Rust

Mes notes de Rust

Introduction

Rust a commencé comme projet personnel de Graydon Hoare en 2006, un employé chez Mozilla, Mozilla va ensuite sponsoriser ce projet et compte l'utiliser pour programmer un futur moteur de navigation prévu pour être plus rapide que Gecko (utilisé pour Firefox).

Vers 2011, Rust se compile soi-même. Ainsi le compilateur Rust est écrit lui même en Rust. Rust va ensuite gagner en popularité et être utilisé dans de plus en plus de grosses entreprises tel que Alphabet (Google), Meta (Facebook), Microsoft, Discord, Amazon, Dropbox. Et un support pour Rust a également été ajouté dans le kernel Linux.

La documentation

Rust maintient 3 documentations officielles pour les débutants : une globale, une par exemples, et une autre basé sur des petits défis à faire en Rust. La pluspars des documentations se font via un site qui a exactement la même forme que celui ci et les exemples de codes peuvent être directement exécuté et modifié dans le navigateur.

La documentation des librairies est standardisées et est écrite directement dans le code, et les exemples d'utilisations écrits dans la documentation peuvent être testé automatiquement au même moment que la génération pour avoir une documentation toujours à jour.

L'utilisation de cette forme de documentation est très agréable car on peut cliquer d'un type à l'autre pour voir comment ça fonctionne, les différents champs, méthodes, fonctions et traits. On peut aussi voir le code source d'une partie de la librairie directement dans la documentation.

L'absence de null (et Options et Result)

Rust n'a pas de valeur "null" pour indiquer l'absence d'une valeur. A partir d'une déclaration de fonction ou de méthode, on peut directement voir si la fonction pourrait ne rien retourner et le compilateur nous force à traiter le cas où une erreur surviendrais ou que la valeur serait None.

Le but du compilateur est de tester un maximum de cas possibles pour empécher des mauvaises surprises plus tard.

Les erreurs Rust et les bonnes pratiques (clippy)

Le système d'erreurs de Rust est magnifique et super clair. Les erreurs sont colorées, avec des inforamations sur comment les résoudres ainsi que des liens vers la documentation de Rust en cas de soucis de compréhension.

Pour appliquer les bonnes pratiques en Rust il y a aussi un outil appellé "clippy" (lui aussi par défault) qui peut dans beaucoup de cas améliorer la qualité du code automatiquement et résoudre certaines erreurs automatiquement également.

La capacité de créer ses propres types

On peut aussi créer ses propres types en Rust comme des classes dans d'autres languages mais en beaucoup plus flexibles (on peut implémenter de nouvelles méthodes sur n'importe quel type). Le but est ainsi de faire en sorte que des cas problèmatiques qui ne sont pas censé exister, ne soit simplement pas représentable dans le programme. (voir le chapitre sur struct et enum)

La standardisation du style de code

Le style de code à employer est standardisé en rust et peut être automatiquement appliqué par les IDE ou par la commande `cargo fmt` ce qui permet d'avoir un code beaucoup plus compréhensible et uniforme.

La magie des macros

Les macros permettent de faire encore plus de tests au moment de la compilation ce qui permet par exemple à la librairie `sqlx` (pour intéragir à des bases de données mysql, postgres ou sqlite) de tester les requêtes au moment de la compilation sur une base de donnée de test ou encore à la librairie `yew` (pour faire du webassembly) de tester la validité du code HTML.

Rust fonctionne quasi partout

Rust fonctionne sur toutes les plateformes, même parfois anciennes, Rust fonctionne aussi sur des systèmes embarqués et même dans le navigateur comme remplacement de Javascript (avec webassembly) et Webassembly est parfois utilisé aussi comme pont avec d'autres architectures, plus obscures.

Le gestionaire de libraries

Je dis librarie depuis le début mais dans Rust on appelle ça des *crates*, et il se trouve que l'on peut explorer les differents crates sur crates.io et leur documentation sur docs.rs. On peut ainsi

facilement trouver des librairies et les ajouter à son projet en faisant `cargo add <nom de la lib>` (ce qui va automatiquement ajouter la version actuelle de la librarie dans la liste des dépendences)

Rapidité, efficacité et optimisation

Rust n'a pas de *garbage collector* comme beaucoup d'autres languages pour gérer la mémoire. A la place, Rust utilise un systeme ingénieux appelé *borrow checker*, qui vérifie au moment de la compilation pour faire en sorte que le code soit le plus optimisé possible.

