Rust

Joel IMBERGAMO GUASCH

INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES TOULOUSE

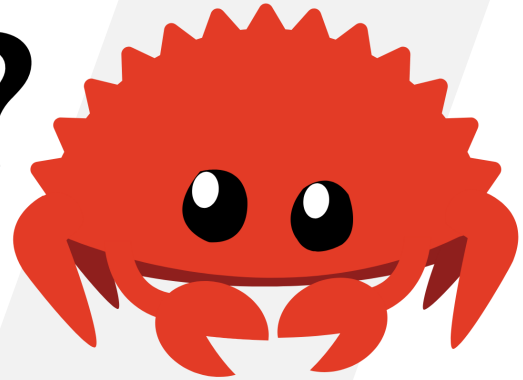
```rust
fn main() {
    let max: u32 = prime_functions::functions::get_number_from_user();
    println!("Searching primes until {}", max);

    let mut primes: Vec<u32> = vec![2]; // base known primes;

    for i :u32 in 2..max {
        prime_functions::functions::add_to_vec_if_prime(&mut primes, num: i);
    }

    println!("Found {}primes in {} num", primes.len(), max);
    println!("Biggest prime: {}", primes.last().unwrap());
}
```

Table de matières

- Qu-est ce que c-est Rust et pourquoi on utilise?
- Cargo & Hello World
- Concepts communs de la programmation
- The ownership model
- Structs
- Traits
- Généricité
- Enum and pattern matching
- Traitement d'erreurs
- Modules and crates
- Collections (Vec, Strings)
- Tests automatiques
- Iterators
- Closures

Comment va-t-on fonctionner?

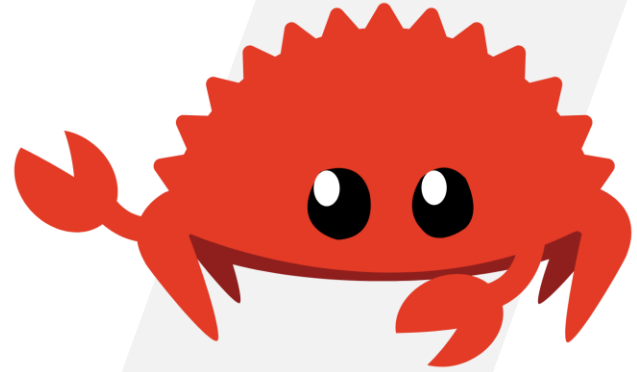
- Ferris va nous guidé sur le chemin pour devenir rustaceans



?

Ne compile pas

Panic!("")



Comportement inattendu

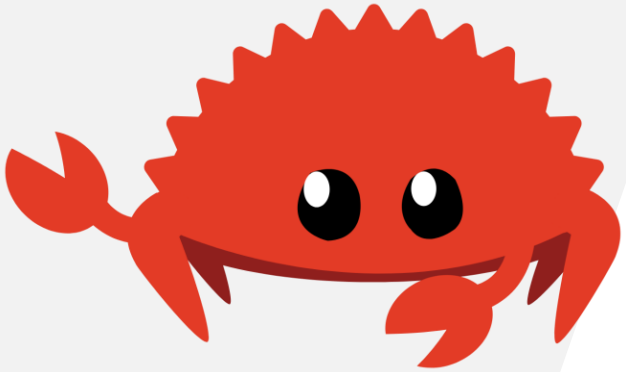
Compétences pas envisagés ici :

1. « Fearless concurrency »
2. Macros!
3. Async
4. Lifetimes
5. The dark side of Rust: Unsafe
6. Smart pointers and heap allocation

Qu-est ce que c-est Rust et pourquoi on utilize?

