

# Starter Lab Rust

23 Mars 2023



Developers  
Group Dijon



Adrien Gras  
Mathias Da Costa



**Developers  
Group Dijon**  
GDG



**BPCE**  
SOLUTIONS INFORMATIQUES



Developers  
GroupDijon



# Starter LAB

## Rust

Quelques mots sur le langage

## Le juste prix

Création d'un premier programme en Rust pour implémenter le jeu du juste prix, et appréhender les concepts et syntaxes de base du langage

---

# Home LAB

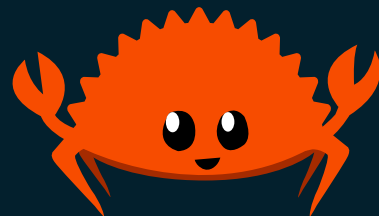
## rpas

Implémentation d'un gestionnaire de mots de passe CLI qui reprend les bases de la commande linux « pass »

# Rust

# Rust en quelques mots

- **Remplaçant du C/C++**
  - Performances des langages bas-niveau
  - Concepts des langages haut-niveau
  - Sécurité
- **Utilisé principalement pour**
  - Des applications systèmes, des backends, des CLI
  - Du web avec WASM
  - Des services réseaux
  - De l'informatique embarqué
  - De la crypto
  - Des apps mobiles \*
- **Créé en 2006 par Graydon Hoare, puis repris à partir de 2010 par la fondation Mozilla**
- **Version actuelle : 1.67.1**
- **Sa mascotte : Ferris**



## ...Avant de commencer



### **Les phases d'exercices sont timées**

Pour avancer tous ensemble et rester dans le temps imparti, les phases sont chronométrées, mais corrigées après le temps d'exercice.



### **Posez des questions**

Il n'y a pas de questions stupides ! N'ayez pas peur de poser des questions et de vous renseigner. Vous pouvez aussi aider votre voisin si vous avez une solution qu'il n'a pas.



### **Allez chercher de la documentation**

Rust est un langage très bien documenté avec une communauté très active, n'ayez pas peur d'aller chercher une solution sur internet !

**Le juste prix**



# Setup

- Exécuter un `rustup update`.
- Cloner le dépôt github :  
[git@github.com:developers-group-dijon/2023-codelab-rust.git](https://github.com/developers-group-dijon/2023-codelab-rust.git)
- Se rendre dans le dossier `starter_lab` puis `le_juste_prix`.
- Lancer VSCode dans le dossier.



```
rustup update
```

```
cd /srv && git clone git@github.com:developers-group-dijon/2023-codelab-rust.git
```

```
cd ./2023-codelab-rust/starter_lab/le_juste_prix
```

```
code .
```

# Lancement

- Dans un terminal, lancer la commande `cargo run`.



```
cargo run
```



30 sec

# Lancement

- Dans un terminal, lancer la commande `cargo run`.



# Les macros

- Écrit du code qui écrit du code : métaprogrammation.
- Injecte le code « macro » au moment de la compilation.
- Permet d'éviter la réplication de code, ou des sets de fonctions « utilitaires ».
- Permet d'ajouter des comportements à des fonctions, des structures, des traits, etc.



```
// macro déclarative

// demande au compilateur de considérer le bloc de
// code suivant comme "valide"
todo!();

// imprime sur la sortie standard
println!("Hello World!");

// formate une chaîne de caractère
format!("Date du jour: {}", Utc::now())
```



```
// macro dérivative

#[derive(Serialize, Deserialize)]
struct User {
    login: String,
    password: String
}
```

# Déclarons nos variables

- Pour le jeu du juste prix, il nous faut deux variables :
  - `random_number` qui sera le nombre choisi par l'ordinateur entre 1 et 100 (qui ne bougera pas).
    - Pour cette variable, vous pouvez utiliser le résultat de la fonction `generate_random_number_between` qui vous est fournie.
  - `found` qui sera la variable qui détermine si nous sommes ou non encore dans la boucle de jeu : si l'utilisateur a trouvé la valeur aléatoire ou pas (qui sera amenée à changer).



```
// variable qui ne va pas changer (constante)
let <nom>[:<type>] = <valeur>;

// variable qui va changer
let mut <nom>[:<type>] = <valeur>;
```

