

Rust FAQ

Par [Songbird_](#)  

Date de publication : 23 août 2016

Dernière mise à jour : 23 août 2016

TOUT PUBLIC

Cette FAQ est très certainement destinée à être modifiée. Si vous parvenez à débusquer une erreur dans les Q/R proposées, reportez-la au responsable de la rubrique ou à un mainteneur de la FAQ, s'il vous plaît.

Note : Il se pourrait qu'il y ait quelques confusions dans les Q/R - un deuxième passage sera fait quand une grande majorité des Q/R auront été écrites.

I - Introduction.....	4
I-A - Informations générales.....	4
I-A-1 - Stade de rédaction.....	4
I-A-2 - Présentation.....	4
I-A-3 - Affiliation.....	4
I-A-4 - Contribution.....	4
I-A-5 - Licence et condition d'utilisation.....	5
II - Langage.....	5
II-A - Questions générales.....	5
II-A-1 - Comment déclarer une variable ?.....	5
II-A-2 - Comment assigner un objet par référence ?.....	5
II-A-3 - Rust possède-t-il un typage dynamique ?.....	6
II-A-4 - Comment typer ses données/variables ?.....	6
II-A-5 - Quelle est la différence entre &str et String ?.....	6
II-A-6 - Comment créer une chaîne de caractères ?.....	7
II-A-7 - Quelle version de Rust est recommandée ?.....	7
II-A-8 - Rust est-il orienté objet ?.....	7
II-A-9 - Qu'est-ce qu'un « trait » ?.....	8
II-A-10 - Rust supporte-t-il la surcharge des fonctions ?.....	8
II-A-11 - Comment déclarer des paramètres optionnels ?.....	8
II-A-12 - Comment créer un tableau ?.....	9
II-A-13 - A quoi sert le mot-clé super ?.....	9
II-A-14 - A quoi sert le mot-clé self ?.....	9
II-A-15 - A quoi sert le mot-clé use ?.....	10
II-A-16 - A quoi sert le mot-clé pub ?.....	11
II-A-17 - A quoi servent les mot-clés extern crate ?.....	12
II-A-18 - A quoi sert le mot-clé mod ?.....	13
II-A-19 - A quoi sert un module ?.....	13
II-A-20 - Comment créer un module ?.....	13
II-A-21 - A quoi sert le mot-clé type ?.....	13
II-A-22 - A quoi sert le mot-clé loop ?.....	14
II-A-23 - A quoi sert le mot-clé where ?.....	14
II-A-24 - A quoi sert le mot-clé unsafe ?.....	15
II-A-25 - Quelles sont les règles non-appliquées dans ces contextes ?.....	15
II-A-26 - Quels comportements sont considérés « non-sûrs » par Rust ?.....	15
II-A-27 - A quoi sert le mot-clé fn ?.....	15
II-A-28 - A quoi sert le mot-clé match ?.....	16
II-A-29 - A quoi sert le mot-clé ref ?.....	17
II-A-30 - A quoi sert le mot-clé mut ?.....	17
II-A-31 - Une erreur survient lorsque que je modifie le contenu de ma variable ! Que faire ?.....	17
II-A-32 - Qu'est-ce qu'une macro ?.....	17
II-A-33 - Comment utiliser une macro ?.....	17
II-A-34 - Que sont les spécificateurs ?.....	19
II-A-35 - A quoi sert le mot-clé usize ?.....	19
II-A-36 - A quoi sert le mot-clé isize ?.....	19
II-A-37 - Existe-t-il des outils de build pour le langage Rust ?.....	19
II-A-38 - Comment utiliser mes fonctions en dehors de mon module ?.....	19
II-A-39 - Comment comparer deux objets avec Rust ?.....	20
II-A-40 - Qu'est-ce que le shadowing ?.....	20
II-A-41 - Qu'est-ce que la destructuration ?.....	20
II-A-42 - Comment effectuer une destructuration sur une liste ?.....	21
II-A-43 - Comment effectuer une destructuration sur une énumération ?.....	22
II-A-44 - Comment effectuer une destructuration sur une structure ?.....	22
II-A-45 - Comment comparer deux objets d'une structure personnalisée avec Rust ?.....	23
II-A-46 - Je n'arrive pas à utiliser les macros importées par ma bibliothèque ! Pourquoi ?.....	24
II-A-47 - A quoi servent les mot-clés if let ?.....	25
II-A-48 - A quoi servent les mot-clés while let ?.....	25
II-B - Mécaniques et philosophies.....	26

