element is {}", third);

    match v.get(2) { // deuxième méthode ; retourne un Option<&T>
        Some(third) => println!("The third element is {}", third),
        None => println!("There is no third element."),
    }
}
main();
```

La première méthode va provoquer une erreur (panic) si l'on pointe vers un élément inexistant. Elle est à privilégier si on souhaite que le programme s'arrête quand il y a une tentative d'accéder à un élément au-delà de la fin du vecteur. Dans le deuxième cas, la méthode `get` va retourner `None` si l'index est à l'extérieur du vecteur et ne produit pas d'erreur. Cette méthode est à privilégier si l'accès d'un élément à l'extérieur de la longueur du vecteur peut arriver occasionnellement en conditions normales (ex. l'utilisateur saisit une valeur erronée).

Prenons l'exemple suivant et rappelons la règle qui interdit d'avoir des références mutables et immuables dans le même contexte (scope) :

[ ]:
```rust
fn main() {
    let mut v = vec![1, 2, 3, 4, 5];

    let first = &v[0];

    v.push(6);

    println!("The first element is: {}", first);
}
main();
```

Ce code produit une erreur à cause de la manière dont les vecteurs fonctionnent : le fait d'ajouter un nouvel élément au vecteur nécessite d'allouer plus d'espace à la suite des éléments déjà stockés

en mémoire. Si ceci n'est pas possible, le contenu est transféré dans une zone de mémoire plus grande et l'ancien espace est libéré, ce qui créerait une référence suspendue.

2.3 Itération sur les valeurs d'un vecteur

[ ]:
```rust
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
for i in v {  // consommer le vecteur
    println!("{}", i);
}
```

Dans le cas des vecteurs mutables, nous pouvons également itérer sur chacune des références mutables afin d'apporter des modifications à tous les éléments du vecteur.

[ ]:
```rust
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
for i in &v {
    println!("{}", i);
}
```

En utilisant des itérateurs:

[ ]:
```rust
let v1 = vec![1, 2, 3];

for val in v1.iter() {
    println!("Got: {}", val);
}
```

2.4 Exercices

Exercice 1: Créer un vecteur qui stocke les données provenant d'un tableur et contenant différents types : i32, f64 et String. Le vecteur contient initialement les éléments : `[3, "blue", 10.12]`

[ ]:
```rust
fn main() {
}
main();
```

Conseil : > Pensez à une structure de données qui permet de stocker des données de différents types. Nous pouvons définir une énumération dont les variantes contiennent les différents types de valeurs ; toutes les variantes de l'énumération auront alors le même type : celui de l'énumération. Nous pouvons ensuite créer un vecteur qui contient cette énumération et donc des types différents.

Solution:

[ ]:
```rust
fn main() {
    enum SpreadsheetCell {
        Int(i32),
        Float(f64),
        Text(String),
    }

    let row = vec![
        SpreadsheetCell::Int(3),
        SpreadsheetCell::Text(String::from("blue")),
        SpreadsheetCell::Float(10.12),
    ];
}
```

Exercice 2: Écrivez une fonction qui calcule la plus grande valeur d'un vecteur d'éléments i32. > * N'oubliez pas de traiter le cas d'un vecteur vide ! > * Proposez une version qui ne consomme pas le vecteur.

[ ]:
```rust
// TO DO (1) Définir une fonction qui calcule la plus grande valeur

fn main() {
  // TO DO (2) Tester votre implémentation ici

}
main();
```

Conseil: > Utiliser la documentation pour identifier les méthodes implementées par le type std::vec::Vec : https://doc.rust-lang.org/std/vec/struct.Vec.html

Solution:

[ ]:
```rust
/// Gets the largest of the vector and consume the vector
fn into_largest(list: Vec<i32>) -> Option<i32> {
    let mut result = None;
    for value in list.into_iter() {
        result = match result {
            None => Some(value),
            Some(previous) => Some(value.max(previous))
        };
    }
    result
}

/// Gets the largest of the slice, but do not consume it
fn largest_of(list: &[i32]) -> Option<i32> {
    let mut result = None;
    for &value in list.iter() {
        result = match result {
```

```
            None => Some(value),
            Some(previous) => Some(value.max(previous))
        };
    }
    result
}

