



DEVIENS UN VRAI HÉROS DU CODE

LINKEDIN

TWITTER

GITHUB

CONTACT

RUST

Premiers pas avec le langage Rust



par ARKERONE / 18 JANVIER 2021



Le langage Rust est de plus en plus populaire. Celui-ci, a été élu pour la cinquième année consécutive, le langage de programmation le plus apprécié par les développeurs selon le sondage annuel de Stack Overflow. De nombreuses sociétés telles que Microsoft ou Amazon s'y intéressent également pour le développement de leurs solutions. Qu'est-ce qui rend Rust si populaire ? Pourquoi s'y intéresser ? C'est ce que nous allons découvrir dans cet article.

Il était une fois...

Rust est né en 2006, d'un projet personnel de Graydon Hoare alors employé chez Mozilla. Voyant le potentiel du langage, Mozilla décide de soutenir le développement de Rust en 2009 et présente celui-ci pour la première fois en 2010 lors du Mozilla summit. La première version stable de Rust voit le jour en 2015.

Autant dire que comparé aux autres langages comme JavaScript né en 1995, Python en 1991 ou même encore le langage C qui lui est né en 1973, Rust est très jeune.

Malgré sa jeunesse, Rust possède de nombreux atouts qui font que celui-ci est, aujourd'hui, de plus en plus utilisé, et ce dans de nombreux domaines tels que :

- La programmation système ;
- Les systèmes embarqués ;
- Le cloud ;
- Le développement web ;
- Les outils de lignes de commandes ;
- etc.

LES DERNIERS ARTICLES



Docker : conteneuriser son application

15 MARS 2022



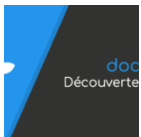
Git : pourquoi écrire des commits atomiques ?

25 OCTOBRE 2021



Git : L'utilisation des hooks avec Husky

11 OCTOBRE 2021



Docker : découverte des bases

14 JUIN 2021



Pourquoi lire le code des autres, fera de toi un meilleur développeur ?

14 AVRIL 2021

Il y a un véritable engouement autour de Rust, qui fait que de plus en plus d'entreprises s'y intéressent et n'hésitent pas à migrer leurs solutions vers celui-ci. Parmi celles-ci, nous pouvons citer [Dropbox](#), [Microsoft](#), [Npm](#), [Facebook](#) ou encore [Amazon](#).

Certains voient même en Rust le successeur du C et du C++.

Qu'est-ce que Rust ?

Rust est un langage de programmation compilé, multiplateforme, multiparadigme, bas niveau, au typage fort et statique qui se concentre sur la sécurité et les performances.

Oui je sais, ça en fait des concepts. Allez n'ayez pas peur je vais vous expliquer tout ça.

Un langage compilé

Contrairement à JavaScript ou Python, Rust est un langage compilé. Cela signifie que le code source est transformé, par un programme appelé compilateur, en code machine directement exécutable par votre ordinateur. L'avantage est qu'il n'est plus nécessaire de faire appel à un programme annexe pour votre programme, comme ça peut être le cas avec un langage interprété. L'autre avantage qui découle du premier est qu'un programme compilé est généralement plus rapide à l'exécution, car il n'y a pas d'étape d'interprétation.

Si vous souhaitez en savoir plus sur la compilation et l'interprétation d'un programme, je vous recommande [cet article](#).

Multiplateforme

Rust permet de développer des programmes pouvant être exécutés sur plusieurs plateformes que ce soit Windows, Linux ou MacOS. Rust est même utilisé pour le développement de [systèmes embarqués](#) et permet donc de développer des programmes destinés à être exécutés sur des microcontrôleurs.

Du fait que Rust est un langage compilé, il est toutefois nécessaire de compiler le code source pour ces différentes plateformes.

Multiparadigme

Multiparadigme signifie qu'il est possible avec Rust de développer des programmes en utilisant différents [concepts de programmation](#) tels que la programmation fonctionnelle ou la programmation orientée objet.

Bas niveau

Rust est souvent considéré comme un langage bas niveau, car celui-ci permet de développer des programmes faisant partie du système d'exploitation, donc souvent proche du matériel comme le permet le langage C par exemple. Mais Rust, offre certaines abstractions provenant de langage de plus haut niveau notamment une gestion de la mémoire efficace ou une gestion des erreurs intelligentes.

Typage fort et statique

Un langage fortement typé (ou à typage fort) signifie que le type d'une variable est garanti et que celui-ci ne pourra pas changer en cours d'exécution comme c'est le cas en JavaScript par exemple. De ce fait, les conversions implicites de type ne sont pas permises. Statique signifie quant à lui que ces vérifications de types sont réalisées à la compilation et non à l'exécution contrairement à un langage au typage dynamique.

Un langage performant et sécurisé

Quand on parle de Rust, il y a deux adjectifs qui reviennent souvent sur la table : performant et sécurisé.

Performant

Rust est rapide, très rapide, ces performances sont souvent comparées au C/C++. Certains évoquent même la possibilité d'utiliser Rust à la place du C++ dans le développement de jeux vidéos, un domaine très exigeant niveau performance.

