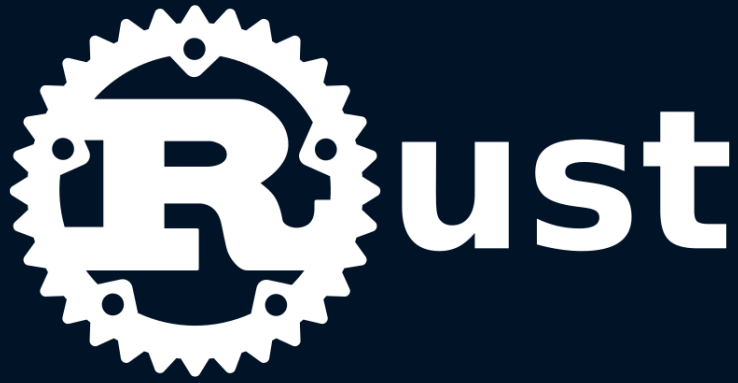
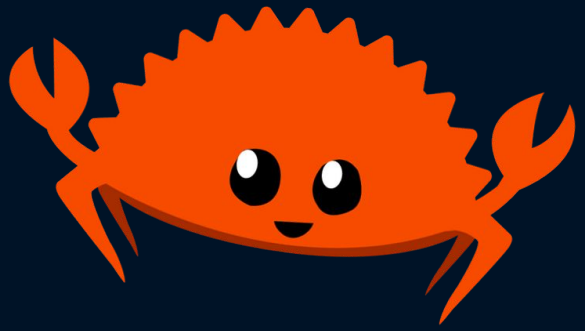


Programmation Rust



RUST : Programmation Système & Réseau

Déroulement du cours :

- Pourquoi Rust ? : Définitions et rappels génériques. Histoire, philosophie et particularités du langage.
- Écosystème Rust : Installation et configuration de l’environnement de dev.
- Exercices pratiques
- Concepts de bases : Variables, types de données, conditions, boucles, gestions des erreurs, …
- Concepts avancés : Ownership, borrowing, lifetimes, structures de données et collections, concurrency, unsafe, …
- Projets

RUST : Programmation Système & Réseau

Objectifs du cours :

- Comprendre les caractéristiques clés de Rust.
- Acquérir une connaissance approfondie des concepts fondamentaux.
- Écrire du code sécurisé, performant, qui suit les conventions et les pratiques recommandées.
- Savoir Développer, Tester, Débugger des applications Rust.
- Progression en autonomie post-formation.
- Préparation pour des projets réels.

RUST

Rust est un langage de programmation système, open source, strict, compilé, multi paradigme conçu initialement par Mozilla Research depuis 2010. Rust vise à construire un langage fiable, évitant les erreurs courantes de sécurité et de gestion de la mémoire possible dans d'autres langage de programmation, tout en offrant des performances comparables, et en facilitant le développement de logiciels concurrents et parallèles de manière plus sûre et simplifiée.

RUST : Objectifs

Sécurité Mémoire: Éliminer les erreurs de segmentation et garantir la sécurité des accès mémoire sans recourir à un garbage collector.

Concurrence : Faciliter la programmation concurrente et parallèle en évitant les data races grâce à son système d'ownership et de types.

Performance: Offrir une performance comparable à celle du C et du C++, permettant un contrôle précis sur l'utilisation des ressources systèmes.

Productivité: Fournir des outils modernes tels que le gestionnaire de paquets Cargo, un typage strict, et des messages d'erreur compréhensibles pour améliorer la productivité des développeurs.

RUST

Garbage Collector : Un mécanisme de gestion automatique, d’allocation et libération de la mémoire qui libère, évitant ainsi les fuites de mémoire.

RUST

Data Race : Dans les programmes multithreads lorsque plusieurs threads accèdent simultanément à une même variable en mémoire notamment en écriture. Si ces accès ne sont pas correctement synchronisés il peut y avoir des résultats imprévisible impliquant des bugs difficiles à détecter et répliquer.

RUST : Un langage de programmation impératif

Un style de programmation où le programmeur donne des instructions séquentielles pour effectuer une tâche (ex. C, C++ Python, Bash, …).

À l'inverse le style déclaratif c'est un style où le programmeur décrit le résultat souhaité sans expliciter les étapes nécessaires pour y parvenir (ex. SQL, HTML).

RUST : Un langage de programmation compilé

Un langage dont le code source est transformé en langage machine et packagé en binaire exécutable par un compilateur. (ex. C, CPP, Rust, …).

