5_Structures_fr

May 2, 2023

1 Structures

Une `struct` ou structure, est un type défini par l'utilisateur qui permet de packager sous un même nom plusieurs valeurs ensemble en un group cohérent. Les structures et les énumérations (voir module suivant) sont les blocks de base pour la création de nouveaux types dans votre domaine métier pour tirer le meilleur partie des vérifications faites par le compilateur.

Tout comme les tuples, les éléments composant une structure peuvent être de types différents. A l'inverse des tuples, chaque membre a un nom spécifique pour qu'il soit clair ce que chaque valeur signifie. Les structures sont donc plus flexibles que les tuples : vous n'avez pas à vous appuyer sur l'ordre des données pour spécifier ou accéder à la valeur d'un membre de l'instance.

[ ]:
```
fn main() {
    struct User {
        username: String,  // ceci est un champ (membre) (field)
        email: String,
        sign_in_count: u64,
        active: bool,
    }

    let user1 = User { // ceci est une instance de `User`
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
}
main();
```

Pour accéder à un membre particulier d'une structure on utiliser la notation usuelle avec le point (.). Si l'instance est mutable, il est possible de changer la valeur de ce membre avec cette même notation en lui assignant une nouvelle valeur.

> **Note:** l'instance entière doit être mutable. Rust ne permet pas à certains membres seulement d'être mutables.

[ ]:
```
fn main() {
    struct User {
        username: String,  // ceci est un champ
```

```
        email: String,
        sign_in_count: u64,
        active: bool,
    }

    let mut user1 = User { // <-- user1 est mutable
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
    // mutation de email in user1
    user1.email = String::from("anotheremail@example.com");
}
main();
```

A l'instar de toutes les expression, nous pouvons construire une instance d'une structure comme la dernière expression d'une fonction pour retourner cette instance.

[ ]:
```
fn main() {
    struct User {
        username: String,
        email: String,
        sign_in_count: u64,
        active: bool,
    }

    fn build_user(email: String, username: String) -> User {
        User { // <-- dernière expression de la fonction `build_user`
            email: email,
            username: username,
            active: true,
            sign_in_count: 1,
        }
    }
}
main();
```

Lorsque le nom d'un paramètre et celui d'un membre de la structure sont exactement le même, il est possible d'utiliser un raccourci syntaxique :

[ ]:
```
fn main() {
    struct User {
        username: String,
        email: String,
        sign_in_count: u64,
        active: bool,
    }
```

```
    fn build_user(email: String, username: String) -> User {
        User {
            email,            // <-- raccourci au lieu de email: email
            username,         // <-- raccourci ici aussi
            active: true,
            sign_in_count: 1,
        }
    }
}
main();
```

1.1 Créer une instance à partir d'autres

Pour créer une instance à partir d'une autre de manière basique :

[ ]:
```
fn main() {
        struct User {
                username: String,  // this is a field
                email: String,
                sign_in_count: u64,
                active: bool,
        };

        let user1 = User {
                email: String::from("someone@example.com"),
                username: String::from("someusername123"),
                active: true,
                sign_in_count: 1,
        };
        let user2 = User { // sans utiliser la syntaxe de mise à jour
                email: String::from("another@example.com"),
                username: String::from("anotherusername567"),
                active: user1.active,
                sign_in_count: user1.sign_in_count,
        };
        println!("user2: {}", user2.username);
        let user3 = User { // syntaxe de mise à jour
                email: String::from("yet_another@example.com"),
                username: String::from("username456"),
                ..user1  // <-- reste des membres copiés depuis user1
        };
        println!("user3: {}", user3.username);
}
main();
```

1.2 Structures sous forme de tuples

Vous pouvez définir une structure sous une forme similaire à celle d'un tuple. Cette syntaxe s'appele *tuple struct*. Cette syntaxe permet d'associer un nom explicite à la structure, là où un tuple simple est anonyme, mais les membres, eux, sont anonymes, à l'instar d'un tuple simple. La syntaxe *tuple struct* est utile pour donner un nom un tuple pour le différecier des autres tuples de types équivalents, et que le nommage des membres serait superflu ou redondant.