Sécurisé

La gestion de la mémoire a toujours été une problématique. En C/C++, c'est aux développeurs de gérer celle-ci, ce qui implique souvent des erreurs de programmation dues à de la mémoire non libérée ou encore des dépassements de mémoire tampon (buffer overflow) pouvant entraîner des bugs et même des failles de sécurité. D'autres langages comme Java ou Python par exemple, résolvent la plupart des problèmes de mémoire avec notamment l'utilisation d'un ramasse-miette (garbage collector), mais au détriment de la performance.

Rust lui propose une gestion de la mémoire sécurisée, sans utilisation de ramasse-miette et donc sans perte de performance. Le compilateur vous indiquera toutes les erreurs à la compilation et non à l'exécution. Je vous préviens, vous allez détester le compilateur de Rust, mais il va vous éviter bon nombre d'erreurs.

Ces deux atouts font de Rust un langage de choix dans la programmation système. Rust est même évoqué comme une alternative du C pour le développement du noyau Linux. Même Microsoft réfléchit sérieusement à l'utilisation de Rust dans le développement de composants Windows.

Nous reviendrons plus en détail sur la gestion de la mémoire dans le prochain article consacré à Rust.

Découvrons Rust

C'est bien beau tout ça mais il est temps de découvrir les bases de Rust.

Installation

La première étape consiste à installer Rust. Pour cela nous devons le télécharger via l'outil `rustup` qui est un outil permettant de gérer les versions de Rust ainsi que ses outils.

Si vous êtes sous Linux ou Mac OS, ouvrez votre terminal et écrivez la commande suivante :

```
1 curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

Cette commande va télécharger un script qui va installer l'outil `rustup` qui lui va procéder à l'installation de Rust. Si tout se déroule bien vous devriez voir le message suivant :

```
1 Rust is installed now. Great!
```

Si vous êtes sous Windows, il suffit de se rendre sur le site officiel et de suivre les instructions.

Pour vous assurer que l'installation s'est bien déroulée, vous pouvez lancer la commande suivante dans votre terminal :

```
1 rustc --version
```

Vous devriez voir message suivant :

```
1 rustc x.y.z (abcabcabc yyyy-mm-dd)
```

Celui-ci indique le numéro de version, le hash de commit de la version ainsi que la date de publication de celle-ci.

Mise à jour et désinstallation

Comme je l'ai dit, `rustup` permet de gérer les versions de Rust, si vous souhaitez mettre à jour celui-ci, il suffit de lancer la commande suivant dans votre terminal :

```
1 rustup update
```

Il en est de même pour la désinstallation, il suffit de lancer la commande suivant dans votre terminal :

```
1 rustup self uninstall
```

Et le code dans tout ça ?

Passons aux choses sérieuses et voyons à quoi ressemble du code écrit en Rust.

Le traditionnel hello world

Commençons par le traditionnel hello world. Ouvrez donc votre éditeur préféré, créez un fichier `main.rs` et écrivez le code suivant :

```
1 fn main() {
2     println!("Hello, world!");
3 }
```

Observons la structure de ce code qui n'est pas bien compliqué à comprendre. Pour déclarer une fonction, nous utilisons le mot-clé `fn` suivi du nom de la fonction. Nous avons donc ici une fonction appelée `main` qui est une fonction un peu particulière puisque chaque programme Rust possède une fonction `main` qui est le point d'entrée du programme. Pour celles et ceux venant du langage C/C++ vous ne devriez pas être perdu.

La fonction `main` contient l'instruction suivante `println!("Hello, world!");` qui permet simplement d'afficher le message `Hello, world!` sur la sortie standard.

Note : Vous pouvez remarquer un `!` à la fin de la fonction `println` qui indique que celle-ci n'est pas une fonction, mais une macro. Nous n'allons pas rentrer dans les détails de ce qu'est une macro nous le verrons peut-être dans un futur article.

Le corps d'une fonction est placé entre des accolades `{ }` et chaque instruction se termine par un point-virgule `;`

Compilons maintenant notre code. Pour cela, écrivez la commande suivante dans votre terminal :

```
1 rustc main.rs
```

Il ne se passe rien ? Mais si voyons, nous avons vu que Rust était un langage compilé et que l'étape de compilation permet de créer un binaire exécutable. Si vous regardez dans votre dossier, vous devriez voir un fichier `main` (ou `main.exe` sur Windows). Pour exécuter le binaire, écrivez simplement la commande suivante si vous êtes sur Linux ou MacOS :

```
1 ./main
```

Si vous êtes sous Windows, la commande est similaire, mais il faut ajouter l'extension `.exe` :

```
1 .\main.exe
```

Vous devriez voir s'afficher le message `Hello, world!`

Bravo, vous venez d’écrire votre premier programme avec Rust.

Petit tour des bases

Pour continuer, faisons un petit tour des bases de Rust.

Les variables

Les variables en Rust sont par défaut immuables (immutable en anglais) c’est-à-dire qu’il n’est pas possible de changer leur valeur après leurs déclarations. Pour déclarer une variable en Rust il suffit d’utiliser le mot-clé `let` :

```
1 let value = 42;
```

Essayons de changer la valeur de notre variable value :