Rust n’est pas un langage interprété, dont le code source est exécuté directement par un interpréteur sans étape de compilation (ex. Python, Bash, Perl, …)

RUST : Un langage haut et bas niveau

Rust fusionne les avantages des langages de bas et de haut niveau en permettant un contrôle système précis tout en offrant des abstractions modernes pour une programmation sécurisée et facile.

Il combine la gestion fine de la mémoire et les performances directes d'un langage de bas niveau avec la sécurité mémoire et les fonctionnalités expressives typique d'un langage de haut niveau.

Cette approche permet de à Rust d'être particulièrement adapté pour développer des programmes complexes avec une efficacité et une sureté accrue.

RUST : Est adapté pour ?

- Le développement de logiciels nécessitant sécurité, stabilité et performance élevées
- Les systèmes et systèmes embarqués
- Applications bas niveau
- Web (avec WASM) pour des applications web très performantes
- Les moteurs de jeu
- …

RUST : Inconvénients

- Learning Curve
- Complexité de la syntaxe
- Nouveaux concepts
- Rigeur
- Temps de compilation

RUST : Programmation Système & Réseau

Conclusion :

Rust est un langage orienté sécurité et performance : (Memory safe, Thread safe and Zero Cost Abstraction)

RUST : Écosystème

- Cargo : gestionnaire de paquets et outil de compilation
- Rustup : gestionnaire de toolchain
- Rustc : compilateur
- Rustlings : Exercices graduels pour progresser en Rust

RUST : Écosystème

Installation de l'écosystème :

```
$ sudo apt install rust-all
    ou
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh

$ rustc --version
$ cargo --version
```

RUST : Écosystème

Installation de Rustlings :

```
$ curl -L
https://raw.githubusercontent.com/rust-lang/rustlings/main/install.sh |
bash
$ cd rustlings
$ rustlings
```

RUST : Variables & Mutabilité

Immuabilité par défaut :

Dans la plupart des langages une variable est modifiable par défaut ; en Rust, les variables sont immuables (non modifiables) par défaut. Cela signifie que, une fois qu'une valeur est attribuée à une variable, elle ne peut pas être modifiée. Pour créer une variable modifiable il faut préciser sa mutabilité lors de sa déclaration.

```
let x = 5;
// x = 6; // Ceci génère une erreur car x est immuable.
```

RUST : Variables & Mutabilités

Variable mutable :

Pour créer une variable modifiable, on utilise `mut`.

```
let mut x = 5;
x = 6; // Ceci ne génère pas d'erreur.
```

RUST : Variables & Mutabilité

Constantes :

Les valeurs constantes et connue à la compilation.

```
const x = 5;
// x = 6; // Ceci génère une erreur car x est une constante.
```

RUST : Variables & Mutabilité

Le masquage / shadowing :

Le masquage permet à une nouvelle variable de porter le même nom qu'une précédente variable dans la même portée.

```
 let x = 5;
 let x = 6; // Cela réserve un autre espace mémoire et oublies la première affectation
```

RUST : Variables & Mutabilité

Les types de données / Data type :

Rust est un langage strict ce qui veut dire que chaque valeur est d'un type bien déterminé qui indique quel genre de données sont manipulées. Les variables doivent être connues au moment de la compilation.

```
 let x = 5;
 let x = 6; // Cela réserve un autre espace mémoire et oublies la
première affectation
```

RUST : Variables & Mutabilité

Les types d'entiers en Rust :

```
let x: i8 = -128; // Signé de -128 à 127
let y: u8 = 255; // Non signé de 0 à 255

let a: i16 = -32768; // Signé de -32768 à 32767
let b: u16 = 65535; // Non signé de 0 à 65535

let c: i32 = -2147483648; // Signé de -2147483648 à 2147483647
let d: u32 = 4294967295; // Non signé, de 0 à 4294967295

let e: i64 = -9223372036854775808; // Signé de -2⁶³ à 2⁶³ - 1
let f: u64 = 18446744073709551615; // Non signé, de 0 à 2⁶⁴ - 1
```

RUST : Variables & Mutabilité

Les types littéraux d'entiers en Rust :

```
let decimal: i32 = 98_222;
let hexadecimal: i32 = 0xff;
let octal: i32 = 0o77;
let binary: i32 = 0b1111_0000;
let byte: u8 = b'A';
```