Stabilité et durabilité

Rust est fait pour que ce qui est programmé en Rust soit toujours compatible avec les futures versions et soit également très durable (autant au niveau écologique que temporel). C'est nottament dû à l'efficacité et l'optimisation citée plus haut.

Installation et Hello World

Pour commencer notre découverte de Rust on va d'abord devoir installer `rustup`. On peut l'installer via des gestionaire de paquets (`chocolatey` sous Windows, `brew` sous mac, ou `apt` sous debian par exemple).

Sinon les instructions pour l'installer sont disponibles sur rust-lang.org.

Si vous ne voulez pas installer Rust, il est aussi possible de tester des choses dans le Rust Playground en ligne ici.

Hello World

On peut maintenant créer un nouveau projet, en créant un nouveau dossier et en y lançant `cargo new hello-rust`

```
cargo new hello-rust
cd hello-rust
# Pour initialiser un projet rust dans un dossier existant on peut utiliser 'cargo init'
```

Maintenant on va avoir un projet Hello World par défault. Dont le fichier `src/main.rs` est le suivant :

```
fn main() {
    println!("Hello, world!");
}
```

Pour lançer le projet il suffit de lancer la commande `cargo run`

```
cargo run
```

Et voilà ! Si vous avez decidez de le faire dans le Rust Playground, vous avez juste à cliquer sur le bouton "Run".

Les outils pour développer en Rust

Un IDE est fortement conseillé pour programmer en Rust car beaucoup d'informations sont données par l'IDE (ou le compilateur) à propos de comment faire le programme.

Ainsi on peut voir et résoudre les problèmes tout de suite avec un IDE à la place de les découvrir au moment de la compilation.

Personellement j'utilise Helix, mais vous pouvez aussi utiliser VS-Code ou encore plus n'importe quel IDE du moment qu'il supporte Rust.

Variables et types primitifs (et cast)

En rust les variables sont définie avec `let` et sont immuable par défault

```
// Cette variable est immuable
let x = 42;

// Cette variable est muable (peut être modifiée)
let mut y = 42;

// On peut préciser un type avec :
// Préciser les différents types n'est pas obligatoire, je le fais ici uniquement pour l'explication
// Rust a la capacité dans beaucoup de cas de deviner de quel type la variable doit être
let z: i32 = 42;

// Une variable qui commence par _ est un message au compilateur qu'il ne doit pas se soucier qu'elle n'est jamais utilisée
let _a = 42;

// On peut bien évidemment faire des opération mathématiques dans nos variables
let b = x + y;
```

Maintenant interessons nous un peu aux types de base. Pour ce qui est des nombres :

Longueur	Signé	Non signé	Flottant (à virgule)
8-bit	`i8`	`u8`	
16-bit	`i16`	`u16`	
32-bit	`i32`	`u32`	`f32`
64-bit	`i64`	`u64`	`f64`
128-bit	`i128`	`u128`	
arch	`isize`	`usize`	

Pour ce qui est des chaines de caractères, le type primitif qui correspond est le `str` (a ne pas confondre avec `String` qui n'est pas un type primitif):

```
// str est toujours sous la forme &str car c'est une référence
let x: &str = "hello ";
let y: &str = "world";

// &str est statique, on ne peut donc pas concaténer x et y
let z: &str = x + y;
```

Ensuite on peut créer un tuple

```
// mut pour pouvoir le modifier après
let mut tuple: (i32, &str, f32) = (42,"answer to life",42.0);

// On peut ensuite faire référence aux éléments du tuple avec .0 .1 .2 etc
println!("{} is {}", tuple.1, tuple.0);

// On peut bien évidemment modifier notre tuple aussi
tuple.0 = 27;
println!("{} is not {}", tuple.1, tuple.0);
```