```
1 let value = 42;
2 value = 0;
```

Tentons maintenant de compiler :

```
1 rustc main.rs
```

On obtient ce joli message d’erreur :

```
1 error[E0384]: cannot assign twice to immutable variable `value`
2 --> main.rs:3:5
3 |
4 2 |     let value = 42;
5 |         ----
6 |         |
7 |         first assignment to `value`
8 |         help: make this binding mutable: `mut value`
9 3 |     value = 0;
10 |         ^^^^^^^^^ cannot assign twice to immutable variable
11
12 error: aborting due to previous error;
13
14 For more information about this error, try `rustc --explain E0384`.
```

Le compilateur de Rust est vraiment génial, les erreurs sont explicites. Celui-ci nous indique qu’il n’est pas possible d’assigner une seconde fois une valeur à une variable immuable. Il nous explique même comment régler notre souci. Nous devons utiliser le mot-clé `mut` avant le nom de la variable lors de sa déclaration pour rendre celle-ci mutable :

```
1 let mut value = 42;
2 value = 0;
```

Il existe également le mot-clé `const` pour déclarer une constante. La première différence avec le mot-clé `let` est que la valeur d’une constante doit être connue à la compilation, il n’est donc pas possible d’avoir une valeur provenant d’un appel de fonction. La seconde différence est que le type d’une constante doit également être déclaré explicitement :

```
1 const a: i32 = 42;
2 let c = return_i32_value(); // OK
3 const b: i32 = return_i32_value(); // Erreur
```

Les types primitifs

Rust pratique ce qu’on appelle l’inférence de type qui permet au compilateur de déduire automatiquement les types associés aux expressions, telle qu’une déclaration de variable par exemple, sans qu’ils soient indiqués explicitement

dans le code. Il est tout de même possible d'indiquer le type d'une variable. Reprenons notre exemple précédent et indiquons le type de la variable `value` :

```
1 let value: i32 = 42;
```

La variable `value` est de type `i32` ce qui correspond à un entier représenter sur 4 octets (32 bits).

Voyons plus en détail les types primitifs de Rust.

bool

Le type pour les valeurs booléennes est `bool`. Ce type peut prendre les valeurs `true` ou `false` :

```
1 let x = true;
2 let y: bool = false;
```

char

Le type `char` correspond à un seul caractère :

```
1 let a = 'a';
2 let b: char = 'b';
```

Faites attention à bien utiliser le caractère simple quote `'` et non pas le caractère double quote `"` qui lui représente une chaîne de caractères. Contrairement à d'autres langages (comme le C par exemple) le type `char` de Rust est représenté sur 4 octets pour le support d'Unicode.

Les entiers signés

Les entiers signés contiennent des valeurs entières qui peuvent être positives ou négatives. Nous distinguons cinq types d'entiers signés :

Type	Valeur min	Valeur max
i8	-128	127
i16	-32768	32767
i32	-2147483648	2147483647
i64	-9223372036854775808	9223372036854775807
i128	-170141183460469231731687303715884105728	170141183460469231731687303715884105727

```
1 let x = 42; // Par défaut le type est i32
2 let y: i8 = -42;
```

Les entiers non signés

Les entiers non signés contiennent uniquement des valeurs entières positives. Nous distinguons cinq types d'entiers non signés :

Type	Valeur min	Valeur max
u8	0	255
u16	0	65535
u32	0	4294967295
u64	0	18446744073709551615
u128	0	340282366920938463463374607431768211455

```
1 let x : u8 = 42;
2 let y: u16 = 1400;
```

Les nombres décimaux

Il existe deux types pour les nombres décimaux `f32` et `f64` qui correspondent respectivement à un nombre décimal représenté sur 4 octets (32 bits) et 8 octets (64 bits).

```
1 let x = 1.42; // Par défaut le type est f64
2 let y: f32 = 1000.01;
```

Les tableaux

Les tableaux en Rust ont une taille fixe et ne peuvent contenir que des valeurs de même type.

```
1 let a = [1, 2, 3]; // Un tableau de 3 entiers de type i32
2 let b: [i8; 3] = [4, 5, 6]; // [type; taille], ici un tableau de 3 éléments représentés sur 1 octet (8 bits)
3
4 let mut c: [i32; 3] = [7, 8, 9]; // Un tableau mutable permet de changer ses valeurs, mais pas sa taille
5 c[0] = 10;
6 c[1] = 11;
7 c[2] = 12;
8 c[3] = 13; // Provoque une erreur : index out of bounds: the length is 3 but the index is 3
9
10 let d = ['d'; 5]; // [valeur; taille], ici un tableau de 5 caractères 'd'
11
12 println!("{:?}", d); // {:?} permet d'afficher le contenu du tableau
```

Note: Il est possible d’avoir des “tableaux” dynamiques en Rust, il faut pour cela utiliser la structure `Vec`. Il existe également d’autres types de collections disponibles dans la librairie standard de Rust comme les listes chaînées ou encore les tables de hachages.

Les tuples

Un tuple est une liste ordonnée d’éléments pouvant être de types différents. Les développeurs Python connaissent déjà bien ce type de données.