- Développé en 2006 par Graydon Hoare à Mozilla.
- Bas niveau
- Compilé
- Memory safe
- Sans garbage collector

```c
int main(){
    int * p = malloc(sizeof(int));
    *p = 42;
    int b = sum(p,27);
    printf("Result: %d \n",b );
    return 0;
}
```



```c
int sum(int * a, int b) {
    free(a);
    return *a+b;
}
```

```
Expected outut:
    Result: 69
```

```
Actual outut:
    Result: 27
```

Impossible d'écrire ce programme en Rust car on « ne peux pas libérer la mémoire » mais on aurait une erreur qui se ressemble à cela dans tous les cas:

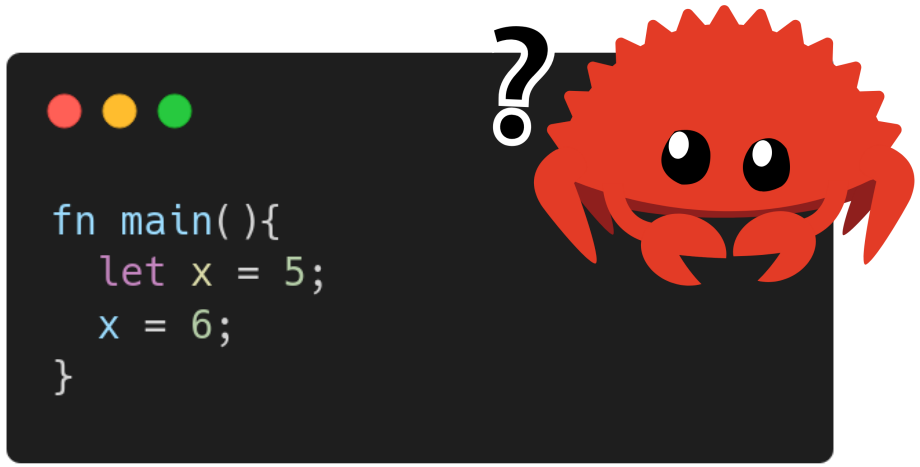
```
   Compiling playground v0.0.1 (file:///playground)
error[E0597]: `value` does not live long enough
  --> src/main.rs:19:31
   |
19 |         my_collection.insert(&value);
   |                               ^^^^^ borrowed value does not live long enough
20 |     });
   |     - `value` dropped here while still borrowed
21 | }
   | - borrowed value needs to live until here

error: aborting due to previous error

error: Could not compile `playground`.

To learn more, run the command again with --verbose.
```

Cargo & Hello World

Cargo: package manager de Rust, équivalent à npm ou pip.

https://play.rust-lang.org/

Concepts communs de la programmation



Variables constantes par défaut !

Constantes (const) substitués au moment de compilation donc elles n'ont pas de temps d'exécution

Le « shadowing » de variables est permis. Très utile pour changer le type.

```rust
fn main() {
    let mut x = 5;
    println!("The value of x is: {}", x);
    x = 6;
    println!("The value of x is: {}", x);
}
```

Types Basiques

Length	Signed	Unsigned
8-bit	`i8`	`u8`
16-bit	`i16`	`u16`
32-bit	`i32`	`u32`
64-bit	`i64`	`u64`
128-bit	`i128`	`u128`
arch	`isize`	`usize`

```rust
fn main() {
  let x= 2.0; //f32

  let y: f16; // f16
}
```

Ce n’est pas nécessaire d’écrire tous les types, Rust essaye de les compléter pour toi.

Types composés

```rust
//this two are equivalent
let a = [3; 5];
let a = [3, 3, 3, 3, 3];


let a: [i32; 5] = [1, 2, 3, 4, 5];
let first = a[0];
let second = a[1];
```

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    // Rust a pattern mathcing donc ce deux
    sont equivalents:
    let five_000 = x.0;
    let six_four = x.1;
    let one = x.2;

    let (five_000, six_four, one ) = x;
}
```

Fonctions

```rust
fn main(a: i32) -> i32 {
    let x = 5;

    let y = {
        let x = 3;
        x + 1
    };

    println!("The value of y is: {}", y);
    a
}
```

Le return est optionnel

Structures de control

