Rust - Introduction

Qualité de Développement — R5.A.08

C. Raïevsky

2026-2027

IUT A Valence
Université Grenoble Alpes

Département Informatique

BUT Informatique 3ème année

Pourquoi Rust dans un cours de qualité de développement ?

"Empowering everyone to build reliable and efficient software"

Langage

- Critères de qualité inclus dans la définition du langage

Outils

- Les outils supportent la démarche qualité

Critères de qualité inclus dans le langage

Langage

- Gestion sûre de la mémoire
- Force le programmeur à être explicite
- Force le programmeur à se poser des questions
- "*No undefined behavior*"

Critères de qualité inclus dans les outils

Les outils aident pour :

- La documentation
- Les tests
- L'intégration de tests au sein de la documentation pour la maintenir à jour
- Le respect de normes de codage

Le compilateur

- Est explicite
- Donne des suggestions de correction

Pourquoi Rust ?

Langage compilé

Plus efficace énergétiquement à l'exécution

Langage le plus "admiré" depuis 2016

2025 : "Rust is yet again the most admired programming language (72%)

Stackoverflow developpers survey 2025

Linux et Windows le poussent pour le développement de pilotes

Principales caractéristiques de Rust

- Compilé
- Fortement et statiquement typé
- Multi-paradigmes :
  - Orienté objet *mais sans héritage*
  - Fonctionnel
  - Concurrent

Pricipales différences avec d'autres langages "systèmes"

- Gestion de la mémoire
  - Borrow checking
  - Ownership
  - Variables constantes et *Move semantic* par défaut
- Pas d'héritage
- Pas d'exceptions
- Gestion des erreurs
  - Type *Result*
  - Type *Option*
  - *Syntactic Sugar* : '?'

Langage compilé

Processus de compilation :

Rust → llvm IR → assembleur ou WebAssembly

Outil incontournable : Cargo

- Gestion du processus de compilation
- Gestion des dépendances
- Creation de projet
- Passage des tests
- Exécution

Fortement et statiquement typé

Les types doivent être connus à la compilation

Pour alléger :

- Déduction de type : le compilateur déduis les types 99% du temps
- Generics : pas besoin de tout ré-écrire pour un autre type

Déduction de type

```
1 let x = 42;
2 let y;
3 if x > 0 {
4     y = "chose";
5 } else {
6     y = "machin";
7 }
```

Generics

- Definition de types génériques
- Définition de fonctions génériques
- Implémentations réalisées par le compilateur

Multi-paradigmes

Orienté objet

- Types de données utilisateurs : `struct`, `enums`
- Méthodes : fonctions associées à une instance
- Fonctions associées au type, similaire aux méthodes de classe ou statiques
- Encapsulation :
  - Au niveau `struct` : contrôle d'accès aux champs et méthodes
  - Au niveau module, crate, bibliothèque

Pas d'héritage !!

Des "`traits`"

Orienté Objet

Types de données utilisateur : `struct`

```
1 struct Point {
2     x: f32,
3     y: f32,
4 }
5
6 let mut p = Point {x:41.0, y:42.0};
7 p.x = p.x * 0.5;
```

Orienté Objet - Méthodes

Méthodes d'un `struct`, associées à une instance

```
1  impl Point {
2      fn get_x(&self) -> f32 {
3          self.x
4      }
5
6      fn displace(&mut self, dx: f32, dy: f32) {
7          self.x += dx;
8          self.y += dy;
9      }
10 }
```

Orienté Objet - Méthodes

Méthodes d'un `struct`, associées à une instance

```
impl Point {
    fn get_x(&self) -> f32 {
        self.x
    }

    fn displace(&mut self, dx: f32, dy: f32) {
        self.x += dx;
        self.y += dy;
    }
}
```

```
let mut p = Point {x:41.0, y:42.0};
p.displace(-0.3, 0.7);
println!("A Point's x: {}", p.get_x());
```

Orienté Objet - Méthodes

Méthodes d'un `struct`, associées à une instance

```
1  impl Point {
2      fn get_x(&self) -> f32 {
3          self.x
4      }
5
6      fn displace(&mut self, dx: f32, dy: f32) {
7          self.x += dx;
8          self.y += dy;
9      }
10 }
```

- `self` : référence à l'instance
- `self` est `mut` si la méthode modifie l'instance

Orienté Objet - Fonctions associées

Fonction associée à un `struct`, associée au type

```
1  impl Point {
2      fn new(x: f32, y: f32) -> Self {
3          Point {x, y}
4      }
5
6      fn origin() -> Self {
7          Point {x: 0.0, y: 0.0}
8      }
9  }
10
11 let mut p1 = Point::new(41.0, 42.0) ;
12 let mut o = Point::origin() ;
```

La fonction `new` est une fonction associée comme une autre

Orienté Objet - Encapsulation

La visibilité est gérée au niveau :

- Des modules
- Des structs et enums

Les éléments d'un module ou d'un struct (fonctions, champs, structs, traits, etc.)