```
1 let a = (true, 'a', 1.5); // Crée un tuple de 3 éléments (bool, char, f64)
2 let b: (char, i32, f64, bool) = ('b', 1, 1.5, false); // Crée un tuple de 4 éléments (char, i32, f64, bool)
3
4 let mut c = (42, 10.5); // Un tuple mutable permet de changer ses valeurs, mais pas sa taille
5 c.0 = 10; // Attention la valeur doit être de même type que celle de la déclaration
6 c.1 = false; // Provoque une erreur : expected floating-point number, found `bool`
7
8 let (d, e, f) = a; // d = true, e = 'a', f = 1.5
9 let (g, _, _, h) = b; // g = 'b' true, h = false
10
11 println!("{:?}", a); // {:?} permet d'afficher le contenu du tuple
```

slice

Sans rentrer dans les détails et pour faire simple, le type `slice` est une référence (d’où la présence du caractère `&` qui n’est pas nouveau pour les développeurs C++) sur une partie d’une structure de données comme les tableaux par exemple :

```
1 let a: [i8; 5] = [1, 2, 3, 4, 5];
2
3 let b: &[i8] = &a; // On crée un "slice" sur l'ensemble du tableau
4 let c = &a[0..3]; // [from..to] On crée un "slice" de l'élément 0 jusqu'à l'élément 2
5 let d = &a[..]; // On crée un "slice" sur l'ensemble du tableau
6
7 let e = &a[1..4]; // [2, 3, 4]
8 let f = &a[1..]; // [2, 3, 4, 5]
9 let g = &a[..3]; // [1, 2, 3]
```

Note: L’opérateur `..` permet de définir un intervalle d’exclusion. Par exemple `0..3` représente les éléments de 0 à 3 (exclus). Au contraire, l’opérateur `..=` définit quant à lui un intervalle d’inclusion. `0..=3` représente donc les éléments de 0 à 3 (inclus).

Les chaînes de caractères

Pour déclarer une chaîne de caractères il suffit d’utiliser les doubles quote `"` :

```
1 let a = "Rust c'est cool !";
2 let b: &str = "Vraiment cool !";
```

Les chaînes de caractères (`str`) en Rust sont bien plus complexes qu’elles ont en l’air. Pour cet article nous n’allons donc pas rentrer dans les détails.

Les opérateurs

Nous n'allons pas nous attarder sur les différents opérateurs. Vous pouvez retrouver ceux-ci dans le tableau ci-dessous.

Les opérateurs arithmétiques	+ - * / %
Les opérateurs de comparaison	== != < > <= >=
Les opérateurs logiques	! &&
Les opérateurs d'affectations	= += -= *= /= %= &= = ^= <<= >>=
Les opérateurs bit-à-bit	& ^ << >>

Les structures de contrôle

Nous allons maintenant nous intéresser aux structures de contrôle.

if - else if - else

Comme pour la plupart des langages on retrouve le traditionnel trio: `if`, `else if` et `else`.

```
1 // if
2 let is_required = true;
3
4 if is_required {
5     // ...
6 }
7
8 // if / else
9
10 let value = 10;
11
12 if value % 2 == 0 {
13     println!("{}", is even", value);
14 } else {
15     println!("{}", is odd", value);
16 }
```

Il n'existe pas de condition ternaire comme dans la plupart des langages, mais il est possible de combiner `let` et `if/else` comme ceci :

```
1 let value = 10;
2 let is_even = if value % 2 == 0 { true } else { false };
```

match

Il n'existe pas d'instruction `switch` en Rust, et c'est pas plus mal, car je la trouve personnellement trop verbeuse et souvent source d'erreur avec l'oubli d'un break par exemple. À la place, Rust utilise ce qu'on appelle le pattern matching via l'instruction `match` qui est bien plus puissante qu'un simple `switch`.

Voici un premier exemple avec l'instruction `match` qui se comporte comme un `switch` classique :

```
1 let language = "FR";
2
3 match language {
4     "FR" => println!("Français"),
5     "EN" => println!("Anglais"),
6     "DE" => println!("Allemand"),
7     "ES" => println!("Espagnol"),
8     _ => println!("Langue inconnue") // Equivalent au default du switch
9 }
```

Tout comme l'instruction `if`, il est également possible de procéder à une affectation :

```
1 let has_access = true;
2
3 let role = match has_access {
4     true => "Admin",
5     false => "User"
6 };
```


Mais l'instruction `match` ne s'arrête pas là, elle propose des comparaisons beaucoup plus poussées :

```
1 let value = 5;
2
3 match value {
4     0 => println!("Zéro"),
5     1 | 2 | 3 => println!("Un, deux ou trois"),
6     4..=10 => println!("Entre quatre et dix"),
7     x if x > 10 => println!("Supérieur à 10"),
8     _ => println!("Autres")
9 };
```

Tout ceci n'est qu'un avant-goût de ce qu'il est possible de réaliser avec l'instruction `match`.

while

L'instruction `while` permet de créer une boucle qui continue tant que sa condition est respectée comme pour les autres langages :

```
1 let mut count = 0;
2
3 while count <= 100 {
4     println!("{}", count);
5     count += 1;
6 }
```

loop

L'instruction `loop` permet de créer une boucle infinie (oui il existe certains cas où une boucle infinie est utile).