[ ]:
```
fn main() {
    struct Color(i32, i32, i32);
    struct Point(i32, i32, i32);

    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);

    println!("black = ({}, {}, {})", black.0, black.1, black.2);
}
main();
```

Notez que les variables `black` et `origin` sont de type différent parce qu'elles sont des instances de structure tuple différentes. Toutes les structures que vous définissez ainsi sont bien des types différents, même si le typage des membres est identique. Les *tuple struct* se manipule de la même façon que les tuples standards : vous pouvez les déstructurer en éléments individuels, utiliser `.` suivit d'un index pour accéder à un membre, etc.

1.3 Définitions de méthodes

Les méthodes sont similaires aux fonctions en ceci qu'elles sont définies également avec le mot clé `fn` et leur nom; elles peuvent avoir des paramètres et un type de retour et contenir du code exécuté à leur appel. Les méthodes diffèrent en cela qu'elles sont définies dans le contexte d'une structure (ou d'une énumération, ou d'un trait). Lorsqu'une méthode a besoin de référencer l'instance de la structure, le premier paramètre doit être `self` qui représente cette instance sur laquelle la méthode est appelée.

[ ]:
```
fn main() {
    struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        fn area(&self) -> u32 {
            self.width * self.height
        }
    }

    let rect1 = Rectangle { width: 30, height: 50 };
```

```
    println!(
        "L'aire du rectange est de {} pixels.",
        rect1.area()
    );
}
main();
```

`&self` est utilisé comme paramètre parce que nous ne voulont pas consommer l'objet `rect1` en récupérant sa propriété, mais seulement y accéder (à travers une référence), et que nous voulons simplement lire son contenu, et non le modifier. Une méthode peut prendre la propriété (ownership) de l'objet avec l'utilisation simple de `self` comme premier paramètre. L'usage de cette technique est relativement rare et dénote en générale une méthode consommant l'objet pour le transformer en quelque chose d'autre. Le système de propriété (ownership) est discuté en détail dans le prochain module.

Il est possible de définir des méthodes qui ne prennent pas `self` ou `&self` en premier paramètre. Ces méthodes sont alors statiques, relativement à leur type.

En réalité, c'est exactement comme ceci que l'on écrit les constructeurs en Rust. Avez-vous remarquer que jusqu'à maintenant nous avons toujours explicitement construit les structures avec leurs membres ? La convention en Rust pour les constructeurs est de définir une méthode statique nommée `new` prenant en paramètres les informations requises et retournant une structure initialisée :

> Notez que le nom `new` de la fonction n'a absolument pas d'interprétation spéciale, il s'agit d'une convention.

[ ]:
```
fn main() {
    struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        /// ceci est un constructeur, le nom new n'a pas de sémantique␣
↪particulière pour Rust
        fn new(width: u32, height: u32) -> Rectangle {
            Rectangle { width, height }
        }

        fn area(&self) -> u32 {
            self.width * self.height
        }
    }

    let rect1 = Rectangle::new(30, 50);

    println!(
        "L'aire du rectange est de {} pixels.",
```

```
        rect1.area()
    );
}
main();
```

2 Exercises

Exercice 1: Réécrivez le code suivant en utilisant une structure :

[ ]:
```
let width: u32 = 30; // the width of the rectangle
let height: u32 = 50; // the height of the rectangle
println!("The width of the rectangle is {} and the height is {}", width,␣
 ↪height);
```

[ ]:
```
fn main() {
    // TO DO (1) Créer une structure Rectangle

    // TO DO (2) Ajouter un constructeur

    // TO DO (3) Ajouter une méthode pour afficher la largeur et la longueur
}
main();
```

Solution:

[ ]:
```
fn main() {
    struct Rectangle {
        width: u32,
        height: u32,
    }

    impl Rectangle {
        fn new(width: u32, height: u32) -> Rectangle {
            Rectangle { width, height }
        }

        fn display(&self) {
            println!("The width of the rectangle is {} and the height is {}",␣
 ↪self.width, self.height);
        }
    }

    let rect1 = Rectangle::new(30, 50);
    rect1.display();
}
main();
```

Comment pourrions-nous faire pour afficher dans la console les informations d'une instance de structure, par exemple pour débugger ?