- Par défaut : privé
- Public :
  - au sein du module et de ses descendants
  - si explicitement déclaré `pub`

Orienté Objet - Encapsulation

Les éléments d'un **module** sont privés par défaut

```
1 mod points {
2     struct PointPrivate {
3         x: f32,
4         y: f32,
5     }
6 }
7 // error: struct `PointPrivate` is private
8 let p_priv = points::PointPrivate {x:41.0, y:42.0};
```

Orienté Objet - Encapsulation

Les éléments d'un **`struct`** sont privés par défaut

```
1 mod points {
2     pub struct PointPublic {
3         x: f32,
4         y: f32,
5     }
6 }
7 // error: x and y are private
8 let p_pub = points::PointPublic {x:41.0, y:42.0};
```

Orienté Objet - Encapsulation

```
1 mod points {
2     pub struct Point {
3         pub x: f32,
4         pub y: f32,
5     }
6 }
7 // OK
8 let p = points::Point {x:41.0, y:42.0};
```

Bonne idée ?

Enum - comme un `enum` Java ou C, mais en mieux

Les enums Rust sont bien plus riches que dans d'autres langages

- Les variantes peuvent avoir des types complexes
- Possibilité de définir des fonctions et des méthodes associées

Enum simple

```
1 enum HealthStatus {
2     Healthy,
3     Ill,
4     Dead,
5     Unknown,
6 }
```

Identiques aux enum Java ou C++

Les `enums` en mieux sont des types "algébriques", "somme" ou "*tagged union*"

Variants de types complexes

- Chaque variants peut être d'un type différent
- Pas de restriction sur les types possibles (types de base, structs, etc...)
- Le type ainsi défini est une union de types

```
1 enum Shape {
2     Circle(f64),
3     Square(f64),
4     Rectangle(f64, f64),
5 }
```

Tagged union on Wikipedia

Enums - fonctions et méthodes associées, comme pour un `struct`

```rust
impl Shape {
    fn area(&self) -> f64 {
        match *self {
            Shape::Circle(radius) => std::f64::consts::PI * radius.powi(2),
            Shape::Square(side) => side.powi(2),
            Shape::Rectangle(length, width) => length * width,
        }
    }
}
```

Les plus attentifs auront remarqué que cela permet de réaliser une forme de polymorphisme "statique", non extensible.

Enums - exemples

Utilisation simple :

```
let c = Shape::Circle(3.0);
println!("Area of circle: {}", c.area());
```

"Polymorphisme" :

```
let mut v = Vec::new();
v.push(Shape::Circle(12.0));
v.push(Shape::Square(12.0));
v.push(Shape::Rectangle(12.0, 12.0));

for shape in v.iter() {
    println!("Shape's area: {}", shape.area());
}
```

Fonctionnel

Fonctionnel

- Closure
- Pattern matching
- Itérateurs

Fonctionnel

Fonctionnel

- Closure
- Pattern matching
- Itérateurs

```
let f = |x| {x+1};
println!("Playing with functions: {}", f(41));
```

Fonctionnel

Fonctionnel

- Closure
- Pattern matching
- Itérateurs

```
1 let s = "Hello";
2 match s {
3     "Init" => println!("Matched Init"),
4     "Truc" | "Chose" => println!("Matched Truc or Chose"),
5     "Hello" => println!("Matched Hello"),
6     _ => println!("Matched something else"),
7 }
```

Tour of Rust on this topic

Fonctionnel

Fonctionnel

- Closure
- Pattern matching
- Itérateurs

```
1 let mut v = vec![41;7];
2
3 v.iter_mut().for_each(|x| *x = *x+1);
4
5 println!("Modified vector: {:?}", v);
6
7 println!("Vector's sum: {}", v.iter().fold(0, |a, x| a + x))
```

Concurrent

- Threads et synchronisation dans la bibliothèque standard
- Channels
- Ownership system
- `async await`