/// Gets the largest of the slice, but do not consume it
fn largest_of_idiomatic(list: &[i32]) -> Option<i32> {
    list.iter().copied().max()
}

fn main() {
    let input = vec![10,2,3,4];
    let a = largest_of(&input);
    let b = into_largest(input); // consume input

    println!("largest_of   says: {:?}", a);
    println!("into_largest says: {:?}", b);
}

main();
```

Exercice 3: Définir une structure `GeoPolygon` avec le champs `ring` qui représente une liste de sommets de type `GeoCoordinate`. Implémenter la méthode `bound` qui retourne un `GeoBoundingBox` représentant les min et max des latitudes et des longitudes du polygone. Utiliser les structures `GeoCoordinate` et `GeoBoundingBox` définies précédemment (dans le module Structures).

Exemple de coordonnées pour test :

```
GeoCoordinate {
    latitude: 46.0,
    longitude: 2.0
},
GeoCoordinate {
    latitude: 47.0,
    longitude: 2.0
},
GeoCoordinate {
    latitude: 47.0,
    longitude: 3.0
},
GeoCoordinate {
    latitude: 46.0,
    longitude: 3.0
}
```

[ ]:
```rust
#[derive(Debug)]
struct GeoCoordinate {
    /// La latitude
    latitude: f64,
    /// La longitude
    longitude: f64
}

#[derive(Debug)]
struct GeoBoundingBox {
    /// Le point ayant les coordonnées minimales
    min: GeoCoordinate,
    /// Le point ayant les coordonnées maximales
    max: GeoCoordinate
}

struct GeoPolygon {
    /// TO DO (1) Définir l'anneau du polygone
    ring:
    // TO DO (2) Implémenter la méthode bound qui retourne un GeoBoundingBox
}
```

Solution:

[ ]:
```rust
#[derive(Debug)]
struct GeoCoordinate {
    /// The latitude
    latitude: f64,
    /// The longitude
    longitude: f64
}

#[derive(Debug)]
struct GeoBoundingBox {
    /// The point with the minimal coordinates for this bounding box
    min: GeoCoordinate,
    /// The point with the maximal coordinates for this bounding box
    max: GeoCoordinate
}

#[derive(Debug)]
struct GeoPolygon {
    /// The ring of this polygon
    ring: Vec<GeoCoordinate>
}

impl GeoPolygon {
```

```
    // Gets the bounding box for a polygon
    fn bounds(&self) -> GeoBoundingBox {
        let mut min_lat = std::f64::MAX;
        let mut max_lat = std::f64::MIN;
        let mut min_lng = std::f64::MAX;
        let mut max_lng = std::f64::MIN;
        for point in self.ring.iter() {
            if point.latitude < min_lat {
                min_lat = point.latitude;
            }
            if point.latitude > max_lat {
                max_lat = point.latitude;
            }
            if point.longitude < min_lng {
                min_lng = point.longitude;
            }
            if point.longitude > max_lng {
                max_lng = point.longitude;
            }
        }
        GeoBoundingBox {
            min: GeoCoordinate {
                latitude: min_lat,
                longitude: min_lng
            },
            max: GeoCoordinate {
                latitude: max_lat,
                longitude: max_lng
            }
        }
    }
}

fn main() {
    let ring = vec![GeoCoordinate {
        latitude: 46.0,
        longitude: 2.0
    },
    GeoCoordinate {
        latitude: 47.0,
        longitude: 2.0
    },
    GeoCoordinate {
        latitude: 47.0,
        longitude: 3.0
    },
    GeoCoordinate {
```

```
        latitude: 46.0,
        longitude: 3.0
    }];
    let polygon = GeoPolygon { ring };
    let bounds = polygon.bounds();
    println!("{:#?}", &bounds);
}
main();
```

3 Strings

Le type String, qui est fourni par la bibliothèque standard de Rust (et non pas codé dans le langage de base) est un type de chaîne de caractères codé en UTF-8, évolutif, mutable et "owned".

[ ]:
```
let mut s = String::new(); // créer un nouveau String
```

[ ]:
```
let data = "initial contents";

let s = data.to_string();