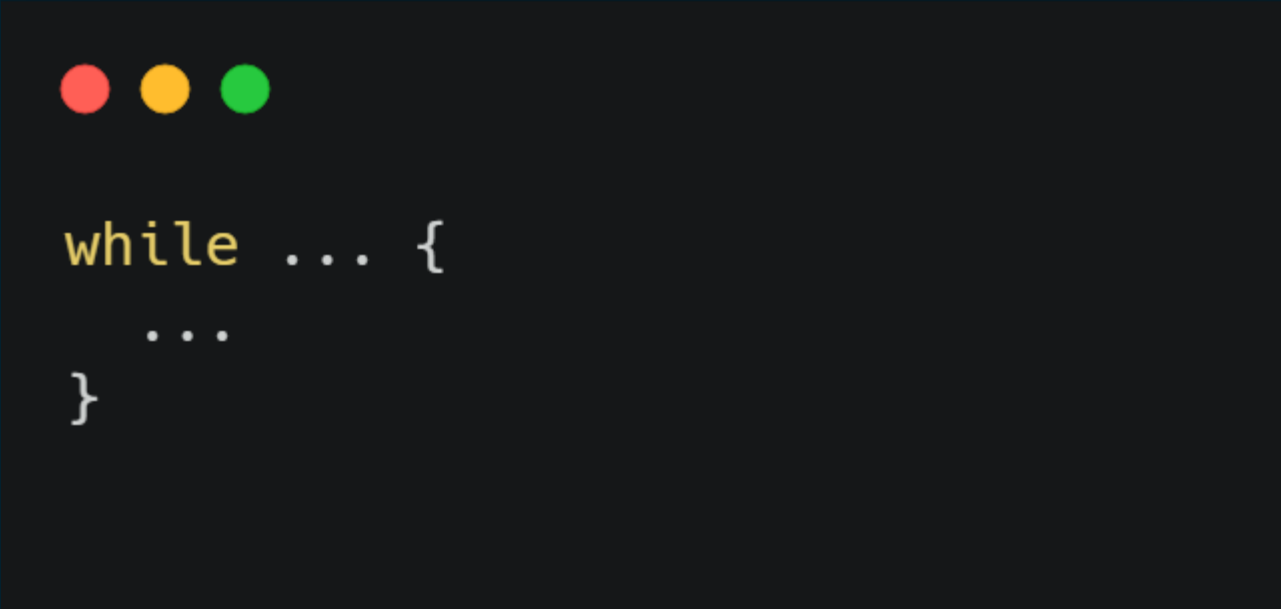
# Déclarons nos variables



```
// récupérer un nombre aléatoire entre 1 & 100  
let random_number = generate_random_number_between(1, 100);  
  
// créer un mutex pour sortir de la boucle de jeu  
let mut found = false;
```

# Créer la boucle de jeu

- Maintenant que les variables initiales du jeu sont posées, il faut poser la boucle de jeu.
- Créer une boucle dont on ne sortira que lorsque l'on aura trouvé le bon chiffre.



```
while ... {  
    ...  
}
```



# Créer la boucle de jeu



```
while !found {  
    // ...  
}
```



# Demander son input à l'utilisateur

- Pour démarrer la boucle de jeu, il faut demander un nombre à l'utilisateur.
- Imprimez la question sur la sortie standard, puis utilisez la fonction `get_input_from_user` pour récupérer sa saisie dans une variable `guess`.



```
Vous avez déjà un texte d'imprimé  
sur la sortie standard en haut de  
la fonction main ;)
```

# Demander son input à l'utilisateur



```
println!("Quel est le juste prix ?");
```

```
let guess = get_input_from_user();
```

# Assainir la saisie utilisateur

- La valeur saisie par l'utilisateur est une chaîne de caractères, or il nous faut une valeur numérique pour la comparaison.
- De plus, une saisie utilisateur peut avoir des espaces involontaires.
- Nous allons commencer par « trimer » l'entrée utilisateur et ranger cette chaîne « trimée » dans une variable.



Le type `String` possède une méthode `trim`.

