Breizh C@mp

INTRODUCTION À RUST

Christophe Augier / @tytouf

RUST

"Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety."

rust-lang.org

FEATURES

- zero-cost abstractions
- move semantics
- guaranteed memory safety
- threads without data races
- trait-based generics
- pattern matching
- type inference
- minimal runtime
- efficient C bindings

rust-lang.org

À l'initiative de Graydon Hoare il y a près de 10 ans.

Supporté par la fondation Mozilla et Mozilla Research.

Une communauté de plus en plus active,

Qui prend part à l'orientation du langage via des RFC approuvées par des sous-équipes de contributeurs reconnus.

Version 1.0 annoncée le 15 mai 2015.

=> *Open Source* + *Open Governance*

Réalisations:

- Rustc - le compilateur Rust
- Servo - moteur de navigateur web multi-threadé
- Iron - framework web
- SprocketNES - PoC d'un émulateur NES
- nom - parser combinators library
- ... > 2300 crates on crates.io

GÉNÉRALITÉS

VARIABLE BINDING

```
let x: i32 = 5;

let mut y = 2u8;
y = 10;
```

```
let arr = [1, 2, 3];
let a = [0; 10];

let two_hearts = '💕';
let s = "hello!";

let t = (1, "ABC");
```

Types primitifs: bool, i8, i16, i32, i64, u8, u16, u32, u64, f32, f64, isize, usize, char, [], str, Tuples, ()

Typage statique, typé explicitement ou par inférence.

STRUCTURES

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

let p = Point { x: 1, y: 2 };
```

Règle de nommage est vérifée par le compilateur : CamelCase

ENUMS

```
enum Shape {
    Circle,
    Square,
}

enum Shape {
    Circle(Point, f64),
    Square(Point, Point),
}
```

FONCTIONS

```
fn add(a: i32, b: i32) -> i32 {
  a + b
}

fn sub(a: i32, b: i32) -> i32 {
  return a - b;
}
```

Règle de nommage est vérifée par le compilateur : snake_case

MÉTHODES

```
impl Point {
    fn new(x: i32, y: i32) -> Point {
        Point { x: x, y: y }
    }

    fn add(&mut self, p: &Point) {
      self.x += p.x;
      self.y += p.y;
    }

    // ...
}

let mut p1 = Point::new(0, 0);
let p2 = Point::new(1, 1);
p1.add(&p2);
```

CONDITIONS

```
fn do_res(v: i32) -> i32 {
  let mut res;
  if v > 10 {
    res = 2 * v;
  } else {
    res = 1;
  }
  return res;
}
```

```
fn do_res(v: i32) -> i32 {
  if v > 10 {
    2 * v
  } else {
    1
  }
}

let res = if v > 10 { 2 * v } else { 1 };
```

BOUCLES

```
while i < 10 {
  // ...
  break;
}

loop {
  // ...
}

for i in 0..10 {
  // ...
}
```

PATTERN MATCHING

```
let y = match res {
  0 | 1 => 0,
  e @ 2 ... 10 => e,
  tmp if tmp > 10 && tmp < 20 => 2,
  _ => 3,
}

match res {
  (a, 0) => println!("a = {}", a),
  (a, b) => println!("a = {}, b = {}", a, b),
}
```

```
match shape {
    Circle(center, radius) => {
        draw_circle(center, radius);
    }
    Rectangle(tl, br) => {
      draw_rectangle(tl, br);
    }
}
```

TRAITS

```
trait Period {
  fn minutes(&self) -> TimeChange;
  fn days(&self) -> TimeChange;
}

impl Period for i32 {
  fn minutes(&self) -> TimeChange { /* ... */ }
  fn days(&self) -> TimeChange { /* ... */ }
}
```

```
println!("{}", 2.minutes().from_now());
println!("{}", 5.days().from_now());
```

https://github.com/wycats/rust-activesupport

Certain opérateurs sont en fait implémentés sous la forme de traits.

GENERICS

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

let x: Option<i32> = Some(5);
```

```
fn do_add<T: Add>(a: T, b: T) -> T::Output {
    a + b
}
let c = do_add(1, 3);
```

CLOSURES

```
let plus_one = |x: i32| x + 1;

assert_eq!(2, plus_one(1));
```

```
let num = 5;
let plus_num = |x: i32| { x + num };

assert_eq!(10, plus_num(5));
```

MACROS

```
println!("{} {}", 0, "abc");

let x: Vec<u32> = vec![1, 2, 3];
```

```
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

MEMORY MANAGEMENT

En C/C++ on gère la mémoire manuellement, en allouant et libérant des objets en mémoire.

En Java, C#, Python, Ocaml, la gestion de la mémoire est automatique : la mémoire est libérée quand elle n'est plus utilisée.

Un équilibre entre contrôle, simplicité et sûreté est à trouver.

Rust est orienté contrôle et sûreté.

OWNERSHIP

Au coeur de Rust

```
fn foo() {
    let v = Box::new(5);
}
```

Une variable prend possession de ce à quoi elle est liée.

MOVE SEMANTIC

Il ne peut y avoir qu'un propriétaire.