```rust
fn main(a: i32) → i32 {
    let number = 3;

    if number < 5 {
        println!("condition was true");
    } else {
        println!("condition was false");
    }

    let number = if condition { 5 } else { 6 };


}
```

```rust
fn main(a: i32) → i32 {
    // while true:
    loop {
        println!("again!");
    }

    let mut number = 3;
    while number == 0 {
        println!("{}!", number);
        number -= 1;
    }

    for number in 1..4 {
        println!("{}!", number);
    }
}
```

Guessing game!

Ownership

```
fn main(){
  let x = String::from("Hello");
  let _a = x;
  println!("{}",x);
}
```

Normes d’Ownership

- **Pour chaque valeur il existe une variable qu’on appelle le propriétaire**
- **Une valeur ne peux avoir qu’un propriétaire à la fois**
- **Quand le propriétaire sort de son scope la valeur est supprimée.**

Normes d’Ownership

- Pour chaque valeur on ne peux avoir qu’un de ces cas au même temps:
  - La variable propriétaire
  - Une référence en droit d’écriture
  - Plusieurs références en lecture

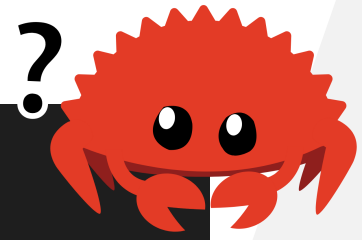
Normes d’Ownership

- Mode par défaut: on transfère le propriété de la valeur


```rust
fn main( ) {
  let mut s = String::from("hello");

  change(s);

  println!("{}",s); --> Err: s moved into change
}

fn change(some_string: String) { --> takes ownership of the value
  some_string.push_str(", world");
}
```

Normes d’Ownership

- Passer par référence non mutable




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

    change(&s);

    println!("{}",s);
}

fn change(some_string: &str) { --> takes a reference
    some_string.push_str(", world"); --> Err: We cannot mutate
}
```

Normes d’Ownership

- Passer par référence mutable (avec droits d’écriture sur la valeur)
- On peut faire cela car le scope de la référence de s est juste la fonction donc quand on fait print le référence a déjà été supprimé

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

    change(&mut s);

    println!("{}",s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Normes d’Ownership

- Clone(): On peut cloner une valeur pour maintenir la possession de la variable. Peu efficace mais utile.
- Rust priorise la efficacité, si une action n’est pass la plus efficace possible c’est au programmateur d’indiqué qu’il veut l’effectuer.

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

    change(s.clone());--> we clone the value

    println!("{}",s); --> we still own it
}

fn change(some_string: String) { --> we take ownership
    some_string.push_str(", world");
}
```

Normes d’Ownership

- The copy trait: si un type implémente la caractéristique « copy » cela nous indique que le coût de copier la valeur est le même ou très similaire à passer la valeur par référence donc rust fait une copie de la valeur au lieu de transférer la propriété.

```
fn main( ) {
    let mut s = 2;

    change(s);--> copies the value

    println!("{}",s); --> we still own it here
}

fn change(some_string: i32) { --> Takes ownership
    some_string += 10;
}
```

Structs

- Permet de stoker plusieurs valeurs dans un groupe de donnés.

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

```
    let mut user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
```

Structs

- On peut implémenter des fonctionnalités à nos structs appelées méthodes si elles utilisent les données de la struct, fonctions associées sinon.

```
struct Piont {
  x: i32,
  y: i32
}

impl Point {
  fn new(x,y) -> Point {
    Point{x,y}
  }
  fn move_to(&mut self, new_x: i32, new_y: i32) -> u32 {
    self.x = new_x;
    self.y = new_y;
  }
}

fn main(){
  let p = Point::new();
  p.move_to(2,3);
}
```

Traits

- On put créer des définitions de caractéristiques et les implémenter pour chaque struct. Cela nous permet après d'avoir des fonctionnalités communes pour des structs différentes.

