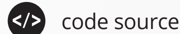
INFORMATIQUE
TECHNIQUE

Expert

Rust

Développez des programmes robustes et sécurisés

En téléchargement



Version en ligne
OFFERTE !
pendant 1 an

Benoît PRIEUR

eni

Les éléments à télécharger sont disponibles à l'adresse suivante :
http://www.editions-eni.fr
Saisissez la référence ENI de l'ouvrage **EIRUST** dans la zone de recherche et validez. Cliquez sur le titre du livre puis sur le bouton de téléchargement.

Avant-propos

Chapitre 1
Comment et pourquoi Rust ?

1. Introduction 13
2. Contexte d'invention et nature du langage Rust 14
3. Pourquoi le langage Rust ? 15
   3.1 La question de la gestion de la mémoire 15
      3.1.1 La pile 15
      3.1.2 Le tas 16
      3.1.3 Le typage sûr 16
   3.2 La question de la gestion concurrentielle des threads 19
      3.2.1 Contexte d'utilisation du parallélisme 19
      3.2.2 Situation de compétition 20
      3.2.3 L'enfer de l'exclusion mutuelle 22
4. Conclusion 24

Chapitre 2
Commençons à utiliser concrètement Rust

1. Installation et exploration 25
   1.1 L'outil rustup 25
   1.2 L'outil central : cargo 28
      1.2.1 Versions installées 28
      1.2.2 Un premier projet 28

1.2.3 Analyse et explications relatives au premier projet 29
1.2.4 Commandes principales cargo 31
2. Premiers travaux en Rust 33
2.1 Un premier exemple de programme Rust 33
2.2 Exécution du programme 36
2.3 Compléments apportés au programme 37
3. Outils complémentaires autour du projet réalisé 39
3.1 Un mot sur les tests unitaires en Rust 39
3.2 Un mot sur l'organisation par module 42
3.3 Un mot sur la ligne de commande 45
4. Nommons les choses - Un peu de vocabulaire 47
4.1 Introduction 47
4.2 Petit lexique du langage Rust 47
4.2.1 Mutabilité 47
4.2.2 Structure (struct) 47
4.2.3 Trait et générique 47
4.2.4 Caisse (crate) et module 48
4.2.5 Propriété et emprunt 48
5. Conclusion 49

Chapitre 3
Types, structures de données en Rust

1. Introduction 51
2. Les types primitifs simples 52
2.1 Types numériques entiers 52
2.1.1 Entiers signés 52
2.1.2 Entiers non signés 52
2.1.3 Entiers codés d'après la taille du processeur 53
2.2 Types numériques flottants 53
2.3 Le type booléen 54
2.4 Le type char 54

3. Les types primitifs, structures de données 54
3.1 Le tuple 54
3.2 La structure struct 56
3.2.1 Structure, cas général 56
3.2.2 Structure et mutabilité. 58
3.2.3 Le tuple structure 59
4. Les types pointeurs en Rust 60
4.1 Références en Rust 60
4.2 Boîte (box) en Rust 64
4.3 Pointeurs bruts (raw pointers) en Rust. 66
5. Les types tableaux, vecteurs et tranches 69
5.1 Introduction 69
5.2 Les tableaux en Rust. 69
5.3 Les vecteurs en Rust 71
5.4 Les tranches en Rust. 75
6. Le type chaîne de caractères (string) 78
6.1 Introduction 78
6.2 Cas pratique autour de string 79
7. Conclusion 81

Chapitre 4
Possession et emprunt en Rust

1. Introduction 83
1.1 Le tas et la pile 84
1.2 Utilité de la possession. 84
2. Fonctionnement de la possession 85
2.1 Grands principes. 85
2.2 Exemples relatifs à la propriété 85
2.2.1 Propriété dans le tas 86
2.2.2 Propriété dans la pile 88

2.3 Plus loin avec la propriété : usage avec fonctions 91
2.3.1 Passage de valeur à une fonction 91
2.3.2 Retour de fonction 92
2.4 Conclusion à propos de la possession 93
3. Fonctionnement de l'emprunt 93
3.1 Introduction 93
3.2 Exemple : transfert de propriété vs emprunt 94
3.2.1 Transfert de propriété 94
3.2.2 Emprunt 95
3.3 Emprunt à base de références mutables 98
4. Conclusion 99