Maintenant on va voir un autre type de groupement d'élément : l'array. A ne pas confondre avec le Vecteur qui n'est pas un type primitif.

```
// mut pour pouvoir le modifier après
// Les arrays ont eu aussi une taille fixe, mais contrairement aux tuples, ils ne peuvent contenir qu'un seul type
// Ici la taille est directement indiquée dans le type par le "; 3" qui signifie que l'array ne peut avoir que 3 éléments
let mut array: [i32; 3] = [42,2048,196883];

// On peut accéder aux éléments de l'array avec [0] [1] [2], etc
println!("{}, {}, {}", array[0], array[1], array[2]);

// Et voici comment modifier un array
array[2] = 24;
println!("{}, {}, {}", array[0], array[1], array[2]);
```

Bien sur on a le type booléen

```
// Valeurs possibles: true, false
let mon_booleen: bool = true;
```

```
println!("{}", mon_booleen);
```

Et enfin comme dernier type on va lister `char` pour contenir un caractère

```
let premier_a: char = 'A';

// Pour le deuxieme on va encoder la valeur UTF 8 de 'A' en hexadécimal
let deuxiem_a: char = '\u{0041}';

// On peut maintenant afficher les deux
println!("{} et {} sont les même caractères.", premier_a, deuxiem_a);
```

Il reste encore deux types à traiter, les `slice` et les `reference` mais on va en parler dans un autre chapitre.

Casting

On peut transformer un type primitif en un autre type primitif en utilisant le mot clé `as`, par exemple :

```
// On converti un f32 en i32 ici mais ça aurait pu être autre chose
let float: f32 = 67.957;
let int: i32 = float as i32;
println!("this is a float: {int}");
```

Pour les &str et String on peut utiliser `.parse()` :

```
let string_number = "67.957";
let float: f32 = string_number.parse().expect("The input is incorrect");
println!("this is a float: {float}");
```

C'est normal si vous ne comprenez pas encore tout de ce code car on a pas encore vu le `.expect()` (voir sur le chapitre de Result et Option) ni le `String`.

En savoir plus

- *docs.rs* - Primitive Types

Strings et affichage formatté

On a vu dans le chapitre sur les types primitifs que l'on peut faire des chaines de caractères avec `&str`, mais ici on va voir la version plus évoluée avec `String`

Ce type plus complexe a beaucoup de méthodes et de possibilités. D'abord voyons comment le créer :

```
// On peut créer un nouveau String vide comme ceci:
let mon_string = String::new();

// On peut aussi créer un String à partir d'un litéral (exemple "hello")
let mon_autre_string = String::from("Hello World");

// Sinon on peut aussi convertir un &str en String avec la méthode de &str 'to_string'
let troisieme_string = "Foo Bar".to_string();

// Affichons tout ça
println!("{} {} {}", mon_string, mon_autre_string, troisieme_string);
```

Contrairement aux &str, on peut aussi concaténer des String avec des &str

```
let un = String::from("hello ");
let deux = "world";
let trois = un + deux;
println!("{}", trois);
```

Je ne vais pas passer en revue toutes les méthodes de String, car après avoir compris les types `Option` que l'on va voir dans un futur chapitre et celui sur les fonctions, vous serez en mesure de comprendre toutes les méthodes à partir de la documentation.

Parmis d'autres, voici quelques exemples de choses qui peuvent être faites avec String: remplacer des chaines de caractères, remplacer le début ou la fin de chaine de caractère (attention peut nécessiter d'utiliser la méthode `trim()` avant), etc.

Toutes la documentation est disponible **ici**.

Affichage formatté

Maintenant pour afficher des choses dans le terminal ou formater des choses d'une certaine manière. On peut utiliser des *macros*.