II-B-1 - Gestion de la mémoire.....	26
II-B-1-a - Le développeur doit-il gérer la mémoire seul ?.....	26
II-B-1-b - Qu'est-ce que « l'ownership » ?.....	26
II-B-1-c - Qu'est-ce que le concept de « borrowing » ?.....	28
II-B-1-d - Qu'est-ce que le concept de « lifetime » ?.....	29
II-B-1-e - Comment étendre un trait sur un autre trait ?.....	30
II-C - Outils de build.....	30
II-C-1 - Comment créer un projet avec Cargo ?.....	30
II-C-2 - Quel type de projet puis-je créer avec Cargo ?.....	31
II-C-3 - Comment compiler son projet ?.....	31
II-C-4 - Peut-on générer de la documentation avec Cargo ?.....	31
II-C-5 - Où trouver de nouvelles bibliothèques ?.....	32
II-C-6 - Comment installer de nouvelles bibliothèques ?.....	32
II-C-7 - Comment publier sa bibliothèque faite-maison ?.....	32
II-C-8 - Comment lancer des tests avec Cargo ?.....	34
II-C-9 - Comment mettre à jour mes bibliothèques ?.....	35
II-C-10 - Comment créer ses benchmarks avec Cargo ?.....	35
II-C-11 - A quoi sert benchmark_group! ?.....	36
II-C-12 - A quoi sert benchmark_main! ?.....	36
II-D - Gestion des erreurs.....	37
II-D-1 - Comment s'effectue la gestion des erreurs avec Rust ?.....	37
II-D-2 - Comment créer un type spécifique d'exceptions ?.....	37
II-D-3 - Est-il possible de créer des assertions ?.....	37
II-D-4 - A quoi sert la macro panic ! ?.....	38
II-D-5 - A quoi sert la méthode unwrap ?.....	38
II-D-6 - A quoi sert la méthode unwrap_or ?.....	39
II-D-7 - A quoi sert la méthode unwrap_or_else ?.....	39
II-D-8 - A quoi sert la méthode map ?.....	40
II-D-9 - A quoi sert la méthode and_then ?.....	40
II-D-10 - A quoi sert la macro try! ?.....	40
II-D-11 - Comment utiliser la macro assert! ?.....	40
II-D-12 - Comment utiliser la macro assert_eq! ?.....	41
II-D-13 - Comment utiliser la macro debug_assert! ?.....	42
II-D-14 - Qu'est-ce que l'énumération Result<T, E> ?.....	42
II-D-15 - Comment utiliser l'énumération Result<T, E> ?.....	42
II-D-16 - Qu'est-ce que l'énumération Option<T> ?.....	43
II-D-17 - Comment utiliser l'énumération Option<T> ?.....	43
II-E - Meta-données.....	43
II-F - I/O.....	44
II-G - Antisèches Rust.....	44
II-H - Trucs & astuces.....	44
II-H-1 - Que puis-je trouver dans cette section ?.....	44
II-H-2 - Comment récupérer le vecteur d'une instance de la structure Chars ?.....	44

I - Introduction

I-A - Informations générales

I-A-1 - Stade de rédaction

WIP (il n'existe aucune publication "propre" de cette FAQ pour le moment)

I-A-2 - Présentation

Cette FAQ a été conçue pour répondre, certes, aux questions les plus courantes, mais également pour paraphraser certaines explications fournies par la documentation officielle qui auraient pu être mal comprises.