Chapitre 5
Structures en Rust

1. Premières structures en Rust 101
1.1 Introduction 101
1.2 Structure à champs nommés 102
1.2.1 Syntaxe 102
1.2.2 La question de la visibilité 103
1.3 Structure - tuple 107
1.4 Structure - unité 109
1.5 Un mot de l'implantation mémoire d'une structure 109
2. Les méthodes de structure en Rust 110
2.1 Prototype des méthodes 110
2.2 Usage du mot-clé self 111
2.3 Un exemple d'utilisation des méthodes 111
2.4 Vers les méthodes statiques en Rust 113
3. Structure générique 115
3.1 Introduction 115
3.2 Exemple support 115
3.3 Exemple support augmenté 117

4. La question des références dans une structure 120
4.1 Introduction 120
4.2 Mise en évidence du besoin d'annotation de durée de vie 121
4.3 Durée de vie basée sur le mot-clé static. 122
5. Notion de traits appliqués aux structures. 124
5.1 Introduction 124
5.2 Les traits prédéfinis appliqués aux structures. 124
6. Conclusion 126

Chapitre 6
Énumérations et motifs en Rust

1. Introduction 127
2. Les énumérations en Rust 128
2.1 Premiers exemples 128
2.2 Exemple de conversion d'entiers vers une énumération 131
2.2.1 Les options dans la librairie standard 131
2.2.2 Premiers filtrages par motif 132
2.3 Un mot sur les méthodes d'énumération 135
2.4 Utiliser des structures dans des énumérations 137
2.4.1 Introduction 137
2.4.2 Exemple d'utilisation 137
2.5 Les énumérations génériques 139
2.5.1 Exemple dans la librairie standard. 139
2.5.2 Arbre binaire avec une énumération générique 140
3. Filtrage par motif 142
3.1 Premier exemple, pour rappel 142
3.2 Plus loin avec les motifs 144

Chapitre 7
Les traits en Rust

1. Introduction 149
2. Premier trait en Rust 150
  2.1 Création d'une caisse et d'un exécutable client 150
  2.2 Définition d'un trait 153
3. Utiliser un trait en paramètre 160
  3.1 Introduction 160
  3.2 Exemple de trait en paramètre. 160
  3.3 Plusieurs paramètres d'un même trait en paramètre 162
4. Notion de trait lié 163
  4.1 Introduction 163
  4.2 Plusieurs traits liés différents pour un même paramètre 163
    4.2.1 Introduction 163
    4.2.2 Le trait prédéfini Display 163
    4.2.3 Création de la structure Tortue. 164
    4.2.4 Un mot sur la clause where 166
5. Un trait comme valeur de retour. 167
  5.1 Introduction 167
  5.2 Exemple support. 167
6. Points d'architecture impliquant les traits 169
  6.1 Traits, génériques et structures 169
  6.2 Un mot sur les sous-traits 171

Chapitre 8
Les traits prédéfinis en Rust

1. Introduction 173
2. Des traits prédéfinis essentiels : les itérateurs 173
3. Notion de surcharge d'opérateurs 176
   3.1 Introduction 176
   3.2 Les opérateurs surchargeables 177
   3.3 Exemple support 178
      3.3.1 Contexte des nombres complexes 178
      3.3.2 Premiers éléments de programmation 178
      3.3.3 Surcharges des opérateurs + et - 179
      3.3.4 Surcharges de la comparaison 181
4. Inventaire et usage de quelques traits prédéfinis 182
   4.1 L'usage de Derive 182
      4.1.1 Explications 182
      4.1.2 Exemple support 183
   4.2 Inventaire des traits 185
      4.2.1 Introduction 185
      4.2.2 Le trait Drop 186
      4.2.3 Les traits Deref et DerefMut 187
      4.2.4 Le trait Default 188
      4.2.5 Le trait From 189