// la méthode marche aussi directement sur les littéraux :
let s = "initial contents".to_string();
```

[ ]:
```
let s = String::from("initial contents");
```

> **Note:** La bibliothèque standard Rust contient d'autres types de chaînes de caractères : `OsString`, `OsStr`, `CString`, and `Cstr`. En plus, il existe des crates qui peuvent fournir davantage d'options. Remarquez les terminaisons *String* et *Str*. Elles font référence à des variantes "owned" ou "borrowed", tout comme les types `String` et `str`. Ces types proposent différentes possibilités d'encodage et de représentation dans la mémoire.

[ ]:
```
let mut s1 = String::from("foo");
let s2 = "bar";
s1.push_str(s2);
println!("s2 is {}, s1 is {}", s2, s1);
```

La méthode push prend un seul caractère en paramètre et il l'ajoute au String.

[ ]:
```
let mut s = String::from("lo");
s.push('l');
```

3.1 Concaténation de Strings

[ ]:
```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // note s1 has been moved here and can no longer be used
```

Pour des combinaisons de plus compliquées, nous pouvons utiliser la macro **format!** :

[ ]:
```
let mut s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");

let s = format!("{}-{}-{}", s1, s2, s3); // ne prend pas la propriété␣
 ↪(ownership) de ses paramètres
println!("{}", s);

s1.push_str("xx");
println!("{}", s1);
```

3.2 Représentation interne

Un String est un wrapper qui encapsule un Vec< u8 >.

[ ]:
```
let len = String::from("Hola").len();
println!("{}",len);
let len = String::from("      ").len();
println!("{}",len);
```

Pour encoder “ ” en UTF-8 on a besoin de 24 bytes parce que chaque valeur Unicode scalar dans cette chaîne prend 2 bytes. Un index sur les bytes de la chaîne n'est pas forcément corrélé à une valeur Unicode scalar valide. Pour cette raison, l'indexation des String n'est pas une opération valide.

[ ]:
```
let hello = "      ";
let answer = &hello[0]; // error
```

3.3 Slice sur les Strings

Pour récupérer un slice, plutôt que d'utiliser la syntaxe [] avec un nombre, on utilise [] un intervalle et on sélectionne les bytes qui nous intéressent :

[ ]:
```
let hello = "      ";

let s = &hello[0..4];
println!("s: {}", s);
```

3.4 Itération sur les Strings

[ ]:
```
for c in "  ".chars() { // chaque itération retourne une valeur de type␣
 ↪caractère (char)
    println!("{}", c);
}
let text = "  ";
println!("First char {:?}", text.chars().nth(0));
```

[ ]:
```rust
for b in "  ".bytes() { // chaque itération retourne un raw byte
    println!("{}", b);
}
let bb: std::str::Bytes = "  ".bytes();
```

3.5 Exercices

Exercice 1: Ecrivez un petit programme qui vérifie si un mot est un palindrome.

[ ]:
```rust
fn main() {

}
main();
```

> **Conseil:** Consulter le manuel pour comprendre le fonctionnement de *Iterator* et de sa méthode *nth*. (https://doc.rust-lang.org/std/iter/trait.Iterator.html) ; utiliser un itérateur sur les bytes ou sur les caractères.

Solution:

[ ]:
```rust
fn palindrome(s: String) -> bool {
    let len = s.len();
    if len == 0 {return true;}
    let mut i = 0;
    for c in s.chars() {
        match s.chars().nth(len-i-1) {
            Some(value) => {if value != c { return false;}},
            None => break,
        }
        i += 1;
    }
    return true
}

fn palindrome_idiomatic(text: &str) -> bool {
    text
        .chars()
        .zip(text.chars().rev())
        .all(|(a, b)| a == b)
}

fn main() {
    let word = String::from("abccba");
    let another_word = String::from("abcccca");
    let empty_word = String::from("");
    match palindrome_idiomatic(&another_word) {
        true => println!("The word is a palindrome"),
        false => println!("The word is not a palindrome"),
```