La premiere que j'ai déjà utilisé avant est `println!()`, l'autre pour formatter est `format!()` et enfin une troisième pour print sans retour à la ligne est `print!()`

```
// On peut print à partir d'un litéral
println!("Hello World");

// Mais on peut aussi print avec une sorte de template
let nombre = 42;
let chaine = "La réponse de la vie est ";

// Pour que l'un type puisse être affiché comme ceci il faut qu'il implémente le type `Display`
// Ne faites pas attention au & maintenant, on va y venir plus tard
println!("{} {}", &chaine, nombre);

let result = format!("Même chose mais avec format d'abord → {} {}", chaine, nombre );
println!("{}", result);

// En revanche on peut utiliser {:?} et {:#?} pour afficher certain types qui ne sont normalement pas affichables
let monvecteur = vec![chaine, "42"];
println!("{:?}", &monvecteur);
println!("{:#?}", monvecteur); // Celui ci est appellé "pretty print"
```

Ownership, lifetimes et références

Maintenant il faut s'attaquer à quelque chose de très important dans Rust : la manière dont laquelle la mémoire est gérée.

Les types de mémoire

Pour comprendre tout ceci, il faut d'abord que l'on s'attaque aux types de mémoire: le **stack** et le **heap**

stack explained with animation

- Le stack (ou la pile), peut être imaginé comme un pile d'assiettes, on peut mettre des assietes sur la pile (push) ou en retirer (pop). C'est une mémoire plus rapide que le Heap, car il n'y a pas besoin de calculer un emplacement mémoire et de l'allouer, mais pour l'utiliser il faut connaitre d'avance la taille des valeurs à stoquer (donc une taille fixe). C'est ici que toutes les valeurs qui ont une taille fixe sont stoquées.
- Le heap c'est là que vont tous les types dont la taille est variable et ne peut pas être connue d'avance. Ainsi on trouve un emplacement mémoire suffisament grand et on y alloue des valeurs. La référence (l'addresse de l'emplacement) peut lui être stoqué dans le stack sous forme de pointeur ou de référence (slice).

> Note : Ce n'est pas par ce qu'un type semble avoir une taille variable à première vue qu'il l'est vraiment. Par exemple, un Vecteur est un struct (donc une taille fixe) d'un `RawVec` qui est sur le Heap. On peut savoir si un type est directement sur le Heap si il a le trait `Allocator`

L'ownership

Chaque valeur en Rust a un *propriétaire* et ne peut en avoir qu'un seul à la fois. Et chaque valeur a une durée de vie après laquelle elle n'existe plus.

Par exemple

```
fn main() {
  let a = String::from("Hello World"); // a est propriétaire du String
  let b = a;  // b devient le nouveau propriétaire de String

  // Par conséquent a n'a plus accès au String et on ne peut plus accéder à a
  println!("Ceci va générer une erreur : {}", a);
}
```

Maintenant voyons en plus la durée de vie

```
fn ma_fonction(s: String) {
  // Fait des choses
} // A partir d'ici la valeur a n'existe plus (out of scope)

fn main() {
  let a = String::from("Hello World"); // a est propriétaire du String
  ma_fonction(a); // ma_fonction devient le nouveau propriétaire
}
```

En général les accolades {} représentes la durée de vie. Toute variable définie dans ceux ci, ne seront valable que pour cette durée. Donc il en va de même pour les `if`, `loop`, `for`, etc.

Copier la valeur

Maintenant si on peut copier la valeur en elle même on peut utiliser deux traits (on va voir dans un chapitre suivant ce que sont les traits) dépendant du type `Copy` ou `Clone`.

En pratique, tous les types primitifs (qui sont dans le stack) ont le trait `Copy`, ce qui signifie que leur valeurs sont automatiquement copiées.

```
fn ma_fonction(x: i32) {
  // Faire des choses
} // A partir d'ici x n'existe plus

fn main() {
  let a = 5; // a est propriétaire de 5
  let b = a; // la valeur de a est copiée dans b. b est propriétaire de 5 également mais pas dans le même emplacement mémoire
  ma_fonction(a); // la valeur de a est copiée dans ma_fonction. ma_fonction est aussi propriétaire de cette copie de valeur
```

```
  println!("{} == {}", a, b); // println n'est pas propriétaire de a et b car c'est une macro qui prends la référence de ceux ci (voir plus tard)
} // a et b n'existe plus à partir d'ici
```