Chapitre 9
Les chaînes de caractères en Rust

1. Introduction 193
2. Encodages Unicode et UTF-8 et caractères en Rust 194
   2.1 Quelques définitions 194
   2.2 Encodage en Rust 194
      2.2.1 Le type char et l'Unicode 194
      2.2.2 Les types String et str et l'UTF-8 194

3. À la découverte des caractères (char) en Rust. 195
3.1 Nature des caractères 195
3.2 Casse des caractères 197
3.3 Conversion vers un entier 197
4. À la découverte de String et str 199
4.1 Introduction 199
4.2 Le type str 199
4.3 Le type String 201
4.3.1 Construction du type 201
4.3.2 Création d'un String 201
4.3.3 Premiers outils autour de String 204
4.3.4 Insertion dans un String 205
4.3.5 Suppression avec un String 207
4.3.6 Recherche et remplacement dans un String 209
5. Un mot sur les expressions régulières 211

Chapitre 10
Les vecteurs en langage Rust

1. Introduction 215
2. Le vecteur Vec<T> 216
2.1 Introduction 216
2.2 Accès aux éléments (référence et copie) 220
2.3 Méthodes avancées d'accès 224
2.4 Considérations sur la taille et la capacité 228
2.5 Ajouts et retraits de valeurs 230
2.6 Autres opérations sur vecteurs 234
2.7 Un mot sur le tri de vecteurs 235
2.8 Un mot sur la recherche dans un vecteur 237

Chapitre 11
Autres collections en langage Rust

1. Introduction 239
2. La collection VecDeque<T> 239
   2.1 Présentation 239
   2.2 Utilisation de VecDeque<T> 241
3. La collection LinkedList<T> 243
   3.1 Présentation 243
   3.2 Utilisation simple de LinkedList<T> 245
4. La collection BinaryHeap<T> 247
   4.1 Présentation 247
   4.2 Utilisation de BinaryHeap<T> 249
5. Table de hachage HashMap<Key, Value> 252
   5.1 Présentation 252
   5.2 Utilisation de HashMap<Key, Value> 253
   5.3 Un mot sur la collection BTreeMap<Key, Value> 254
6. Approche ensembliste avec HashSet<T> 256
   6.1 Présentation 256
   6.2 Utilisation de HashSet<T> 257
   6.3 Un mot sur la collection BTreeSet<T> 260
7. Conclusion 262

Chapitre 12
Les closures en Rust

1. Introduction 263
2. Considérations théoriques 264
3. Première utilisation d'une closure 265
4. Tri facile avec une closure 266
5. Les closures, résumé des premières notions 269

6. Considérations sur les traits FnOnce, FnMut et Fn 270
6.1 Explications du fonctionnement général 270
6.2 Le mot-clé move 271

Chapitre 13
Les threads et le multithreading en Rust

1. Introduction 273
2. Premiers threads et parallélisme 274
2.1 Introduction 274
2.2 Usage de spawn 275
2.2.1 Introduction 275
2.2.2 Premier exemple concret 275
2.3 Attendre la fin des threads secondaires (Fork-Join) 278
2.4 L'alternative rayon 281
2.4.1 Premier exemple 281
2.4.2 Un peu de parallélisme 284
3. Communication entre threads 287
3.1 Introduction 287
3.2 Usage des canaux en Rust 287
3.2.1 Définition 287
3.2.2 Première utilisation d'un canal 288
3.2.3 Considérations sur la sécurisation du canal 289
3.2.4 Envois multiples dans le canal 290
4. Usage des verrous mutuels exclusifs 294
4.1 Présentation et définition 294
4.2 Les structures Rc et Arc 295
4.2.1 Introduction 295
4.2.2 Rc<T> 296
4.2.3 Arc<T> 299
4.3 Usage de Mutex 301

Chapitre 14
Rust et WebAssembly

1. Introduction 305
1.1 Première considération 305
1.2 Explications de WebAssembly 305
2. Exemple détaillé de WebAssembly 306
2.1 Installation d'outils 306
2.2 Exemple détaillé 307
2.2.1 Introduction 307
2.2.2 Développement côté Rust 308
2.2.3 Développement côté JavaScript 310
3. Pour aller plus loin 318
3.1 Quelques caisses utiles 318
3.1.1 Caisse wasm-bindgen 318
3.1.2 Caisse console_error_panic_hook 319
3.1.3 Caisse console_log 319
3.2 Le gestionnaire de paquets npm 319
3.2.1 Introduction 319
3.2.2 Déploiement sur npm 320

