

Introduction à rust

Frédéric Wagner

Novembre 2023

Variables

```
let x = 3;  
let y: u32 = 4;  
let mut z = 5;  
z -= 3;  
let x: f64 = 2.5;
```

- ▶ par défaut en lecture seule
- ▶ tout est fortement typé
- ▶ types de base : *i8, u16, f32, i32, ...*
- ▶ *usize* pour les tailles
- ▶ inférence de type

Fonctions

```
fn addition(x: u32, y: u32) -> u32 {  
    x+y  
}
```

```
let a = 3;  
let b = addition(a, 4);
```

- ▶ pas de fonction variadique
- ▶ les fonctions guident l'inférence de type

Affichage

```
fn main() {  
    let w: String = "world".to_string();  
    println!("hello {}", w);  
}
```

- ▶ println!
- ▶ placeholders
 - ▶ “{}” pour les types simples
 - ▶ “{:?}” pour les types plus complexes

Généricité

```
let mut v1:Vec<u32> = Vec::new();  
v1.push(1);  
v1.push(2);  
let mut v2: Vec<f64> = Vec::new();  
v2.push(3.14);  
  
fn ajout<T>(v: Vec<T>, e: T) -> Vec<T>;
```

- types paramétrés par d'autres types

Macros

```
fn main() {  
    println!("hello", "world");  
    let v = vec![1,2,3,4];  
}
```

- ▶ regarder les !
- ▶ souvent pour remplacer des fonctions variadiques
- ▶ beaucoup plus complexes que les macros C

Conditionnelles

```
if x % 2 == 0 {  
    println!("x est pair");  
}  
let s = if x % 2 == 0 { x / 2 } else { x*3 + 1 };
```

Boucles For

```
let mut s = 0;
for x in 0..10 {
    s += x;
}
assert_eq!(s, 9*10/2);
let v = vec![1,2,3];
for x in &v {
    println!("x vaut {}", x);
}
```


Références

```
let x = u32;  
let r = &x;  
let x2 = *r;  
let mut y = 3.5;  
let ry = &mut y;  
*ry = 4.2;  
assert_eq!(y, 4.2);
```

- ▶ on distingue les références mutables ou non
- ▶ utilisé pour le passage des arguments

Tranches

```
let v = vec![0,1,2,3,4];  
let s = &mut v[2..4];  
s[0] = 3;  
assert_eq!(v, vec![0,1,3,3,4]);
```

- ▶ pointeur vers un bloc mémoire
- ▶ bien mieux qu'en python

Structures, Méthodes

```
struct Point {  
    x: f64,  
    y: f64,  
}  
  
impl Point {  
    fn est_origine(&self) -> bool {  
        self.x == 0 && self.y == 0  
    }  
    fn changer_y(&mut self, nouvel_y: f64) {  
        self.y = nouvel_y  
    }  
}  
  
let p = Point {x: 0., y: 0.54}; assert!(!p.est_origine());  
p.changer_y(0.); assert!(p.est_origine());
```

Types énumérés

```
enum Couleur {  
    Incolore,  
    Colore(u8, u8, u8),  
}
```

```
let c = Couleur::Incolore;
```

```
match c {  
    Couleur::Incolore => println!("pas de couleur"),  
    Couleur::Colore(0, _, _) => println!("pas de rouge"),  
    Couleur::Colore(r, _, _) => println!("le rouge est {}"),  
}
```

- déstructuré avec du pattern matching

Options

```
enum Option<T> {  
    None,  
    Some(T)  
}  
  
let o = Some(3u32);  
println!("o vaut {:?}", o);  
if let Some(contenu) = o {  
    println!("o n'est pas vide et contient {}", contenu);  
}  
  
let contenu_ou_zero = match o {  
    None => 0,  
    Some(contenu) => contenu,  
};
```

- ▶ force à penser aux cas spéciaux
- ▶ toujours du pattern matching

Traits

```
trait PartialEq {  
    fn eq(&self, other: &Self) -> bool;  
    fn ne(&self, other: &Self) -> bool {  
        !self.eq(other)  
    }  
}  
  
fn nb_cibles<T: PartialEq>(s: &[T], cible: &T) -> usize {  
    let mut c = 0;  
    for x in s {  
        if s == cible {  
            c += 1;  
        }  
    }  
    c  
}
```

- ▶ décrit une capacité d'un type
- ▶ permet de contraindre les types abstraits

Itérateurs

```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
}
```

- ▶ deux traits, similaire à python
 - ▶ *Intolterator* : itérable
 - ▶ *Iterator* : itérateur