```
trait Movable {
  fn moove_to(&mut self, new_x, new_y);
}
```

```
struct Piont {
  x: i32,
  y: i32
}

impl Point {
  fn new(x,y) -> Point {
    Point{x,y}
  }
}

impl Movable for Point {
  fn move_to(&mut self, new_x, new_y){
    self.x = new_x;
    self.y = new.y;
  }
}

fn main(){
  let p = Point::new();
  p.move_to(2,3);
}v
```

Traits

- On put créer des définitions de caractéristiques et les implémenter pour chaque struct. Cela nous permet après d'avoir des fonctionnalités communes pour des structs différentes.

```
trait Movable {
  fn moove_to(&mut self, new_x, new_y);
}
```

```
struct Segment {
  start: Point,
  end: Point
}

impl Movable for Segment {

  fn move_to(&mut self, new_x, new_y){
    let diff = (
    self.end.x - self.start.x,
    self.end.y - self.start.y
    );

    let start = Point {x: new_x, y: new_y}
    let end = Point {x: new_x + diff.0, y: new_y + diff.1}
    self.start = start;
    self.end = end;
  }
}


fn main(){
  // some declaration code here
  seg.move_to(2,3);
}
```

Généricité (Genrics)

- On peut créer des fonctions, types, structs etc avec des types génériques. Cela nous permet d'implémenter la fonctionnalité d'une fonction une fois pour plusieurs types au même temps sans avoir à réécrire la fonction pour chaque type.

- On peux aussi ajouter des « contidions » sur les types qu'on accepte grâce aux traits.

```rust
// for i32 dosen't work for u32
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

// for u32 dosen't work for i32
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

Généricité (Genrics)

```rust
// fonctionne pour les deux
// here we are asking for a type that can be ordered and copied
fn largest<T: std::cmp::PartialOrd + std::marker::Copy >(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main(){
  let a: Vec::<i32> = vec![1,2,3,4];
  let b: Vec::<u32> = vec![5,4,3,2];
  let max_i32 = largest(&a);
  let max_u32 = largest(&b);
}
```

Enums

- On put créer des définitions de caractéristiques et les implémenter pour chaque struct. Cela nous permet après d'avoir des fonctionnalités communes pour des structs différentes.

```
fn main() {
    // declaration enum
    enum IpAddrKind {
        V4,
        V6,
    }

    struct IpAddr {
        kind: IpAddrKind,
        address: String,
    }

    let home = IpAddr {
        kind: IpAddrKind::V4,
        address: String::from("127.0.0.1"),
    };

    let loopback = IpAddr {
        kind: IpAddrKind::V6,
        address: String::from("::1"),
    };
}
```

Enums

- On peut aussi associer des données à une des valeurs de l'enum. Et puis faire un match pour les utiliser et faire une disjonction des cas.

```rust
fn main() {
    enum IpAddr {
        V4(u8, u8, u8, u8),
        V6(String),
    }

    let home = IpAddr::V4(127, 0, 0, 1);
    let _loopback = IpAddr::V6(String::from("::1"));

    match home {
        IpAddr::V4(first, second, third, forth) => {
               println!("{}", forth);
        }
        IpAddr::V6(str) => println!("{}", str)
    }
}
```

Enums: Option<T>

En Rust la valeur Null n'existe pas, il faut alors utiliser l'enum Option<T> qui peut contenir soit une valeur soit None.
On peux prendre la valeur directement avec .unwrap() ou .except() qui vont prendre la valeur de Some ou panic!() sinon. Utile en débogage ou si on a mis en place des vérifications pour assurer l'existence de la valeur**. Jamais utiliser unwrap() or expect() dans d'autres cas!!!!!**

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

```rust
fn main() {
  let i = Some(5);
  match i {
    Some(val) => println!("We have the value {}", val),
    None => println!("We don't have a value")
  }
  // will panic without a message if None
  println!("i = {}", i.unwrap());
  // will panic with a message if None
  println!("i = {}", i.expect("Err mesage"));
}
```

Enums: Result<T,Err>, Traitement d'erreurs

En Rust les exceptions n'existent pas, il y a deux options, soit le programme crash directement ce qu'en rust on appelle panic!(); soit on retourne un result si une opération peut échouer.
On peux aussi faire match des erreurs et même les renvoyer comme sortie des fonction avec l'operateur '?' .