Elle n'a en revanche pas pour but de traduire, mais bien de réexpliquer les passages qui pourraient s'avérer compliqués à comprendre de prime abord. Vous pourrez donc y trouver des explications complètes, mais aussi des liens vers la documentation officielle si, à l'inverse, vous venez de découvrir Rust et ne vous êtes pas encore rendu(e) vers cette dernière.

I-A-3 - Affiliation

Les ressources proposées par ce dépôt ne sont pas officielles ou affiliées à l'équipe en charge du projet Rust et/ou la fondation Mozilla. Ce document peut toujours contenir des erreurs et/ou confusions pouvant être invalidés; Bien que cette FAQ soit rédigée avec le plus grand soin, référez-vous toujours à la documentation officielle si vous avez un doute quant à la véracité des propos entretenus par cette ressource.

I-A-4 - Contribution

Vous souhaiteriez contribuer ? Super, nous vous remercions pour votre intérêt à l'égard de cette ressource. :)

Il existe actuellement plusieurs moyens de contribuer à la maintenance (ou à l'enrichissement) de ce repo:



Ce qui suit provient du [Source repo github](#). (d'où les différents moyens de contact)

- La façon la plus simple et directe de contribuer est la relecture orthographique du document. Pour ceci, récupérez le **fichier xml** et ne vous préoccupez que des paragraphes. (les méta-données ne sont pas importantes pour cette tâche.)
- Il est également possible pour vous de corriger le document xml en utilisant les outils proposés par **developpez.com**, vous évitant ainsi de modifier directement le document si sa lecture vous incommoder; Si cette méthode vous intéresse, n'hésitez pas à me contacter **ici** ou **ici**.
 - Aucun de ces liens ne vous convient pour me contacter ? Envoyez moi un mail à cet adresse: [chaacygg\[at\]gmail\[dot\]com](mailto:chaacygg[at]gmail[dot]com).
- Enfin, il vous est possible d'enrichir ce document en proposant de nouvelles Questions/Réponses ou tout simplement en créant de nouvelles sections accueillant d'autres types de ressources. Contrairement à la relecture et l'édition mineure, il serait plus sage d'opter pour utiliser le kit d'exportation que propose **developpez.com** pour vous évitez des tâches d'édérations fastidieuses.
- Cette solution ne vous convient pas ? Aucun problème, une version markdown de la FAQ va bientôt être publiée !

I-A-5 - Licence et condition d'utilisation



Ce qui suit provient du [Source repo github](#). (d'où les différents moyens de contact)

Des questions concernant l'utilisation de cette ressource ? Je vous invite à consulter le fichier LICENCE.md pour plus d'informations.

Les informations contenues dans le fichier ne vous suffisent pas ? Contactez-moi:

- [Twitter](#)
- [Profil developpez](#)

II - Langage

II-A - Questions générales

II-A-1 - Comment déclarer une variable ?

La déclaration d'une variable en Rust se fait par le biais du mot-clé `let`, permettant ainsi de différencier une assignation d'une expression.

Vous pouvez bien entendu déclarer et initialiser plusieurs variables en même temps de cette manière :

Assignation multiple

```
1. fn main()
2. {
3.     let (foo, bar, baz) = (117, 42, "Hello world!");
4. }
```

Ou effectuer une déclaration multiligne :

Déclaration multiligne

```
1. fn main()
2. {
3.     let foo = 117;
4.     let bar = 42;
5.     let baz = "Hello world!";
6. }
```

Voir aussi : [Rust possède-t-il un typage dynamique ?](#)

II-A-2 - Comment assigner un objet par référence ?

Il existe deux façons de faire :

- 1 Préciser par le biais du caractère `&`. (C-style)
- 2 En utilisant le mot-clé `ref` comme ceci :

```
1. fn main()
2. {
3.     let foo = 117i32;
4.     let ref bar = foo;
5.     let baz = &foo; //idem
```

```
6. }
```

II-A-3 - Rust possède-t-il un typage dynamique ?

Non.