RUST : Variables & Mutabilité

Nombres réels :

```
let var_f32: f32 = 3.14; //min: -3.40282347 × 10³⁸ max: 3.40282347 × 10³⁸
let var_f64: f64 = 2.71828; //min: -1.7976931348623157×10³⁰⁸ max: 1.7976931348623157
×10³⁰⁸
```

RUST : Variables & Mutabilité

Opérations arithmétiques :

```
let a: i32 = 10;
let b: i32 = 3;
let addition = a + b;
let soustraction = a - b;
let multiplication = a * b;
let division = a / b;
let modulo = a % b;
a += 1; // incrémente a de 1
b -=1; // décrémente b de 1
```

RUST : Variables & Mutabilité

Chaine de caractère :

```
let my_string: &str = “Hello world!”;
// ou
let mut my_string: Option<&str> = None;
my_string = Some("Hello world!");
// ou
let mut my_string: String = String::new();
my_string = "Hello world!".to_string();
```

RUST : Variables & Mutabilité

Les tuples :

Les tuples en Rust sont des collections hétérogènes de valeurs, ils peuvent contenir plusieurs valeurs de types différents dans une seule structure de données. Les tuples sont très utiles pour retourner plusieurs valeurs d'une fonction et pour gérer des données qui sont naturellement regroupées ensemble.

```
let mon_tuple: (i32, f64, &str) = (500, 6.4, "Bonjour");
```

RUST : Les fonctions

Définition d’une fonction :

```
fn saluer() {
    println!("Bonjour !");
}
```

RUST : Les fonctions

Appel d'une fonction :

```rust
fn main() {
    saluer();
}

fn saluer() {
    println!("Bonjour !");
}
```

RUST : Les fonctions

Les paramètres :

```rust
fn affiche_nombre(x: i32) {
    println!("Le nombre est : {}", x);
}
```

RUST : Les fonctions

Les retours :

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

RUST : Les fonctions

Les retours implicite :

Pour faciliter l'écriture de fonctions courtes et concises.

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

RUST : Les conditions

`if` :

```rust
let nombre = 6;

 if nombre % 2 == 0 {
     println!("{} est pair", nombre);
 } else {
     println!("{} est impair", nombre);
 }
```

RUST : Les conditions

`else if` :

```rust
let nombre = 15;

 if nombre % 4 == 0 {
     println!("{} est divisible par 4", nombre);
 } else if nombre % 3 == 0 {
     println!("{} est divisible par 3", nombre);
 } else if nombre % 2 == 0 {
     println!("{} est divisible par 2", nombre);
 } else {
     println!("{} n'est divisible ni par 4, ni par 3, ni par 2", nombre);
 }
```

RUST : Les conditions

`if` as an instruction :

```
let condition = true;
let nombre = if condition { 5 } else { 6 };
```

RUST : Les conditions

`match` expression :

```rust
let nombre = 2;

 match nombre {
      1 => println!("Un"),
      2 => println!("Deux"),
      3 => println!("Trois"),
      _ => println!("Quelque chose d'autre"),
 }
```

RUST : Les tests

Les tests en Rust sont un moyen intégré et puissant pour vérifier que votre code se comporte comme prévu. Rust place une grande importance sur la sécurité et la fiabilité, et les tests unitaires et d'intégration font partie intégrante de cette philosophie.

RUST : Les tests unitaires

Les tests unitaires sont utilisés pour tester des parties isolées de votre code, typiquement des fonctions ou des modules individuels. En Rust, les tests unitaires sont généralement placés dans le même fichier que le code qu'ils testent, sous un module nommé tests annoté avec cfg(test).

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

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn cela_fonctionne() {
        assert_eq!(additionner(2, 2), 4);
    }

    #[test]
    #[should_panic]
    fn cela_doit_panic() {
        //vérifier qu'une panique est bien déclenchée
        panic!("Ce test va paniquer.");
    }
}
```

RUST : Les tests d’intégration

Les tests d'intégration vérifient que plusieurs parties de votre bibliothèque fonctionnent correctement ensemble. Les tests d'intégration sont placés dans un dossier nommé tests à la racine de votre projet Cargo.

Chaque fichier dans ce dossier est compilé comme un crate de test séparé.