En revanche pour ce qui est des types qui sont dans le Heap, on doit utiliser le trait `Clone` manuellement :

```
fn ma_fonction(s: String) {
  // Faire des choses
} // la copie du string n'existe plus

fn main() {
  let a = String::from("Hello World"); // a est propriétaire du String
  let b = a.clone(); // On copie la valeur de a dans b. B est donc propriétaire de la copie de a
  ma_fonction(b.clone()); // On copie la valeur de b dans ma_fonction. ma_fonction est propriétaire de la copie de b

  // Par conséquent les deux valeurs restent accessibles
  println!("{} == {}", a, b);
}
```

Les références (slices)

animation référence
Image not found or type unknown

Maintenant quand on ne veut pas copier la valeur mais quand même accéder à la mémoire, on peut utiliser `&` aussi appellé *slice* ou *référence*

```
fn ma_fonction(s: String) {
  // faire des choses
}

fn main() {
  let a = String::from("Hello World");
  let b = &a; // b est de type slice (type primitif stoqué dans le stack) et est la référence vers le String de a
  // n'est donc pas propriétaire du String mais seulement de la référence

  // Mais String et &String ne sont pas les même types, donc une fonction tel que ma_fonction qui demande string ne peut pas acceuillir &String
```

```
    ma_fonction(b);
}
```

Voici à quoi ressemble en mémoire le slice. *s* est une référence (slice) de *s1* qui est un String.

Les slices sont donc des types à part entière qui ont leur propres fonctions et méthodes. Vous pouvez trouver la doc ici

En savoir plus

- *Le language de programmation Rust* - Qu'est ce que la possession ?
- *Le language de programmation Rust* - Les références et les emprunts
- *Le language de programmation Rust* - Le type slice
- Les animations de ce chapitre viennent de l'article □□□ Rust Visualized: The Stack, the Heap, and Pointers par ender minyard

Vecteurs et HashMap

Vecteurs

Les vecteurs sont des listes à taille variable

```
fn main() {
  // Je peux créer un vecteur mutable (qui peut être modifié) comme ceci :
    let mut ma_liste = vec![1,2,3,4,5,6,7,8,9];
    println!("Ma liste est {:?}", ma_liste);

    // Je peux ensuite ajouter des éléments
    ma_liste.push(10);

    // Récupérer la longueur ou n'importe quel élément
    println!("Ma liste a une longueur de {} et son premier élément est {}", ma_liste.len(), ma_liste[0]);

    // Supprimer le dernier élément
    ma_liste.pop();

    // Ou supprimer un élément d'index précis (dans ce cas, le premier)
    ma_liste.remove(0);

    println!("A présent ma liste est {:?}", ma_liste);
}
```

Les HashMap

Un vecteur stoque les données par index (0, 1, 2, 3, 4, 5, etc) tandis qu'une hashmap stoque les données par clé ("foo", "hello")

```
use std::collections::HashMap;
```

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

    // On peut ensuite insérer des données
    dictionnaire.insert("foo", "bar");
    dictionnaire.insert("hello", "world");
    dictionnaire.insert("life", "42");

    // On peut récupérer la valeur correspodante à une clé avec .get
    println!("Foo: {}", dictionnaire.get("foo").unwrap());

    // Pour supprimer un élément on utilise .remove
    dictionnaire.remove("life");

    // Mais on peut aussi itérer une hashmap
    for (key, val) in dictionnaire.iter() {
        println!("{}: {}", key, val);
    }
}
```

En savoir plus

- *Rust By Example* - Vectors
- *Rust By Example* - HashMap
- *docs.rs* - Vec
- *docs.rs* - HashMap

Enums et gestion d'erreurs (Result et Option)

Rust exige toujours de gérer les cas où il y aurait une erreur d'une manière ou d'une autre. Toute commmande qui pourrait donner une erreur rendera un type Result<> et toute commande qui pourrait ne donner rien retournera un type Option<>.

Il y a plusieurs manières de gérer ce genre de types :

- Avec `.unwrap()`, non-recommendé. Fait planter le programme quand il y a une erreur (pour les Result) ou quand il n'y a rien (dans les Option)
- Avec `.expect("message")`, fait également planter le programme, mais indique un message d'erreur personalisé.
- Avec `.unwrap_or(valeur)` ou encore `.unwrap_or_else(|c| { valeur })`, dans le cas d'un None ou d'une erreur, il va retourner une valeur par défault à la place
- Avec `match` ou `if let` pour faire des choses pour chaque cas ou pour définir une valeur par défault
- Avec `?` dans une fonction et va donc créer une erreur au niveau du retour de la fonction en elle même