[ ]:
```
struct Rectangle {
    width: u32,
    height: u32,
}

let rect1 = Rectangle { width: 30, height: 50 };

println!("rect1 is {}", rect1);
```

La macro `println!` peut réaliser beaucoup d'opérations de formattage pour l'affichage. Par défaut, les accolades `{}` permettent d'indiquer à `println!` d'utiliser le formattage du trait `Display`, qui en général vise plutôt les utilisateurs finaux. Les types primitifs que nous avons vu jusqu'à maintenant implémentent tous `Display`, parce qu'il n'y a qu'une seule manière d'afficher `1`. La manière d'afficher les structure, par contre, n'est pas toujours aussi claire, il y a beaucoup de possiblités. Rust ne fait pas d'hypothèse et par défaut Rust ne fournit pas d'implémentation de `Display` pour les structures.

Mais Rust fournit les fonctionnalités pour afficher des informations utiles au debug, mais nous devons explicitement le demander pour nos structures. Pour cela, nous ajoutons l'annotation `#[derive(Debug)]`juste avant la définition de la structure.

[ ]:
```
fn main() {
    #[derive(Debug)]
    struct Rectangle {
        width: u32,
        height: u32,
    }


    let rect1 = Rectangle { width: 30, height: 50 };

    println!("rect1 is {:?}", rect1); // tell println! to use an output format␣
 ↪called Debug
}
main();
```

En utilisant **`:?`** à l'intérieur des accolades, nous indiquons à `println!` que nous voulons utiliser le formattage fournit par le trait `Debug`. Ce dernier formatte la structure sous une forme utile pour les développeurs pour que nous puissions l'inspecter au debug.

> **Note:** Les traits sont similaires à ce que l'on appele interface dans d'autres langages, avec cependant quelques différences.

Lorsque nous interagissons avec des structures de données avec beaucoup de membres, il peut être nécessaire de formatter la structure sous une forme plus facile à lire, dans ce cas, le formattage **`{:#?}`** est plus utile.

Exercise 2: Essayer d'utiliser le formattage **{:#?}** dans l'exemple précédent.

Exercise 3: Définissez une structure *GeoCoordinate*, contenant les membres *latitude* et *longitude*. Puis implémentez les méthodes *to_degrees()* and *to_radians()* qui transforment ces coordonnées en degrés ou en radians.

Implémentez une structure GeoBoundingBox contenant les coordonnées min et max. Testez votre implémentation.

Sample test data: latitude: 44.84, longitude: -0.58

> **Tips:** Regardez la documentation du type `f64` et cherchez les méthodes `to_degrees` and `to_radians`. https://doc.rust-lang.org/std/primitive.f64.html#method.to_degrees

[ ]:
```
// TO DO (1) Define the GeoCoordinate structure

// TO DO (2) Implement for the GeoCoordinate structure the to_degrees and␣
 ↪to_radians methods

fn main() {
    // TO DO (3) Test your implementation

}
main();
```

Solution:

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

impl GeoCoordinate {
    /// Transforms to values in degrees
    fn to_degrees(&self) -> GeoCoordinate {
        GeoCoordinate {
            latitude: self.latitude.to_degrees(),
            longitude: self.longitude.to_degrees()
        }
    }

    /// Transforms to velues in radians
    fn to_radians(&self) -> GeoCoordinate {
        GeoCoordinate {
            latitude: self.latitude.to_radians(),
            longitude: self.longitude.to_radians()
        }
```

```
    }
}

fn main() {
    let x = GeoCoordinate {latitude: 44.84, longitude: -0.58}; // coordonnées␣
 ↪de la ville de Bordeaux
    println!("{:#?}", x.to_radians());
}
main();
```