Bien qu'il en donne l'air grâce à une syntaxe très aérée, Rust dispose d'un typage statique mais « optionnel » pour le développeur si il désire faire abstraction des types, mais il perdra, en toute logique, l'avantage de choisir la quantité de mémoire que sa ressource consommera.

Vous ne pouvez, par exemple, pas faire ceci :

```
1. fn main()
2. {
3.     let mut foo = 1;
4.     foo = " Hello world !";
5. }
```

Le type ayant été fixé par la première donnée, il n'est plus possible de changer en cours de route.

Voir aussi : [Comment typer ses données/variables ?](#)

II-A-4 - Comment typer ses données/variables ?

Pour les types primitifs, il existe deux manières de typer une variable :

Les différentes manières de typer

```
1. fn main()
2. {
3.     let foo : i32 = 117;
4. }
```

Ou :

Les différentes manières de typer

```
1. fn main()
2. {
3.     let bar = 117i32;
4. }
```

II-A-5 - Quelle est la différence entre &str et String ?

Du point de vue des packages, `String` se trouve dans le package `std::string` et `&str` dans le package `std`.

Du point de vue des types, `String` est un wrapper de `&str` et ce dernier est tout simplement l'alias représentant le type primitif des chaînes de caractères.

```
1. fn main()
2. {
3.     let foo : &str = "Hello world!"; //ça fonctionne
4.     let bar : String = foo; //erreur
5.     let baz : String = String::from(foo); //Ok !
6. }
```

II-A-6 - Comment créer une chaîne de caractères ?

La question pourrait paraître évidente dans d'autres langages, toutefois, après avoir écrit quelque chose de ce style :

Créer une chaîne de caractères

```
1. fn main()
2. {
3.     let foo : String = "Hello world!";
4. }
```

Le compilateur vous a renvoyé cette erreur :

Message d'erreur

```
|>
4 |>   let foo : String = "Hello world!";
    |>                        ^^^^^^^^^^^^^^^^^ expected struct `std::string::String`, found &-ptr
```

Il se trouve que la structure `String` est un wrapper.

Vous vous retrouvez donc à typer votre variable pour accueillir une instance de la structure `String` alors que vous créez une chaîne de caractères primitive.

Pour remédier au problème (si vous souhaitez malgré tout utiliser le wrapper), vous pouvez convertir une chaîne de caractères de type `&str` grâce à la fonction `String::from()` :

```
1. fn main()
2. {
3.     let foo : String = String::from("Hello world!");
4.     //ou
5.     let foo : &str = "Hello world!";
6. }
```

II-A-7 - Quelle version de Rust est recommandée ?

Actuellement (1), la version stable la plus récente est la **1.12.0**.

Mais vous pouvez toutefois utiliser une version un peu plus vieille.

Pour cette Q/R, la version de Rust sur mon poste était la **1.9.0**.



Gardez tout de même à l'esprit que si vous n'avez pas encore entrepris des projets qui vous tiennent à cœur avec Rust, il va de soit de se tourner vers la dernière version en date pour bénéficier des derniers ajouts/corrections.



La 1.9.0 comporte un bogue concernant les tests unitaires.

Voir aussi : [🇬🇧 Page officielle du langage Rust](#)

II-A-8 - Rust est-il orienté objet ?

Rust hérite des structures du C, elles n'incluent donc pas l'encapsulation des données comme nous pourrions l'imaginer avec une classe.

Il dispose d'un aspect de la POO, de prime abord, assez primitif ; Rust permet toutefois de bénéficier du polymorphisme grâce aux « traits » qui pourraient être comparées aux interfaces Java/C#.

Cependant, le langage ne supporte pas l'héritage multiple (ni l'héritage simple) entre les structures : comme il serait possible de le faire avec des classes.



Il est tout à fait possible de faire hériter plusieurs traits entre-eux, en revanche.

Voir aussi : [Qu'est-ce qu'un « trait » ?](#)

II-A-9 - Qu'est-ce qu'un « trait » ?