```
1 loop {
2     println!("Boucle infinie!");
3 }
```

C'est plus simple que de l'écrire avec l'instruction `while` :

```
1 while true {
2     println!("Boucle infinie!");
3 }
```

Pour arrêter une boucle `loop` il suffit de faire appel à l'instruction `break` :

```
1 let mut count = 0;
2
3 loop {
4     println!("{}", count);
5
6     count += 1;
7
8     if count == 100 {
9         println!("STOP!");
10        break;
11    }
12 }
```

for

L'instruction `for` de Rust se rapproche de l'instruction `for` de Python. Elle permet de parcourir une à une les valeurs d'une expression renvoyant un itérateur (pour faire simple) :

```
1 // Parcours des valeurs de 1 à 10
2 for value in 0..=10 {
3     println!("{}", value);
4 }
5
6 // Parcours des valeurs de 0 à 10 avec un pas de 2
7 for value in (0..=10).step_by(2) {
8     println!("{}", value);
9 }
10
11 // Parcours d'un tableau
12 let arr = [1, 2, 3, 4];
13
14 for value in &arr {
15     println!("{}", value);
16 }
```

Les fonctions

Nous avons déjà vu avec notre premier exemple que chaque programme Rust doit avoir une fonction principale nommée `main` qui est le point d'entrée du programme. Mais il est bien entendu possible de définir d'autres fonctions :

```
1 // Fonction sans paramètre ni valeur de retour
2 fn print_hello_world() {
3     println!("Hello, world!");
4 }
5
6 // Fonction avec paramètres, mais sans valeur de retour
7 fn print_sum(x: i32, y: i32) {
8     println!("{}", x + y, x, y, x + y);
9 }
10
11 // Fonction avec paramètres et valeur de retour
12 fn sum(x: i32, y: i32) -> i32 {
13     x + y
14     // Il n'est pas nécessaire de faire appel à l'instruction return
15     // x + y est l'équivalent de return x + y;
16 }
17
18 // Fonction avec paramètres et valeur de retour
19 fn sum_with_return(x: i32, y: i32) -> i32 {
20     return x + y;
21     /* Sachez qu'il est préférable d'utiliser la forme précédente.
22      Il vaut mieux utiliser return uniquement quand on souhaite quitter
23      une fonction avant la fin de celle-ci
24      */
25 }
```

Il est également possible de définir des fonctions anonymes que l'on appelle des closures (à ne pas confondre avec les closures de JavaScript, ici cela ressemble plus aux arrow functions) :

```
1 // Avec déclaration des types
2 let sum = |x: i32, y: i32| -> i32 { x + y };
3
4 // Sans déclaration des types
5 let sum_without_types = |x, y| x + y;
```

Et la POO dans tout ça ?

Nous allons maintenant voir comment il est possible de faire de la programmation orientée objet avec Rust.

Les structures

Les structures sont ce qui se rapproche le plus des classes des autres langages de programmation. Pour créer une structure, il faut utiliser le mot-clé `struct` :

```
1 struct Rectangle {
2     width: u32,
3     height: u32,
4 }
```

L'instanciation se fera ensuite de la manière suivante :

```
1 let rect = Rectangle {
2     width: 10,
3     height: 20,
4 };
```

Pour accéder aux attributs de notre objet, il suffit de spécifier leur nom, comme il est coutume de faire en JavaScript :

```
1 println!("width: {}, height: {}", rect.width, rect.height);
```

Il est également possible de définir des méthodes sur nos structures à l'aide du mot-clé `impl`. Reprenons notre structure `Rectangle` et ajoutons deux méthodes, une méthode statique `new` servant de constructeur et une méthode non statique `print` permettant d'afficher la longueur et la largeur de nos objets `Rectangle` :

```
1 struct Rectangle {
2     width: u32,
3     height: u32,
4 }
5
```

```
6 impl Rectangle {
7     // Méthode statique : ici new sera utilisé comme constructeur
8     fn new(width: u32, height: u32) -> Rectangle {
9         Rectangle {
10             width: width,
11             height: height,
12         }
13     }
14
15     // Méthode non statique : Le paramètre self fait référence à l'instance courante
16     fn print(&self) {
17         println!("width: {}, height: {}", self.width, self.height);
18     }
19 }
```

Utilisons maintenant nos deux méthodes :

```
1 let rect = Rectangle::new(10, 20);
2
3 rect.print();
```

Note: Il n'existe pas de concept de constructeur en Rust comme nous avons l'habitude de le voir dans d'autres langages, la méthode statique `new` vue précédemment est simplement une convention.

Les traits

Il n'existe pas à proprement parler d'héritage en Rust à la place nous avons le concept de trait. Les traits permettent de définir des comportements à partager, mais également des comportements à implémenter comme pour les interfaces des autres langages de programmation. Déclarons un trait `Shape` avec une méthode `area` qui devra être implémentée par les structures utilisant ce trait :

```
1 trait Shape {
2     fn area(&self) -> u32;
3 }
```

La syntaxe pour l'implémentation d'un trait est la suivante `impl <trait> for <struct>`. Reprenons notre structure `Rectangle` et implémentons le trait `Shape` pour celle-ci :