# Transformer la saisie utilisateur en nombre



```
let trimmed = guess.trim();
```

# Transformer la saisie utilisateur en nombre



```
let trimmed: &str
```



# &str, String, wtf ?

- `String` représente une chaîne de caractères.
- Une chaîne de caractères est en fait un tableau (Vecteur) de caractères.
- Un caractère se représente dans la mémoire comme une suite de 8 bit, soit un type `u8`.
- `&str` est donc un pointeur direct vers la valeur contenue dans une chaîne déclarée dynamiquement.

```
// en interne, une String est un vecteur
// (tableau chaîné en mémoire)
// d'entier sur 8bits, qui constituent
// des caractères.
pub struct String {
    vec: Vec<u8>,
}

// de manière interne, Rust va déclarer un
// vecteur de u8 dans la mémoire de la taille
// de la chaîne déclaré dynamiquement (directement
// entre quote), et retourner le pointeur
// direct vers ce vecteur.
//
//          v-- (& = référence/pointeur)
// d'où le &str <-- (str = marqueur de chaîne dynamique)
let my_str = "Hello world ! 🦀";
```

# Transformer la saisie utilisateur en nombre

- Maintenant que nous avons nettoyé la saisie utilisateur, il nous faut la transformer en « nombre » pour comparaison.
- Utilisez la fonction `parse()` sur votre variable trimée pour transformer votre `&str` en `u32`.



Il existe deux manière de faire !

ChatGPT ou StackOverflow pourront vous aider ;)

# Transformer la saisie utilisateur en nombre



```
// les deux manières de faire
```

```
// inférence par type de variable
```

```
let parsed: u32 = trimmed.parse().unwrap();
```

```
// inférence par syntaxe turbofish
```

```
let parsed = trimmed.parse::<u32>().unwrap();
```



## Et si...

- Et si on essayait de rentrer un nombre invalide, par exemple « abc » ?

## Et si...

- Et si on essayait de rentrer un nombre invalide, par exemple « abc » ?



# La gestion des erreurs

- En Rust il n'y a pas d'exception.
- Pour gérer les erreurs, et les traiter proprement, on encapsule un résultat incertain dans un `Result`.
- Un `Result` est une énumération, c'est-à-dire un ensemble de possibilités.
- Une fois le code donnant le résultat incertain, la variable contenant un `Result` sera soit `Ok(T)` en cas de réussite, soit `Err(E)` en cas d'erreur.
- Un `Result` doit être traité ; le compilateur n'autorisera pas le build si un `Result` n'est pas traité correctement.



```
// Résultat possible incertain :  
// - Soit T, le résultat attendu en cas de succès  
// - Soit E, l'erreur produite lors de l'opération  
enum Result<T, E> {  
    Ok(T),  
    Err(E)  
}
```

# La gestion des erreurs

- Pour récupérer le résultat contenu dans un `Result`, on peut utiliser `unwrap()` qui retourne le bon résultat si il est correct, ou plante le programme si il est en erreur.
- `unwrap()` est pratique mais plante le programme, on peut utiliser la syntaxe `if let...` pour traiter les erreurs proprement.

```
let my_result = op_risky().unwrap();  
// va ranger le résultat de op_risky si le  
// résultat est correct, ou planter le programme  
// si c'est une erreur
```

```
let uncertain_result = op_risky();  
  
if let Err(error) = uncertain_result {  
    // traiter l'erreur  
}  
  
// on peut utiliser unwrap car on a déjà  
// validé que ce n'était pas une erreur  
let certain_result = uncertain_result.unwrap();
```

```
let uncertain_result = op_risky();  
  
if let Ok(safe_result) = uncertain_result {  
    // traiter uniquement le cas de succès  
}
```

# Sécuriser la boucle de jeu

- Utiliser la syntaxe `if let...` pour ne traiter l'entrée utilisateur que si elle est valide (elle est bien parsée en type numérique).
  - Vous pouvez imprimer le résultat à l'utilisateur pour confirmer sa saisie.
- Dans le cas inverse, on ne traite simplement pas le cas et on recommence la boucle.
- Vous pouvez aussi ajouter un message d'erreur en cas de saisie erronée.

```
if let ... {  
    ...  
} else {  
    ...  
}
```

```
let uncertain_result = op_risky();  
  
if let Ok(safe_result) = uncertain_result {  
    // traiter uniquement le cas de succès  
}
```

# Sécuriser la boucle de jeu



```
if let Ok(num) = guess.trim().parse::<u32>() {  
    print!("Vous proposez : {num}");  
  
    // ...  
} else {  
    println!("ERREUR: saisie invalide !");  
}
```