Un trait pourrait être comparé aux interfaces que l'on peut retrouver dans la plupart des langages orientés objet. (e.g. Java, C#)

Les traits vous permettent de déclarer des fonctions abstraites/virtuelles pour ensuite les implémenter au sein d'une structure grâce au mot-clé `impl` comme ceci :

```
1. trait Greeter
2. {
3.     fn greetings(&self);
4. }
5.
6. struct Person;
7.
8. impl Greeter for Person
9. {
10.    fn greetings(&self)
11.    {
12.        println!("Hello, my friends!");
13.    }
14. }
15.
16. fn main()
17. {
18.    let person = Person;
19.    person.greetings();
20. }
```

II-A-10 - Rust supporte-t-il la surcharge des fonctions ?

Rust ne supporte pas la surcharge des fonctions.

Le langage repose sur le « Builder Pattern » qui consiste à concevoir des « fabriques/factories » chargées de générer l'objet désiré.

Vous pouvez retrouver quelques explications à propos de ce design pattern  [ici](#) ou encore  [ici](#).

Voir aussi : [Comment déclarer des paramètres optionnels ?](#)

II-A-11 - Comment déclarer des paramètres optionnels ?

Il n'est pas possible de déclarer des paramètres optionnels avec Rust dans sa version actuelle.

Toutefois, il est toujours possible d'user de macros pour capturer différentes expressions et ainsi adapter votre code en fonction de la situation.

Le langage repose sur le « Builder Pattern » qui consiste à concevoir des « fabriques/factories » chargées de générer l'objet désiré.

Vous pouvez retrouver quelques explications à propos de ce design pattern  [ici](#) ou encore  [ici](#).

Voir aussi : [Comment utiliser une macro ?](#)

II-A-12 - Comment créer un tableau ?

Un tableau dans sa forme la plus primitive se déclare comme ceci :

Déclaration d'un tableau d'entiers

```
1. let foo : [i32; 10] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
```

Note : la taille du tableau doit être explicite, sous peine de recevoir une erreur de la part du compilateur.

II-A-13 - A quoi sert le mot-clé super ?

Contrairement à ce que l'on pourrait croire, le mot-clé `super` ne représente pas une référence vers l'instance courante d'une classe mère, mais représente seulement le « scope » supérieur. (dans un module)

Exemple :

Utilisation du mot-clé super

```
1. mod mon_module
2. {
3.     pub fn ma_fonction()
4.     {
5.         println!("Scope supérieur");
6.     }
7.     pub mod fils
8.     {
9.         pub fn fonction_enfant()
10.        {
11.            super::ma_fonction();
12.        }
13.    }
14.    pub mod fille
15.    {
16.        pub fn fonction_enfant()
17.        {
18.            super::ma_fonction();
19.        }
20.    }
21. }
22.
23. fn main()
24. {
25.     mon_module::fils::fonction_enfant();
26.     mon_module::fille::fonction_enfant();
27. }
```

II-A-14 - A quoi sert le mot-clé self ?

Le mot-clé `self` renvoie à une copie (ou la référence(`&self`)) de l'instance courante.

Il est souvent rencontré :

- lorsqu'une fonction virtuelle/abstraite est implémentée au sein d'une structure,

- lorsque le développeur doit utiliser une fonction dans le module courant,
- ...

Exemple :

Utilisation du mot-clé self

```
1. trait My_Trait
2. {
3.     fn my_func(&self);
4. }
5.
6. mod My_Mod
7. {
8.     fn foo()
9.     {
10.         self::bar();
11.     }
12.
13.     fn bar()
14.     {
15.
16.     }
17. }
```

II-A-15 - A quoi sert le mot-clé use ?

Le mot-clé `use` permet de raccourcir le « chemin » des dépendances du programme, vous évitant ainsi d'expliquer les dépendances de chacune de vos ressources.

Exemple :

Utilisation du mot-clé use

```
1. extern crate mon_package ;
2.
3. use mon_package::mon_module::ma_fonction ;
4.
5. fn main()
6. {
7.     ma_fonction() ;
8. }
```

Autrement dit, toute structure composée de différentes ressources peut être exploitée par le mot-clé `use`.