```
1 impl Shape for Rectangle {
2     fn area(&self) -> u32 {
3         self.width * self.height
4     }
5 }
```

La méthode `area` peut ensuite être appelée sur les instances de la structure `Rectangle` :

```
1 let rect = Rectangle::new(10, 20);
2
3 println!("Area : {}", rect.area());
```

Il est tout à fait possible pour une structure d'implémenter plusieurs traits. Il reste encore beaucoup de choses à dire à propos des traits, mais cela vous donne déjà une bonne idée de leurs utilisations avec ce petit exemple.

Les modules

Rust propose un système de module permettant de créer des espaces de noms (ou namespace) et d'organiser son code en fichiers et dossiers.

Les espaces de noms

Commençons par voir la première utilisation des modules à savoir la création d'espace de noms. Créons donc un espace de noms `http` à l'aide du mot-clé `mod` :

```
1 mod http {
2     fn request(){
3         println!("Make a request !")
4     }
5 }
6
7 fn main() {
8     http::request();
9 }
```

Mais lorsque nous compilons, nous avons l'erreur suivante :

```
1 error[E0603]: function `request` is private
2 --> main.rs:8:11
3 |
4 8 |     http::request();
5 |         ^^^^^^^ private function
6 |
```

Par défaut toutes les fonctions (ou structures) se trouvant dans un module sont privées. Pour rendre celles-ci publiques il faut utiliser le mot-clé `pub` :

```
1 mod http {
2     // La fonction est maintenant publique
3     pub fn request() {
4         println!("Make a request !")
5     }
6 }
```

Il est également possible de rendre toutes les fonctions d'un module publiques en ajoutant le mot-clé `pub` avant le mot-clé `mod` :

```
1 // Toutes les fonctions du module seront publiques
2 pub mod http {
3     fn request() {
4         println!("Make a request !")
5     }
6 }
```

Séparer son code en plusieurs fichiers

Il est également possible de séparer son code en plusieurs fichiers. Pour cela créons un fichier `http.rs` et copions notre code, mais cette fois-ci sans utiliser le mot-clé `mod` :

```
1 pub fn request() {
2     println!("Make a request !")
3 }
```

Importons notre module via le mot-clé `mod` suivi du nom du fichier (ici `http`) dans le fichier `main.rs`, puis utilisons notre fonction `request` :

```
1 mod http;
2
3 fn main() {
4     http::request();
5 }
```

Nous pouvons également mettre notre module dans un dossier. L'avantage de cette méthode est qu'il est possible de séparer son code en plusieurs fichiers.

Pour cela il suffit de créer un dossier `http` et d'ajouter un fichier `mod.rs` (le nom est très important) contenant le code de notre module. Ajoutons également un fichier `status_message.rs` contenant une fonction permettant d'obtenir le message de statut d'une requête à partir du code HTTP :

```
1 pub fn from_code(code: i16) -> &'static str {
2     match code {
3         200 => "OK",
4         201 => "Created",
5         // ...
6         _ => "Unknown code"
7     }
8 }
```

Ce qui nous donne l'arborescence suivante :

```
1 .
2 └─ main.rs
3 └─ http
4     └─ status_message.rs
5     └─ mod.rs
```

Nous venons en fait de créer un nouveau module nommé `status_message`. Incluons ensuite celui-ci dans notre fichier `http.rs` :


```
1 // status_message est un sous-module du module http
2 pub mod status_message;
3
4 pub fn request() {
5     println!("Make a request !")
6 }
```

Nous pouvons maintenant utiliser notre module `http` ainsi que le sous-module `status_message` dans notre fichier `main.rs` :

```
1 mod http;
2
3 fn main() {
4     http::request();
5     println!("{}", http::status_message::from_code(200));
6 }
```

Il peut-être parfois lourd de devoir écrire les espaces de noms au complet, c'est pourquoi il est possible d'utiliser le mot-clé `use`. Si l'on reprend notre exemple précédent nous pouvons utiliser l'espace de nom `http` comme suit :

```
1 use http::{
2     request,
3     status_message::from_code,
4 };
5
6 mod http;
7
8 fn main() {
9     request();
10    println!("{}", from_code(200));
11 }
```

Cargo

Nous avons fait le tour des bases de Rust, mais il reste un point que j'aimerais vous montrer avant de terminer, il s'agit de l'outil Cargo. Cargo est le système de compilation et de gestion de paquets de Rust.

Pour l'utiliser, il suffit de créer un nouveau projet avec la commande suivante :

```
1 cargo new nom_du_projet
```

Cela crée automatiquement un dépôt Git de votre nouveau projet avec la structure suivante :

```
1 .
2 ├── Cargo.toml
3 ├── .git
4 ├── .gitignore
5 ├── src
6 │   └── main.rs
```

Pour le moment, nous n'avons que les fichiers et dossiers suivants :

- `Cargo.toml` : Il s'agit du fichier de configuration Cargo de votre projet. Ce fichier est au format TOML (Tom's Obvious, Minimal Language inventé par Tom Preston-Werner l'un des fondateurs de Github pour la petite info).