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

```rust
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => {
          panic!("Problem opening the file: {:?}", error)
        }
    };
}
```

Modules and Crates

Crates: des librairies externes.

Rust package manager: https://crates.io/

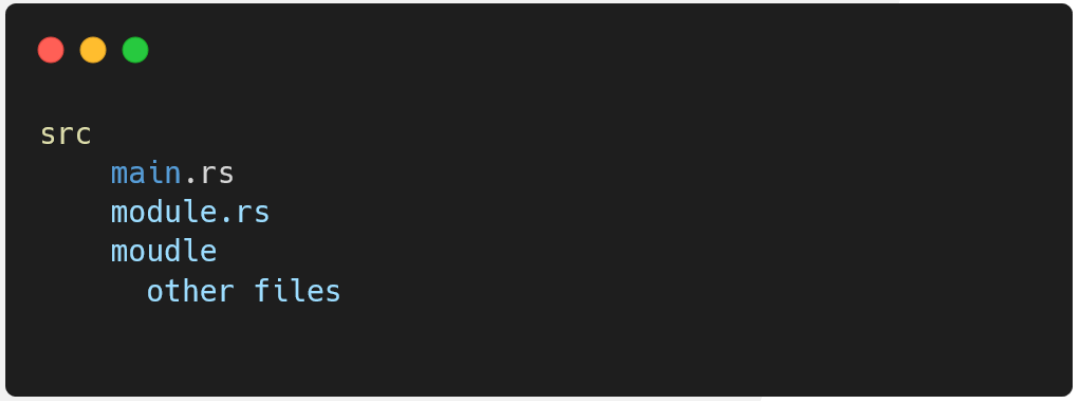
```
[dependencies]
reqwest = { version = "0.11", features = ["json"] }
tokio = "1.7.1"
```

Modules and Crates

Modules: équivalent aux package en Ada.
Permet la division du code en plusieurs modules et fichiers.
On a deux façons de créer des modules:

- Créer un directoire avec le nom du module et un fichier appelée mod.rs dans le directoire.

- Créer un fichier avec le nom du module et un directoire avec le même nom et les autres fichiers dans ce directoire.



Collections

- Vectors: Array the taille variable. Ce ne sont pas des listes, on alloue plus de mémoire que c'est qu'on a besoin et si on en a besoin de plus on alloue une array plus grande et on copie tout. Il faut s'en souvenir de cette pénalité de performance.

- HashMaps: "Dictionaires de python". On ne va pas les utiliser juste sachez qu'ils existent.

```rust
let mut i: Vec::<i32> = vec![1,2,4];
i.push(5);
let z: Option<&i32> = i.get(3); // returns option
let z = i[1]; // panics if out of range
```

Strings

- 2 Types de string: Owned et reference.

- &str : référence à un String. La mémoire ne nous appartient pas il est donc immutable.

- String: On a la propriété de la mémoire on peux donc modifier la valeur de la variable.


```rust
fn main() {
    let str = String::from("Hello");
    // to concatenate we need an owned value + a ref
    let str2: String = str + " world"; //value moved here to str2
    let str3: &str = "Don't panic!()";
    let s = str; // value dosen't exist
}
```

Tests

- Tests automatiques qui permettent valider le bon fonctionnement du code de façon automatique. Lancer la commande « cargo test » pour exécuter tous les tests.