```
// Cette fonction peut retourner un booléen si tout se passe bien et peut planter avec un &str
fn beverage_checker(beverage: &str) -> Result<bool, &str> {
    if beverage == "water" {
        Ok(true)
    } else if beverage == "coffee" {
        Err("Wait, you actually drink that???")
    } else {
        Ok(true)
    }
}

fn main() {
    let mut status = beverage_checker("water").expect("Le développeur de ce programme n'aimais vraiment pas cette boisson");
    println!("Le status de 'water' est {}", status);

    // Ici on va remplacer l'erreur par un false avec unwrap_or
```

```
    let status = beverage_checker("coffee").unwrap_or(false);
    println!("Le status de 'coffee' est {}", status);

    // On peut aussi utiliser unwrap_or_else pour exécuter du code dans une closure en plus de retourner une autre valeur
    let status = beverage_checker("coffee").unwrap_or_else(|err| {
        println!("Message: {}", err);
        false
    });
    println!("Le status de 'coffee' est {}", status);

    // Sinon on peut encore utiliser un match dans lequel on exécute du code
    match beverage_checker("coffee") {
        Ok(s) => println!("Le status de 'coffee' est {}", s),
        Err(e) => println!("Voici le message du développeur : {}", e)
    }

    // On peut aussi faire similaire à précédemment avec un if let
    if let Ok(s) = beverage_checker("coffee") {
        println!("Le status de 'coffee' est {}", s);
    } else {
        println!("Le développeur n'aime pas le café");
    }

    // Ou encore utiliser match comme on la fait avec unwrap_or ou unwrap_or_else
    let status = match beverage_checker("coffee") {
        Ok(s) => s,
        Err(e) => {
            println!("Voici le message du développeur : {}", e);
            false
        }
    };
    println!("Le status de 'coffee' est {}", status);

    // Cette ligne va planter le programme :
    let status = beverage_checker("coffee").expect("Le développeur de ce programme n'aimais vraiment pas cette boisson");
    println!("Le status de 'coffee' est {}", status);
}
```

```
    // Cette ligne planterais le programme sans message d'erreur
    status = beverage_checker("coffee").unwrap();
    println!("Le status de 'coffee' est {}", status);
}
```

Enum

Result et Option sont tous les deux des enums que voici :

```
pub enum Result<T, E> {
    Ok(T),
    Err(E),
}

pub enum Option<T> {
    None,
    Some(T),
}
```

Un enum permet d'énumérer plusieurs variantes. Voici un exemple d'enum personalisé

```
enum IpAddrKind {
    V4,
    V6,
}

fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;
}
```

Maintenant on peut aussi avoir des valeurs dans un enum

```
enum IpAddrKind {
    V4(String),
    V6(String),
}

fn main() {
```

```
    let four = IpAddrKind::V4("127.0.0.1".to_string());
    let six = IpAddrKind::V6("::1".to_string());
}
```

Et quand on ne sait pas encore les types qui vont être compris dans notre enum on peut utiliser cette notation :

```
enum IpAddrKind<A, B> {
    V4(A),
    V6(B),
}

fn main() {
    let four: IpAddrKind<String, String> = IpAddrKind::V4("127.0.0.1".to_string());
    let six: IpAddrKind<String, String> = IpAddrKind::V6("::1".to_string());
}
```

On peut aussi gérer les enums avec des `match` ou encore avec des `if let`

```
enum IpAddrKind<A, B> {
    V4(A),
    V6(B),
}

fn main() {
    let four: IpAddrKind<&str, &str> = IpAddrKind::V4("127.0.0.1");
    let six: IpAddrKind<&str, &str> = IpAddrKind::V6("::1");

    // En utilisant match
    match four {
        IpAddrKind::V4(ip) => println!("ipv4: {}", ip),
        IpAddrKind::V6(ip) => println!("ipv6: {}", ip)
    }

    match six {
        IpAddrKind::V4(ip) => println!("ipv4: {}", ip),
        IpAddrKind::V6(ip) => println!("ipv6: {}", ip)
    }