# Comparer les deux valeurs

- Créez le code qui va comparer la valeur `random_value` (valeur de l'ordinateur) avec `num` (valeur saisie par l'utilisateur et transformée).
- Si `random_value > num` : C'est plus !
- Si `random_value < num` : C'est moins !
- Si `random_value == num` : C'est gagné !
  - Dans ce cas il faudra sortir de la boucle de jeu

## Comparer les deux valeurs



```
println!("Vous proposez : {num}");

if random_number > num {
    println!("C'est plus !");
} else if random_number < num {
    println!("C'est moins !");
} else {
    println!("C'est gagné !");
    found = true;
}
```



# Le pattern matching


- Rust possède un système de pattern matching pour effectuer des actions selon une valeur donnée par une énumération.
- Ce pattern matching doit traiter tous les cas possibles, ou en définir un par défaut (avec `_`).
- Par exemple, il peut être appliqué aux `Result` pour traiter, et le cas correct, et le cas d'erreur.

```
match ... {  
    pattern => ...,  
    pattern => ...,  
    _ => ...,  
}
```

```
match guess.trim().parse::<u32>() {  
    Ok(value) => ...,  
    Err(error) => ...,  
}
```

# Comparer les deux valeurs : pattern matching

- Utilisez la structure `match` pour comparer `num` à `random_number` en utilisant la fonction `comp(&value)` sur `num`.
- `comp` peut avoir trois valeurs :
  - `Ordering::Less`
  - `Ordering::Greater`
  - `Ordering::Equal`



```
match ... {  
  pattern => ...,  
  pattern => ...,  
  _ => ...,  
}
```

## Comparer les deux valeurs : pattern matching

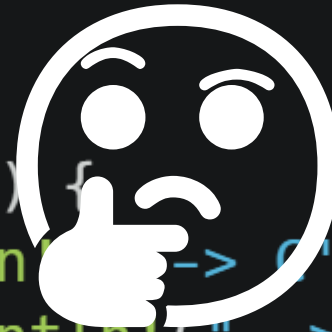


```
match num.cmp(&random_number) {  
    Ordering::Less => println!(" -> C'est plus !"),  
    Ordering::Greater => println!(" -> C'est moins !"),  
    Ordering::Equal => {  
        println!(" -> Gagné !");  
        found = true;  
    }  
}
```

???



```
match num.cmp(&target) {  
  Ordering::Less => println!(" -> C'est plus !"),  
  Ordering::Greater => println!(" -> C'est moins !"),  
  Ordering::Equal => {  
    println!(" -> Gagné !");  
    found = true;  
  }  
}
```




# Borrow, mutable borrow, ownership

- Rust possède un système de références permettant de garantir un code sans fuite mémoire.
- Une référence est un lien direct vers l'adresse mémoire d'une valeur, et non le contenu de la valeur directe.
- Cette garantie est mise en place par le compilateur qui va valider quelques règles du « borrow » sur chaque référence dans le code.



# Borrow et ownership : le magasin de livres



```
let neuromancer = Book {};  
// le propriétaire du contenu du livre est neuromancer.  
  
alice_look_at(&neuromancer);  
// alice regarde le livre, consulte le contenu, mais sans  
// l'acheter, c'est un "borrow".  
//  
// neuromancer est toujours propriétaire du contenu du livre.  
  
bob_look_at(&neuromancer);  
// bob regarde le livre, consulte le contenu, mais sans  
// l'acheter, c'est un autre "borrow".  
//  
// neuromancer est toujours propriétaire du contenu du livre.  
  
sell_to_charly(neuromancer);  
// cette fois-ci, neuromancer change de propriétaire, car ce  
// n'est pas la référence marqué par "&" qui est donné, mais  
// bien directement la valeur.  
//  
// C'est un changement de "ownership", aussi appelé un "move".  
  
dany_look_at(&neuromancer);  
// ERROR : neuromancer ayant changé de propriétaire, il ne peut  
// plus être consulté
```