Chapitre 15
Notions avancées en Rust

1. Introduction 323
2. Les objets-traits 323
2.1 Présentation 323
2.2 La surcharge statique 324
2.3 La surcharge dynamique 326
2.3.1 Introduction 326
2.3.2 Contexte de l'exemple 326
2.3.3 Définition d'un trait dédiée à la surcharge dynamique . 327
2.3.4 Usage du mot-clé dyn 328

3. Le code Rust non sûr 329
3.1 Introduction 329
3.2 Déréférencer un pointeur brut 330
3.3 Modifier une variable statique mutable 332
3.4 Implémenter des traits non sécurisés 334

Chapitre 16
Projet final : coder et publier une caisse

1. Introduction 335
2. Périmètre fonctionnel 337
3. Développement Rust de la librairie 340
4. Autres aspects et tests de la caisse 345
5. Publication de la caisse 350

Index 353

Chapitre 4
Possession et emprunt en Rust

1. Introduction

En Rust, on maîtrise totalement la gestion de la mémoire. Tant du point de vue de l'allocation que de celui de la désallocation. Cette opération s'effectue, bien sûr, sans *garbage collector* (ramasse-miettes) comme c'est le cas en C# ou en Java. On est significativement plus dans la situation du C++, dans laquelle la programmeuse ou le programmeur alloue et désalloue à sa guise.

Remarque

Un ramasse-miettes est un dispositif en code informatique en charge de la désallocation automatique de la mémoire, dès lors qu'elle n'est plus utilisée (ou supposée l'être). La personne en charge du développement n'a plus alors (en théorie) à se poser la question de la libération de la mémoire allouée.

Il existe cependant une différence notable entre C++ et Rust : en C++, une situation problématique n'apparaîtra qu'à l'exécution, que ce soit le fait d'accéder à un espace mémoire déjà désallouée, ou encore que l'on essaie de stocker de la donnée dans un espace non encore alloué. En Rust, ce type de problème est détecté dès la compilation et entraîne un échec de celle-ci. En cas d'erreur, il convient de corriger le code, afin que la compilation réussisse, donnant ainsi la garantie que l'exécution se passera bien, sans tentative d'accès à de l'espace corrompu.

Le langage Rust peut réaliser cela grâce à un principe qui le rend singulier : la **possession** (*ownership* en anglais). Ce principe est absolument central en Rust.

1.1 Le tas et la pile

Évidemment, la gestion de la mémoire implique d'avoir une vision assez claire du fonctionnement du tas et de la pile. Nous avons abordé la plupart des usages respectivement de la pile et du tas dans le chapitre précédent. Pour rappel, voici quelques points à garder à l'esprit :

- Quand on fait un appel de fonctions, les paramètres, c'est-à-dire les valeurs passées à la fonction, sont placés sur la pile. Ce sont des valeurs comme des entiers par exemple. Ce peut être des pointeurs vers quelque chose dans la mémoire. Dans le cas d'une fonction, dès lors que la fonction est terminée, les paramètres présents sur la pile sont retirés de la pile.
- Lorsqu'on alloue des valeurs dans le tas (grâce à *box*), une référence est maintenue sur la pile vers l'emplacement allouée.

1.2 Utilité de la possession

Avant d'expliquer le fonctionnement de la possession en Rust, voyons les quelques points relatifs à son intérêt quant à la gestion de la mémoire :

1. Premier intérêt : la minimisation des données en double (ou triple ou quadruple) dans le tas, et éventuellement dans la pile.

2. Second intérêt : libérer au plus tôt l'espace mémoire alloué dans le tas qui n'est plus utile.

3. L'intérêt principal sans doute : envisager la possession comme le garant d'une gestion optimisée de la mémoire qui, justement, permet de sécuriser l'ensemble du programme.