    // En utilisant if let
    if let IpAddrKind::V4(ip) = four {
```

```
        println!("ipv4: {}", ip);
    }

    if let IpAddrKind::V6(ip) = six {
        println!("ipv6: {}", ip);
    }
}
```

Structures, implémentations et traits

> *No Boilerplate →* ***Building a space station in Rust (Simple Rust patterns) [RUST-8]***

Structure

Un `struct` ou *structure* est un type de donnée personnalisé qui permet de rassembler plusieurs propriétés. (C'est un peu comme un "objet" dans la POO)

```
// Création d'une première structure "Chapitre"
struct Chapitre {
  titre: String,
  numero: u64,
  contenu: String
}

// Création d'une deuxième structure "Livre" qui contient entreautre, une liste de "Chapitre"
struct Livre {
  titre: String,
  auteur: String,
  contenu: Vec<Chapitre>
}

fn main() {
    // Création d'un chapitre et d'un livre
    let chapitre = Chapitre {
        titre: "Mon super petit chapitre".to_string(),
        numero: 1,
        contenu: "voilà, le chapitre est fini...".to_string()
```

```
    };

    let livre = Livre {
        titre: "Livre court".to_string(),
        auteur: "Anonyme".to_string(),
        contenu: vec![chapitre]
    };

    println!("Mon livre est '{}' et le premier chapitre est '{}'", livre.titre, livre.contenu[0].titre);
}
```

Implémentation et trait

Un *trait* sert à définir un comportement partagé de manière abstraite par plusieurs types.

Une *implémentation* permet de grouper des méthodes et fonctions pour un type donné. Par exemple sur un *struct*, il va y avoir à la fois les propriétés du *struct* et les méthodes de son implémenation. La différence entre une fonction et une méthode c'est que:

- Une méthode prends comme argument elle même (`self`) et on l'utilise en finissant par `.nom_de_la_methode()`
- Une fonction n'a pas besoin d'accéder au contenu (donc pas de `self`), et on l'utilise en finissant par `::nom_de_la_fonction()`

Ainsi une implémentation peut être utilisée pour implémenter un *trait* sur un type.

```
// Le trait "Resumable" dit que tout type implémentant ce trait doit avoir la méthode "resumer"
pub trait Resumable {
    // Le &self est important car sinon il devient propriétaire de lui même et on ne peut donc plus l'utiliser
    fn resumer(&self) -> String;
    // On utilise &mut pour pouvoir faire en sorte que l'objet puisse modifier une instance de lui même
    fn anonymiser(&mut self);

    // L'absence de self signifie que ceci est une fonction et non pas une méthode
    fn new(titre: &str, lieu: &str, auteur: &str, contenu: &str) -> ArticleDePresse;
}

// La structure ArticleDePresse contient 4 propriétés de type String
pub struct ArticleDePresse {
    pub titre: String,
```

```
    pub lieu: String,
    pub auteur: String,
    pub contenu: String,
}

// Implémente le trait Resumable pour le type ArticleDePresse
impl Resumable for ArticleDePresse {
    // Implémente la méthode "résumer"
    // Le &self est important car si on met juste 'self' il devient propriétaire de lui même et on ne peut donc plus l'utiliser
    fn resumer(&self) -> String {
        format!("{}, par {} ({})", self.titre, self.auteur, self.lieu)
    }

    // On utilise &mut pour pouvoir faire en sorte que l'objet puisse modifier une instance de lui même
    fn anonymiser(&mut self) {
        self.auteur = "Anonyme".to_string();
        self.lieu = "Inconnu".to_string();
    }

    // Ceci ne prends pas self, donc c'est une fonction et pas une méthode
    // Cette fonction va créer un élément de type ArticleDePresse à partir de valeurs données
    fn new(titre: &str, lieu: &str, auteur: &str, contenu: &str) -> ArticleDePresse {
        ArticleDePresse {
            titre: titre.to_string(),
            lieu: lieu.to_string(),
            auteur: auteur.to_string(),
            contenu: contenu.to_string()
        }
    }
}

// Crée ArticleDePresse dans la variable "article" et exécute la méthode "resumer"
fn main() {
    // On va utiliser la méthode anonymiser plus tard donc on met "mut" sur notre variable pour pouvoir la modifier
    // On utilise notre fonction new avec ::
    let mut article = ArticleDePresse::new("Mon titre", "Liège", "John Doe", "ayayyayayayaya");

    // On utilise nos méthodes avec .
```

```
    // On modifie notre article pour modifier les valeurs lieu et auteur
    article.anonymiser();

    // Enfin on retourne le résumé de notre article
    println!("{}", article.resumer());
}
```

Mais il est aussi possible de faire la même chose, sans trait.