```
1 [package]
2 name = "nom_du_projet"
3 version = "0.1.0"
4 authors = ["Your Name <you@example.com>"]
5 edition = "2018"
6
7 [dependencies]
8 rand = "0.8.0"
```

Ce fichier est séparé en plusieurs parties :

- `package` : Liste l'ensemble des informations concernant votre paquet : le nom, le numéro de version, l'auteur, et l'édition de Rust à utiliser.
- `dependencies` : Liste l'ensemble des dépendances du projet que l'on appelle des crates.

D'autres parties existent, mais nous n'allons pas les aborder ici. Vous pouvez néanmoins aller jeter un coup d'œil sur la [documentation officielle](#).

- `src` : Le dossier qui contiendra votre code source. Cargo crée automatiquement le point d'entrée de votre programme à savoir le fichier `main.rs`

Compiler avec Cargo

Plutôt que de passer par la commande `rustc` pour compiler, il est possible d'utiliser Cargo avec la commande suivante :

```
1 cargo build
```

Cette commande va permettre de créer l'exécutable de votre programme dans le dossier `./target/debug`

Vous remarquerez également qu'un fichier `Cargo.lock` est créé. Comme pour le fichier `package-lock.json` d'un projet JavaScript ou `Composer.lock` d'un projet PHP, celui-ci va permettre de verrouiller les versions des dépendances.

Pour exécuter votre programme, il suffit de lancer la commande suivante :

```
1 cargo run
```

Cette commande permet également de compiler, si nécessaire, en plus d'exécuter votre programme.

Si vous souhaitez uniquement vérifier que votre code ne contient pas d'erreur de compilation, vous pouvez utiliser la commande suivante :

```
1 cargo check
```

Cette commande compile votre code sans créer d'exécutable et est donc beaucoup plus rapide que la commande `cargo build`.

Et la production ?

Comme je l'ai dit, la compilation de notre programme avec la commande `cargo build` produit un exécutable dans le dossier `target/debug`. Mais cet exécutable n'est pas optimisé pour la production. Si vous souhaitez distribuer votre programme ou l'utiliser en production, il suffit de lancer la commande `cargo build` avec l'option `release` comme ceci :

```
1 cargo build --release
```

Ajout de dépendances

Si vous souhaitez ajouter une dépendance à votre projet, il suffit de se rendre sur le site crates.io qui regroupe les modules créés par la communauté Rust et de rechercher ce qui vous intéresse. Une fois trouvé, il suffira de l'ajouter manuellement dans le fichier Cargo.toml dans la partie `dependencies`. Par exemple si vous souhaitez ajouter la librairie `rand` permettant de générer des nombres aléatoires, il suffit de l'ajouter comme ceci :

```
1 [package]
2 name = "nom_du_projet"
3 version = "0.1.0"
4 authors = ["Your Name <you@example.com>"]
5 edition = "2018"
6
7 [dependencies]
8 rand = "0.8.0"
```

Et de l'utiliser de cette façon dans votre code :

```
1 use rand::Rng;
2
3 fn main() {
4     let mut rng = rand::thread_rng();
5     println!("Nombre entier : {}", rng.gen_range(0, 10));
6     println!("Nombre décimal : {}", rng.gen_range(0.0, 10.0));
7 }
```

Note : Si la dépendance que vous utilisez n'est utile que pour le développement et non pour la production, vous pouvez l'ajouter dans une nouvelle partie `[dev-dependencies]`

Si vous souhaitez mettre à jour une dépendance, vous pouvez utiliser la commande suivante :