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Iterators

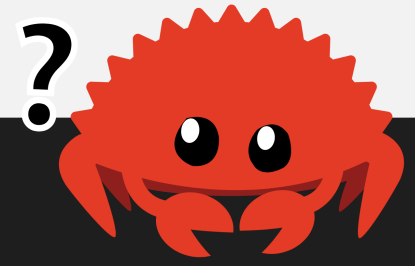
- Un itérateur est un objet qui implémente le trait « iterator », et principalement la méthode .next(). Typiquement une liste ou un tableau mais pas seulement.
- Quand on appelle la méthode .next() on reçoit une option avec la prochaine valeur de l'itérateur.
- L'usage le plus commun c'est un for où on a besoin d'un itérateur pour l'écrire.
- Un des itérateur les plus communs est: 0..5, l'équivalent à range(0,5) en python. Note: 5 exclu.

```rust
#[cfg(test)]
mod tests {
    #[test]
    fn iterator_demonstration() {
        let v1 = vec![1, 2, 3];

        let mut v1_iter = v1.iter();

        assert_eq!(v1_iter.next(), Some(&1));
        assert_eq!(v1_iter.next(), Some(&2));
        assert_eq!(v1_iter.next(), Some(&3));
        assert_eq!(v1_iter.next(), None);
    }
}
```

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

let v1_iter = v1.iter();

for val in v1_iter {
    println!("Got: {}", val);
}
```

Closures

- En rust les fonctions peuvent être utilisés comme un type. Pour faciliter cette utilisation on a les fonctions anonymes ou closures. Fonctions sans nom qui ont comme différence principale avec les fonctions qu'elles capturent les valeurs ou « contexte » qu'elles utilisent.
- Les closures vont inférer le type grâce à la première utilisation, on ne peux pas définir une closure qui fonctionnera avec deux types même si c'est la même syntaxe.




```rust
fn main() {
    let example_closure = |x| x;