Sans plus attendre, voyons comment cela fonctionne.

© Editions ENI - All rights reserved

2. Fonctionnement de la possession

2.1 Grands principes

Les principes de la **possession** sont extrêmement simples, c'est leur mise en œuvre concrète qui peut parfois s'avérer compliquée. Voici les deux principes en question :

1. Toute valeur en Rust a un propriétaire et un seul.
2. Quand le propriétaire sort de la portée, la valeur est supprimée.

Cela appelle trois remarques :

– Une valeur peut changer de propriétaire. Auquel cas, le premier propriétaire n'a plus aucun pouvoir sur la valeur après cession.

– Cela s'applique aussi bien à des valeurs stockées sur le tas qu'à celles stockées dans la pile.

– L'unique propriété serait limitative : c'est pour cela que l'on peut emprunter une valeur (**emprunt**, *borrowing*). L'utiliser sans pouvoir la modifier, avant de la rendre.

2.2 Exemples relatifs à la propriété

Dans cet exemple, nous allons tâcher de détailler le fonctionnement de la possession, pour deux valeurs :

– Une valeur entière stockée sur la pile.

– Une valeur relative à une chaîne de caractères, stockée dans le tas.

Remarque

L'exemple proposera différentes séquences dans lesquelles la compilation échoue car le code enfreint justement les grands principes de la possession.

2.2.1 Propriété dans le tas

On définit une chaîne de caractères allouée dans le tas, appartenant à la variable `chaine`. En termes de **possession**, la variable `chaine` possède la valeur `"ENI"` :

```
let chaine = String::from("ENI");
```

Puis il y a un **déplacement** à `chaine2` :

```
let chaine2 = chaine;
```

Si on essaie d'afficher `chaine`, on aura une erreur de compilation :

```
println!("{}", chaine);
```

En effet, on essaie d'accéder à une valeur que le propriétaire a prêtée à une autre variable (`chaine2`). On obtient donc ce message d'erreur :

```
error[E0382]: borrow of moved value: `chaine`
--> src/main.rs:9:20
  |
6 |     let chaine = String::from("ENI");
  |         ------ move occurs because `chaine` has type `String`,
which does not implement the `Copy` trait
7 |     let chaine2 = chaine;
  |                   ------ value moved here
8 |     println!("{}", chaine2);
9 |     println!("{}", chaine);
  |                    ^^^^^^ value borrowed here after move
```

Le petit code associé à l'exemple est pour l'instant le suivant :

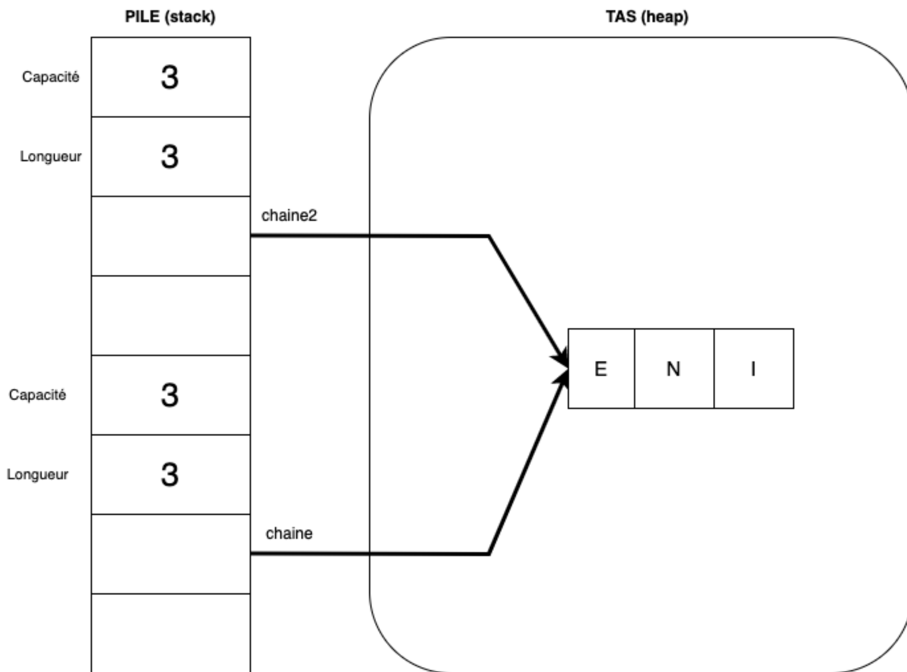
```
fn main() {
   possession_chaine()
}

fn possession_chaine() {
   let chaine = String::from("ENI");
   let chaine2 = chaine;

   println!("{}", chaine2);
   //println!("{}", chaine); // provoque une erreur car la valeur
                            // a été prêtée.
}
```