```rust
use nom_du_crate; // Remplacez `nom_du_crate` par le nom de votre crate

#[test]
fn test_integration() {
    assert_eq!(nom_du_crate::additionner(2, 3), 5);
}
```

RUST : Variables & Mutabilité

Booléen :

```
let x: bool;
let y: bool = false;
```

RUST : Variables & Mutabilité

Caratère:

```
let c: char;
c = ‘a’;
```

RUST : les tableaux / arrays

Un tableau en Rust est une collection de données de taille fixe où tous les éléments sont du même type.

- La taille d'un tableau est déterminée lors de sa déclaration et ne peut pas changer.
- Les éléments d'un tableau sont stockés en mémoire de manière contiguë, ce qui rend l'accès aux éléments très rapide.

```
let array0: [Type; Taille] = [valeur1, valeur2, ..., valeurN];
let array1: [i32; 100] = [0; 100];

let jours: [&str; 7] = ["Lundi", "Mardi", "Mercredi", "Jeudi",
"Vendredi", "Samedi", "Dimanche"];
let premier_jour = jours[0]; // lundi
```

RUST : les vecteurs

Les vecteurs en Rust sont des structures de données flexibles qui permettent de stocker une collection d'éléments du même type. Ils sont dynamiques, ce qui signifie que leur taille peut changer au fur et à mesure que des éléments sont ajoutés ou retirés.

```rust
// Création d'un vecteur vide
let mut vecteur: Vec<i32> = Vec::new();
// Création d'un vecteur avec des valeurs initiales
let vecteur = vec![1, 2, 3, 4, 5];

let mut vecteur = Vec::new();
vecteur.push(1);
vecteur.push(2);
…

let dernier = vecteur.pop(); // Retire et retourne Some(4)
let deuxieme = vecteur.remove(1); // Retire et retourne 2, le vecteur est maintenant [1, 3]
```

RUST : Propriété / Ownership

En Rust, la propriété (ownership une des plus unique particularité du langage) est un mécanisme qui assure que chaque valeur / espace mémoire n’est géré que par une seule “variable” à la fois, ce qui élimine les problèmes courant de gestion de la mémoire, tels que les fuites de mémoire et les doubles libérations (sans garbage collector) et encourage un dev plus rigoureux.

```
let s1 = String::from("hello");
let s2 = s1; // La propriété de la chaîne est transférée à s2.
// s1 n'est plus valide ici.
```

RUST : Clonage

Si on veut garder une valeur mais aussi la transférer, on peut la cloner. Cela crée une nouvelle instance avec la même valeur.

```
let s3 = s2.clone(); // s3 est une nouvelle chaîne avec le même contenu que s2.
```

RUST : Portée et Libération

La portée est la partie d'un programme où une variable est accessible. La portée est souvent délimitée par des blocs de code c'est à dire entre accolades `{...}`.
En rust une variable est valide depuis le point où elle est déclarée jusqu'à la fin du bloc de coude où elle est déclarée.

```
{
    let x = 5; // x est valide à partir d'ici
    // Utilisation de x possible ici
} // x n'est plus valide après cette accolade
```

RUST : les structures

Les structures en Rust sont utilisées pour créer des types de données personnalisées et aider à une organisation claire et une modélisation souple des modèles de données de vos programme.

```
struct Malware {
    name: String,
    type: String,
    severity_level: u8,
    target_platforms: Vec<String>,
    …
}
```

RUST : les structures

```
    let wannacry = Malware {
         name: "WannaCry".to_string(),
         malware_type: "Ransomware".to_string(),
         severity_level: 5,
         target_platforms: vec!["Windows XP".to_string(), "Windows 7".to_string(),
"Windows 8".to_string(), "Windows Server 2003".to_string(), "Windows Server 2008
R2".to_string(), "Windows Server 2012".to_string()],
     };
```

RUST : Portée d’une fonction

Les paramètres d'une fonction ont leur propre portée, qui est la portée de la fonction.
Lorsqu'un paramètre est passé à une fonction, s'il est propriétaire d'une valeur, cette propriété est soit transférée à la fonction (si la valeur est passée directement), soit empruntée temporairement (si passée par référence).

Lorsqu'une variable est empruntée (par référence), la portée de la référence est également limitée au bloc dans lequel elle est déclarée.
Rust s'assure qu'une référence ne survit jamais à son propriétaire, prévenant ainsi les références pendantes.

RUST : Programmation Système & Réseau

```rust
fn main() {
    let s = String::from("hello");
    prend_et_retourne(s); // La propriété de s est transférée
    // s n'est plus valide ici
}

fn prend_et_retourne(a: String) -> String {
    // a est valide ici
    a // a est retournée et sa propriété est transférée
}
```