```
fn foo() {
    let v = Box::new(5);
    let v2 = v;
    println!("{}", v);
}
```

```
<anon>:4:18: 4:19 error: use of moved value: `v`
<anon>:4   println!("{}", v);
```

MOVE SEMANTIC 2

```
fn print(v: Box<i32>) {
    println!("{}", v);
}

let v = Box::new(5);
print(v);
print(v);
```

```
<anon>:8:11: 8:12 error: use of moved value: `v`
<anon>:8     print(v);
```

```
fn print(v: Box<i32>) -> Box<i32> {
    println!("{}", v);
    v
}

let v = Box::new(5);
let v1 = print(v);
let v2 = print(v1);
```

BORROWING

Emprunte la resource au lieu de la posséder

```
fn print(v: &Box<i32>) {
    println!("{}", v);
}

let v = Box::new(5);
print(&v);
print(&v);
```

Les réferences sont immuables

Autant de références que l'on veut

&MUT REFERENCES

Référence muable

```
fn print_and_inc(v: &mut Box<i32>) {
    println!("{}", v);
    *v += 1;
}

let mut v = Box::new(5);
print_and_inc(&mut v);
print_and_inc(&mut v);
```

Une seul référence muable, pas d'autres références.

```
let mut v = Box::new(5);
let r = &v;
print_and_inc(&mut v); // Error
```

```
let mut v = Box::new(5);
let r1 = & mut v;
let r2 = & mut v;  // Error
```

LIFETIME

```
let a = &i32;
{
    let c = 3;
    a = &c;
}
println!("{}", a);
```

```
<anon>:5:14: 5:15 error: `c` does not live long enough
<anon>:5          a = &c;
```

BORROWING & LIFETIME

Fonctionne même au travers de structures

```
struct MyStruct { inner: i32 }

fn get(s: &MyStruct) -> &i32 {
    &s.inner
}

let s = MyStruct { inner: 42 };
let inner = get(&s);
```

REFERENCE COUNTING

Un autre moyen de gérer la mémoire et le partage des ressources.

```
fn main() {
  let data = Rc::new(42);  // count = 1
  {
      let data2 = data.clone();  // count = 2
      do_something(data2);  // do_something propriétaire de data2
  }  // count = 1
  do_something(data);
} // count = 0; libère la mémoire

fn do_something(data:: Rc<i32>) { /* ... */ }
```

GESTION DE LA CONCURRENCE

```
let mut data = vec![1u32, 2, 3];

for i in 0..3 {
    thread::spawn(move || {
        data[i] += 1;
    });
}
```

```
8:17 error: capture of moved value: `data`
        data[i] += 1;
        ^~~~
```

GESTION DE LA CONCURRENCE

Solution:

- Mutex pour modification sûre
- Arc pour le partage

```
let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));

for i in 0..3 {
    let data = data.clone();
    thread::spawn(move || {
        let mut data = data.lock().unwrap();
        data[i] += 1;
    });
}
```

GESTION DE LA CONCURRENCE

```
let (tx, rx) = mpsc::channel();

thread::spawn(move || {
    let result = 5;
    tx.send(result);
});

let result = rx.recv().unwrap();
println!("{}", result);
```

AU FINAL RUST

- libère la mémoire pour vous comme un GC sauf qu'il n'y en a pas.
- évite les accès aux ressources désallouées.
- évite la modification de ressources pendant leur utilisation.
- évite les bugs de corruption mémoire (jardinage en mémoire).
- libère tout type de ressource, pas juste de la mémoire.
- évite l'accès simultané par plusieurs threads à la même ressource.

OPTION => PAS DE NULL

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

let x: Option<i32> = Some(5);
```

Le pattern Option permet de s'affranchir de NULL

GESTION DES ERREURS AVEC RESULT

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

Pas d'exception en Rust, trop coûteux, difficile à documenter.

AU FINAL POURQUOI CHOISIR RUST?

- C'est toujours bon de découvrir un nouveau langage
- Le code devient plus explicite.
- Le code devient plus sûr, fini les segfault

POUR EN APPRENDRE PLUS

- Rust book https://doc.rust-lang.org/book
- Rust by example http://rustbyexample.com
- Forum: http://users.rust-lang.org
- Reddit: r/rust
- Irc: #rust, #rust-fr sur irc.mozilla.org
- This Week in Rust: http://this-week-in-rust.org
- Stack Overflow: #rust