© Editions ENI - All rights reserved

Représentant l'état de la mémoire à ce stade de l'exemple, le schéma ci-dessous montre que la chaîne de caractères n'a évidemment pas été copiée dans le tas. C'est bien une seconde référence depuis la pile vers le tas qui est créée selon une sorte de triplet dans la pile (adresse, longueur de la chaîne, capacité allouée).



L'état de la mémoire à ce stade de l'exemple

Rien n'interdit de faire une copie de la chaîne de caractères dans le tas. Auquel cas, on utiliserait la méthode `clone` disponible sur la classe `String` :

```
let chaine_clone = chaine2.clone();
println!("{:p}", &chaine2);
println!("{:p}", &chaine_clone);
```

Quand on affiche les deux adresses respectives, elles sont différentes. En effet, on a bien une copie dans le tas de la chaîne de caractères `ENI`. Chacun de ses emplacements mémoire dans le tas sont pointés par une référence propriétaire dans la pile, respectivement `chaine2` et `chaine_clone`.

2.2.2 Propriété dans la pile

On définit une variable de type `i64`, c'est-à-dire un entier signé codé sur 64 bits. C'est typiquement un cas où la valeur est stockée sur la pile elle-même. Dans l'exemple, la variable « `annee_hector` » est la propriétaire de la valeur :

```
let annee_hector : i64 = 2011;
```

Puis on effectue une copie. Cependant, ce n'est pas une référence vers la première valeur (pour raisonner de façon analogue à ce que l'on a vu avec le tas). Ici, il y a une copie sur la pile. « `annee_hector_2` » est la propriétaire de la valeur copiée :

```
let annee_hector_2 : i64 = annee_hector;
```

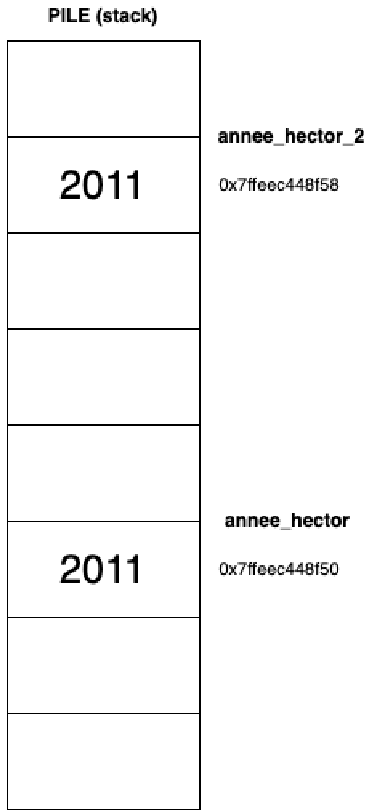
On affiche pour chacune des variables sa valeur ainsi que son adresse (au sein de la pile :

```
println!("{}", annee_hector);
println!("{:p}", &annee_hector);
println!("{}", annee_hector_2);
println!("{:p}", &annee_hector_2);
```

En sortie, on obtient ceci (on remarque que les deux adresses sont différentes) :

```
> 2011
> 0x7ffeec448f50
> 2011
> 0x7ffeec448f58
```

© Editions ENI - All rights reserved



Les deux valeurs entières dans la pile

À la fin de la portée courante (*scope*), les valeurs seront retirées de la pile car leurs propriétaires n'ont plus d'existence. Nous venons donc de voir au travers de deux exemples très simples les mécanismes de propriété en Rust.