```
    }
}
main();
```

4 Hash map

Le type `HashMap< K, V >` stocke une correspondance entre clés de type `K` et valeurs de type `V`. Ceci se fait via une fonction de hachage, qui détermine comment placer ces clés et valeurs en mémoire. Les hash maps sont utiles lorsqu'on recherche des données non pas en utilisant un index (comme avec des vecteurs), mais en utilisant une clé qui peut être de n'importe quel type.

4.1 Créer une hash map

[ ]:
```
use std::collections::HashMap;
fn main()
{
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
    println!("Scores: {:?}", scores);
}
main();
```

Une autre façon de construire une hash map consiste à utiliser la méthode **`collect`** sur un vecteur de tuples, où chaque tuple est constitué d'une clé et de sa valeur.

[ ]:
```
use std::collections::HashMap;
fn main()
{
    let teams  = vec![String::from("Blue"), String::from("Yellow")];
    let initial_scores = vec![10, 50];
    let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).
 ↪collect();
    println!("Scores: {:?}", scores);
}
main();
```

4.2 Hash map et ownership

Pour les types qui implémentent le trait `Copy`, comme `i32`, les valeurs sont copiées dans la hash map. Pour les valeurs "owned" comme `String`, les valeurs sont déplacées et la hash map devient le propriétaire (owner) de ces valeurs :

[ ]:
```
use std::collections::HashMap;

fn main() {
    let field_name = String::from("Favorite color");
```

```
    let field_value = String::from("Blue");

    let mut map = HashMap::new();
    map.insert(field_name, field_value);
    // field_name et field_value sont invalides à ce moment là,
    // testez et vérifier l'erreur !
    println!("Field name: {}", field_name);
}
main();
```

Conclusion: Si nous insérons des références à des valeurs dans la hash map, les valeurs ne seront pas déplacées dans la hash map. Les valeurs auxquelles les références renvoient doivent être valables au moins aussi longtemps que la table de hachage.

4.3 Accès aux valeurs

Pour obtenir une valeur de la hash map on fournit sa clé à la méthode `get`. Consultez la documentation pour voir la définition de la méthode `get` : https://doc.rust-lang.org/std/collections/struct.HashMap.html#method.get

[ ]:
```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name);
    match score {
        Some(&value) =>  println!("The score for the {} team is {}", team_name,␣
 ↪value),
        None => (),
    }
}
main();
```

On peut itérer sur une paire clé/valeur de la hash map en utilisant, comme pour les vecteurs, *for loop* :

[ ]:
```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Yellow"), 50);
```

```
    for (key, value) in &scores {
        println!("{}: {}", key, value);
    }
}
main();
```

4.4 Mise à jour de la Hash Map

Si l'on essaye d'insérer dans la hash map une paire clé/valeur et que la clé existe déjà, l'ancienne valeur se trouvant à la même clé sera remplacée avec la nouvelle valeur.

[ ]:
```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();

    scores.insert(String::from("Blue"), 10);
    scores.insert(String::from("Blue"), 25);

    println!("{:?}", scores);
}
main();
```

Il est courant de vérifier si une clé particulière a une valeur et, si ce n'est pas le cas, d'insérer une valeur. Le hash map a une API spéciale **`entry`** qui prend en paramètre la clé à vérifier. La valeur retournée est un enum appelée **`Entry`** qui représente une valeur qui peut exister ou pas.

[ ]:
```
use std::collections::HashMap;

fn main() {
    let mut scores = HashMap::new();
    scores.insert(String::from("Blue"), 10);

    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

    println!("{:?}", scores);
}
main();
```

La méthode `or_insert` de `Entry` retourne une référence mutable à la valeur qui correspond à la clé, si la clé existe. Si elle n'existe pas, le paramètre est inséré en tant que valeur associée à la clé et la méthode retourne une référence mutable vers la nouvelle valeur.

Une autre opération fréquente sur les hash maps est de rechercher la valeur d'une clé et de la mettre à jour en fonction de l'ancienne valeur.

4.5 Exercices

Exercice 1 : Écrire le code qui permet de calculer le nombre d'apparitions de chaque mot dans un texte.

> **Conseil :** Chercher la méthode *split_whitespace* dans la documentation. https://doc.rust-lang.org/std/primitive.str.html#method.split_whitespace

[ ]:
```rust
use std::collections::HashMap;

fn main() {
    let text = "hello world wonderful world";
    // TO DO (1) Créer une hashmap nommée "map"
    // TO DO (1) Itérer sur les mots
    // TO DO (2) Calculer le nombre d'apparitions des mots
    // TO DO (3) Afficher la hashmap avec le nombre d'apparitions pour chaque␣
 ↪mot
}
main();
```

Solution:

[ ]:
```rust
use std::collections::HashMap;

fn main() {
    let text = "hello world wonderful world";

    let mut map = HashMap::new();

    for word in text.split_whitespace() {
        let count = map.entry(word).or_insert(0);
        *count += 1;
    }

    println!("{:?}", map);
}
main();
```