    let s = example_closure(String::from("hello"));
    let n = example_closure(5);
}
```

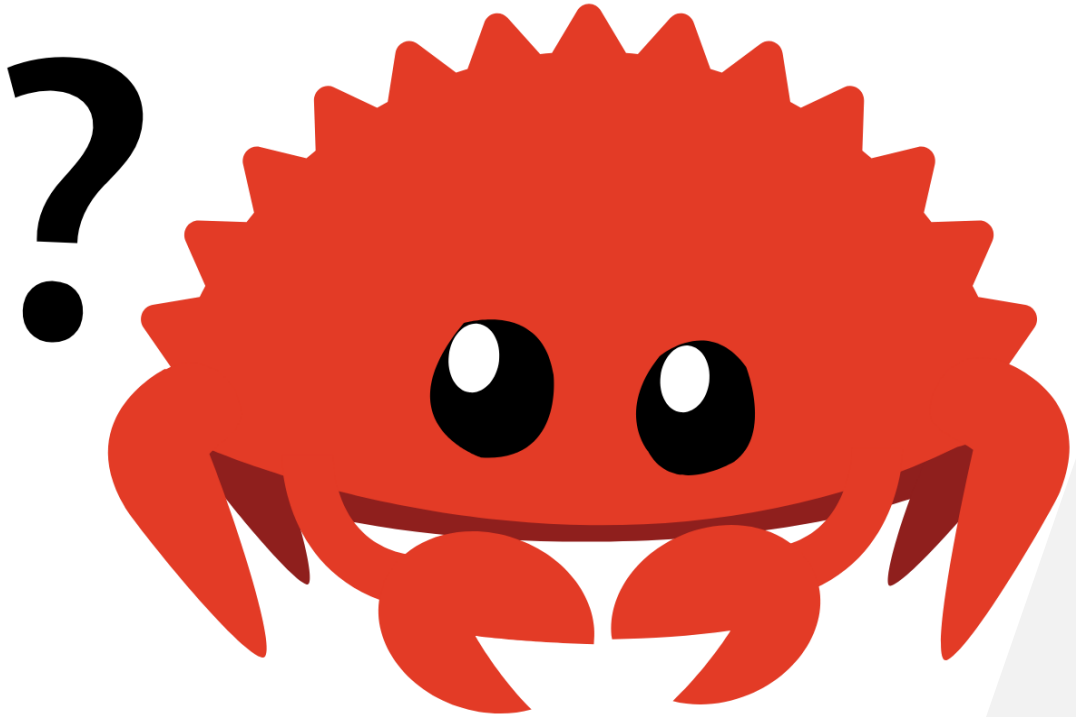
```rust
fn  add_one_v1   (x: u32) -> u32 { x + 1 }
let add_one_v2 = |x: u32| -> u32 { x + 1 };
let add_one_v3 = |x|             { x + 1 };
let add_one_v4 = |x|               x + 1  ;
```

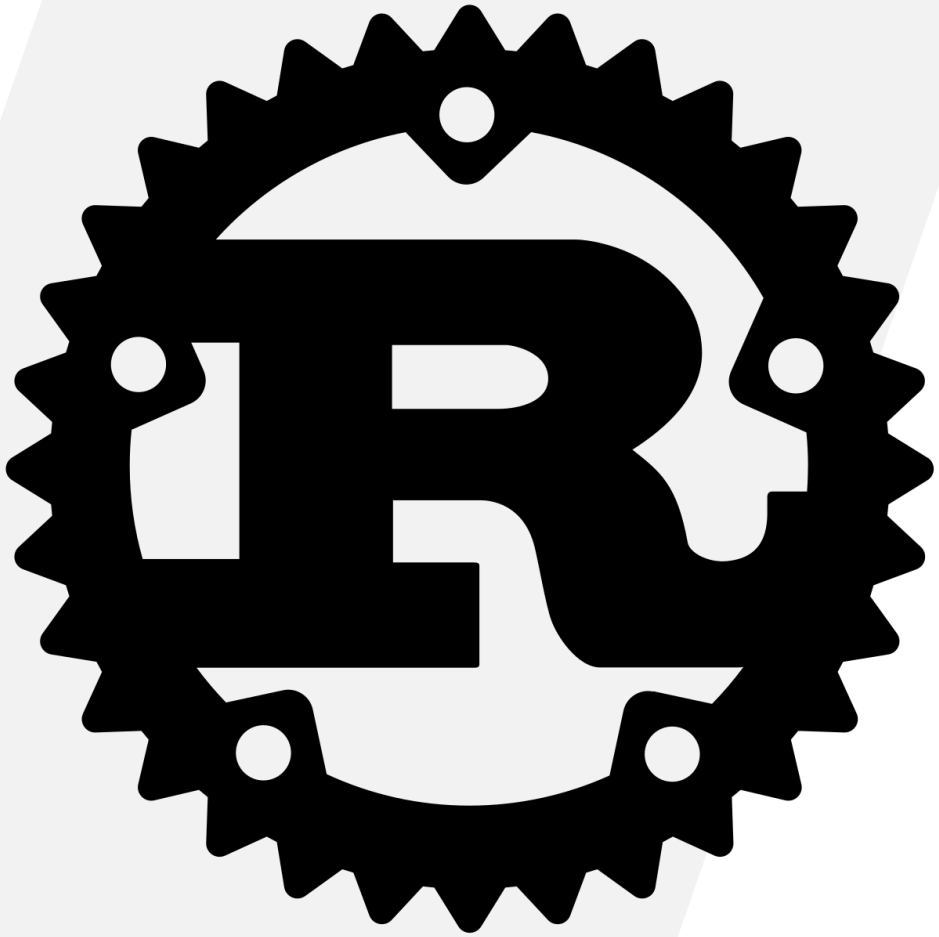
Closures

- On peux passer des fonctions comme arguments d'où c'est parfois utile de déclarer la fonction au moment d'appeler la fonction, d'où l'utilité des Closures.

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];

    let v2: Vec::<i32> = v1.iter().map(|x| x + 1).collect();
    println!("{:?}", v2)
}
```

Questions!( );





Merci!
.iter().map(|y| y + '♡')
.collect();

FAILURE IS NOT AN OPTION<T>

It's a Result<T,E>

Exos!

https://github.com/rust-lang/rustlings