```
1 cargo update
```

Pour finir...


Nous avons vu dans cet article uniquement les bases de Rust et nous n’avons qu’effleuré la surface de ce qu’offre Rust tellement le langage est riche. La courbe d’apprentissage est certes ardue, contrairement à d’autres langages, mais le jeu en vaut clairement la chandelle. Si vous souhaitez sortir de votre zone de confort et n’avez pas peur d’apprendre de nouvelles choses, lancez-vous dans l’apprentissage de Rust, cet investissement vous sera bénéfique pour le futur.


J’espère que je vous ai donné envie d’en apprendre un peu plus. Si c’est le cas, je vous invite à lire [les ressources disponibles sur le site officiel](#). Nous, on se retrouve plus tard pour d’autres articles consacrés à Rust.

Annonces partenaire

Offres d'emploi pour développeurs Javascript

en partenariat avec WeLoveDevs.com





Ingénieur Fullstack JS

Synapse Medicine - Bordeaux, France


Il y a 27 jours

CDI

3 ans et +

Salaire d'un développeur Javascript

Offres d'emploi développeur Javascript



Developpeur web full stack


Divioseo - 100% Télétravail !

Il y a 24 jours

28k ➡ 43k

CDI

3 ans et +



Développeur front-end H/F


Lexfo - Paris, France - et 2 autres villes

Il y a 7 jours

38k ➡ 60k

CDI

2 ans et +



Développeur Logiciel H/F

PAARLY - Toulouse, France

Il y a 9 jours

42k ➡ 48k

CDI

3.5 ans et +

Vous êtes recruteur? [Diffusez vos annonces Javascript ici avec WeLoveDevs.com](#)

Arkerone

Développeur back (nodejs & php), je fais aussi du front (react). Je partage mes connaissances et ma passion au travers de mes articles. N'hésitez pas à me suivre sur [Twitter](#).

https://www.codeheroes.fr/2021/01/18/premiers-pas-avec-le-langage-rust/

15/18

découverte

rust

← RÉUSSIR SON ENTRETIEN TECHNIQUE #1 : LES LISTES CHAÎNÉES

MES PIRES (OU MEILLEURES) ANECDOTES EN TANT QUE DÉVELOPPEUR →

Arkerone

Développeur back (nodejs & php), je fais aussi du front (react). Je partage mes connaissances et ma passion au travers de mes articles. N'hésitez pas à me suivre sur [Twitter](#).

7 commentaires

axdev dit :

19 janvier 2021 à 22 h 59 min

Bonjour, très sympa de trouver un article sur les bases de Rust en Français: merci <3.

En passant, une petite coquille, il me semble que:

«La seconde différence est que le type d'une constante doit également être déclaré ~~im~~[ex]plicitement»

Répondre

Arkerone dit :

19 janvier 2021 à 23 h 08 min

Merci pour ton commentaire. Bien vu c'est corrigé 😊

Répondre

Jerry Wham dit :

20 janvier 2021 à 16 h 22 min

Merci pour cet article. J'avais commencé à me pencher sur la documentation officielle qui est très bien faite mais par manque de temps et sans projet actuellement pour mettre en pratique ce que j'y ai appris, j'ai un peu mis de côté. Mais ton article est un bon résumé.

Petites coquilles :

“D'autres langages comme Java ou Python par exemple, résous la plupart des problèmes de mémoire”. => “résolvent”

“Ouvrez donc votre éditeur préféré, créer un fichier main.rs” => “Ouvrez donc votre éditeur préféré, créez un fichier main.rs”

“Les variables en Rust sont par défaut immuables (immutables en anglais) c'est-à-dire qu'il n'est pas possible de changer leurs valeurs après leurs déclarations.” => “leur valeur après leur déclaration”

“Les chaînes de caractères (str) en Rust sont bien plus complexes qu'elles ont en l'air.” => “Les chaînes de caractères (str) en Rust sont bien plus complexes qu'elles en ont l'air.”

“structures de contrôles” => “structures de contrôle”

” (oui il existe certains cas ou une boucle infinie est utile).” => ” (oui il existe certains cas où une boucle infinie est utile).”

“// Fonction sans paramètres ni valeur de retour

fn print_hello_world() {}” => “// Fonction sans paramètre ni valeur de retour”

“Pour accéder aux attributs de notre objet, il suffit de spécifier leurs noms, ” => “Pour accéder aux attributs de notre objet, il suffit de spécifier leur nom, ”

“Cela créer automatiquement un dépôt Git de votre nouveau projet avec la structure suivante :” => “Cela crée automatiquement un dépôt Git de votre nouveau projet avec la structure suivante :”

“Pour le moment, nous avons que les fichiers et dossiers suivants :” => “Pour le moment, nous n'avons que les fichiers et dossiers suivants :”

“Cargo créer automatiquement le point d'entrée de votre programme à savoir le fichier main.rs” => “Cargo crée automatiquement le point d'entrée de votre programme à savoir le fichier main.rs”

“d'un projet PHP, celui-ci va permettent de verrouiller les versions des ” => “d'un projet PHP, celui-ci va permettre de verrouiller les versions des ”

“Si vous souhaitez uniquement vérifier que votre code ne contient pas d'erreurs de compilation” => “Si vous souhaitez uniquement vérifier que votre code ne contient pas d'erreur de compilation”

Répondre

Arkerone dit :

20 janvier 2021 à 16 h 40 min

Merci beaucoup c'est corrigé, décidément je devais être fatigué lors de ma relecture.

Répondre

Lilian dit :

23 janvier 2021 à 22 h 16 min

Bonsoir et merci pour cet article très intéressant pour une première approche.

Si j'ai bien compris, les slices se comportent comme en Python, c'est-à-dire que « 0..3 » par exemple prend en compte les éléments 0, 1 et 2 mais en excluant le 3 ?

Dans ce cas je pense qu'une petite précision au sujet d'un tel « gotcha » serait bienvenue ainsi, peut-être qu'une correction du commentaire de l'exemple lié à « 0..3 ».

Répondre

Arkerone dit :

23 janvier 2021 à 23 h 25 min

Merci pour ton commentaire. Les slices sont une référence sur une partie d'une structure de données comme les tableaux par exemple. L'opérateur ".." se comporte effectivement comme en Python. J'ajouterais prochainement une explication sur ces différents opérateurs.

Répondre

Sleipnir dit :

25 janvier 2021 à 19 h 57 min

Impatient de voir le 2ème pas 😊

Répondre

Laisser un commentaire

Votre adresse e-mail ne sera pas publiée. Les champs obligatoires sont indiqués avec *

Commentaire *

Nom *

E-mail *

Site web

LAISSER UN COMMENTAIRE

Ce site utilise Akismet pour réduire les indésirables. En savoir plus sur comment les données de vos commentaires sont utilisées.

TWITTER

GITHUB

CONTACT