# Le « mutable borrow »



```
let neuromancer = Manuscript {};  
// neuromancer est propriétaire du contenu du manuscrit du livre.  
  
let ace_books = Editor {};  
let molly = Editor {};  
  
edit(&mut neuromancer, ace_books);  
// ici, ace_books se réserve le droit de réécrire le manuscrit,  
// on dit qu'il fait un "mutable borrow".  
//  
// Rust stipule qu'il ne peut y avoir qu'un seul mutable borrow  
// dans la vie d'une variable.  
  
edit(&mut neuromancer, molly);  
// Error: il ne peut y avoir qu'un seul mutable borrow par variable  
  
sell(neuromancer);  
// neuromancer est vendu, le propriétaire change, son contenu n'est  
// donc plus accessible.  
  
edit(&mut neuromancer, ace_books);  
// neuromancer est déjà vendu (changé de propriétaire), son contenu  
// ne peut plus être modifié
```

# Le « borrow » : les règles

- Il peut y avoir autant de références (emprunt) que voulu à une variable donnée.
- Il ne peut y avoir qu'une seule référence mutable par variable donnée.
- Il ne peut pas y avoir d'emprunt mutable et immutable en même temps.
- Une fois qu'une variable change de propriétaire (passage direct: « move »), elle n'est plus utilisable.





# Lancement

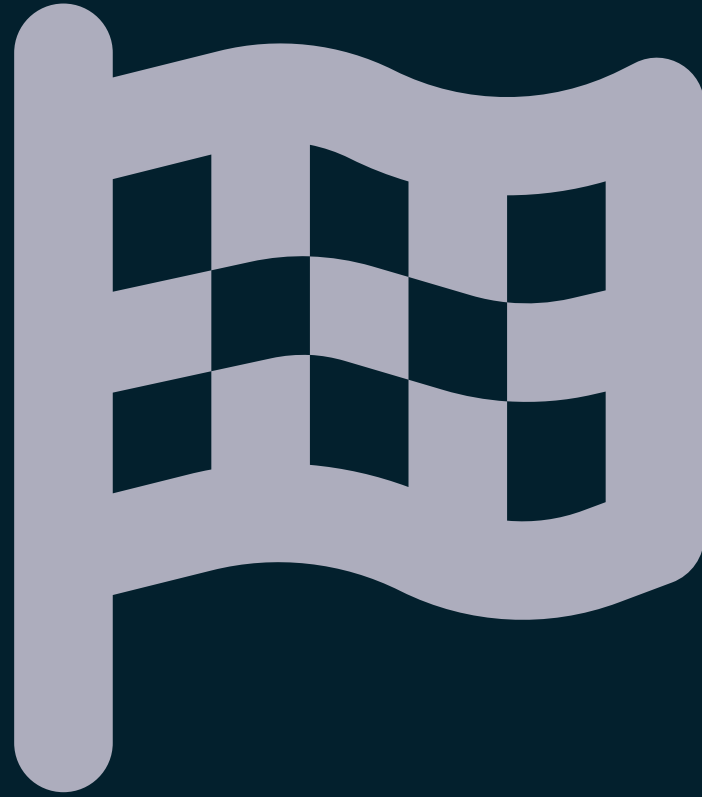
- Dans un terminal, lancer la commande `cargo run`.



```
cargo run
```



30 sec



**Bien joué !**

# Home LAB

# rpas

- rpass est une réimplémentation « libre » du gestionnaire de mots de passe « pass » linux.
- Il permettra d'ajouter, de lister et de retirer des mots de passe du gestionnaire.
- Il permettra aussi d'insérer un mot de passe dans une suite de commandes.
- Enfin, il permettra de générer des mots de passe sécurisés.



Usage: rpass [OPTIONS] <COMMAND>

## Commands:

list	List all the password stored in the DataStore
init	Initializes a new DataStore
add	Adds a new password to the DataStore
delete	Delete a given password from the DataStore
dump	Dumps a given password into standard output
generate	Generates a new strong password and stores it
help	Print this message or the help of the given subcommand(s)

## Options:

-m, --master-password <MASTER_PASSWORD>	master password to unlock the DataStore
-h, --help	Print help
-V, --version	Print version

## On ne vous laisse pas tomber 🤝

- Si vous avez le moindre souci, ou que vous voulez échanger en dehors de nos meetups ; rejoignez-nous sur le serveur Discord du developers group !



<https://discord.gg/Pp6pHUUBXd>

## Rejoignez la crab rave !

- Vous pouvez retrouver de quoi revoir et approfondir les notions de ce starter lab, et de quoi continuer à vous amuser avec Rust sur cette page.



# Merci à tous !



Developers  
Group Dijon