```
// La structure ArticleDePresse contient 4 propriétés de type String
pub struct ArticleDePresse {
    pub titre: String,
    pub lieu: String,
    pub auteur: String,
    pub contenu: String,
}

// Implémente le type ArticleDePresse
impl ArticleDePresse {
    fn resumer(&self) -> String {
        format!("{}, par {} ({})", self.titre, self.auteur, self.lieu)
    }

    fn anonymiser(&mut self) {
        self.auteur = "Anonyme".to_string();
        self.lieu = "Inconnu".to_string();
    }

    fn new(titre: &str, lieu: &str, auteur: &str, contenu: &str) -> ArticleDePresse {
        ArticleDePresse {
            titre: titre.to_string(),
            lieu: lieu.to_string(),
            auteur: auteur.to_string(),
            contenu: contenu.to_string()
        }
    }
}
```

```
// Crée ArticleDePresse dans la variable "article" et exécute la méthode "resumer"
fn main() {
    // On va utiliser la méthode anonymiser plus tard donc on met "mut" sur notre variable pour pouvoir la modifier
    let mut article = ArticleDePresse::new("Mon titre", "Liège", "John Doe", "ayayyayayayaya");

    article.anonymiser();
    println!("{}", article.resumer());
}
```

En savoir plus

- *Le language de programmation Rust* - Définir et instancier des structures
- *Le language de programmation Rust* - Définir des comportements partagés avec les traits
- *Rust By Example* - Structures
- *Rust By Example* - Traits
- *Rust By Example* - Implementation

Les fonctions (méthodes et closures)

> La différence entre une méthode et une fonction sera évoquée au chapitre sur les implémentations

Définition d'une fonction

```
fn ma_super_fonction() {
    // Cette fonction ne retourne rien
    println!("Hello World");
}

fn addition(a: i32, b: i32) -> i32 {
    // Cette fonction retourne un i32
    let result = a + b;

    // Il suffit d'indiquer le résultat sans terminer par un point virgule pour retourner une valeur
    result
}

// On peut aussi créer des fonctions génériques en utilisant <T> pour définir n'importe quel type
fn foobar<L,R>(left: L, right: R) -> (Vec<L>,Vec<R>) {
    (vec![left], vec![right])
}
```

Définition et utilisation des closures

Les closures (ou "fermeture" en français) sont des fonctions anonymes qui peuvent être sauvegardées dans une variable et peuvent être passée comme argument à d'autres fonctions.

Contrairement aux autres fonctions les closures peuvent également capturer des variables du contexte dans lequel elle est exécutée.

```
fn main() {
    let addition = |x, y| {
        println!("Une addition est en cours...");
        x + y
    };
    println!("1 + 1 = {}", addition(1, 1));
}
```

Certaines fonctions vous obligeront à utiliser des closures dans leurs arguments. Cela est représenté par `Fn`, `FnOnce` ou encore `FnMut`

Egalement quand on déplace des données vers une nouvelle tâche, on peut utiliser le mot clé `move` en face de la closure pour donner l'appartenance de toutes les variables qu'elle utilise à la fonction.

```
fn main() {
    let hello = String::from("Hello ");

    // Les {} sont facultatif dans la closure quand il n'y a qu'une seule instruction comme ici
    // L'appartenance de y passe à notre closure
    let concat = move |x| hello + x;

    // On ne peut donc plus utiliser y
    println!("Cette commande génère une erreur, commentez là pour supprimer l'erreur: {}", y);

    // Utilisons notre closure
    println!("{}", concat("World"));
}
```