Exemple :

Utilisation du mot-clé use sur une énumération

```
1. enum MonEnum
2. {
3.     Arg1,
4.     Arg2,
5. }
6.
7. fn main()
8. {
9.     use MonEnum::{Arg1};
10.    let instance =
11.    Arg1; //plus la peine d'expliquer d'où provient l'instance Arg1 comme ceci:
12.    // let instance = MonEnum::Arg1;
13. }
```

II-A-16 - A quoi sert le mot-clé pub ?

Le mot-clé `pub` peut être utilisé dans *trois* contextes différents :

- 1 Au sein [et sur] des modules ;
- 2 Au sein [et sur] des traits ;
- 3 Au sein [et sur] des structures.

Dans un premier temps, qu'il soit utilisé sur des `modules`, `traits`, ou `structures`, il aura toujours la même fonction : rendre publique l'objet concerné.

Exemple :

Structure du projet

```

1. |— Cargo.lock
2. |— Cargo.toml
3. |— src
4. |   |— lib.rs
5. |   |— main.rs
6. |— target
7. |   |— debug
8. |       |— build
9. |       |— deps
10. |       |— examples
11. |       |— libmon_projet.rlib
12. |       |— mon_projet
13. |       |— native
  
```

Dans le fichier lib.rs

```

1. pub mod ma_lib //la module représentant ma bibliothèque
2. {
3.     pub mod mon_module //un module lambda
4.     {
5.         pub fn ma_fonction() //ma fonction
6.         {
7.             println!("Hi there !");
8.         }
9.     }
10. }
  
```

Dans le fichier main.rs

```

1. extern crate mon_projet;
2. use mon_projet::ma_lib::mon_module::ma_fonction;
3. fn main()
4. {
5.     ma_fonction();
6. }
  
```

Renvoie :

Console

```
1. Hi there !
```



Mais d'où provient « mon_projet » ?

« mon_projet » est le nom porté par votre projet dans le manifest Cargo.toml.

Pour cet exemple, voici le manifest rédigé :

Dans le manifest

```
1. [package]
2. name = "mon_projet"
3. version = "0.1.0"
4. authors = ["Songbird0 <chaacygg@gmail.com>"]
5.
6. [dependencies]
```

Quid lorsque pub est utilisé au sein de ces structures ?

Lorsque le mot-clé `pub` est utilisé au sein d'un `trait` ou d'une `structure` sur une fonction, cela rend cette dernière indépendante de l'instance d'un objet. (mais peut toujours être appelée par l'une d'elles)

Autrement dit, la fonction est statique.

```
1. struct A;
2.
3. impl A
4. {
5.     pub fn foo()
6.     {
7.         println!("Je suis statique !");
8.     }
9.
10.    pub fn new() -> A
11.    {
12.        return A;
13.    }
14.
15.    fn bar(&self)
16.    {
17.        println!("Fonction non-statique");
18.    }
19. }
20.
21. fn main()
22. {
23.     let instance : A = A::new();
24.     A::foo();
25.     instance.bar();
26.     //instance::bar() -> erreur
27. }
```

II-A-17 - A quoi servent les mot-clés `extern crate` ?

Les mot-clés `extern crate` permettent d'importer un paquet entier de modules dans le fichier courant.

Le principe est simple, il vous suffit seulement de créer en premier lieu un projet en mode « bibliothèque » pour réunir tous les modules que vous créerez, de créer un fichier qui accueillera le point d'entrée de votre programme, puis d'importer votre paquet.

Bien entendu, si vous souhaitez importer un paquet qui n'est pas de vous, il vous faudra l'inscrire dans votre manifest.

Voir aussi :

Pour voir un exemple de création de paquet, vous pouvez vous rendre à la Q/R : « [A quoi sert le mot-clé `pub` ?](#) »

Comment installer de nouvelles bibliothèques ?

II-A-18 - A quoi sert le mot-clé mod ?

Le mot-clé `mod` vous permet de créer un module.

Voir aussi :

[A quoi sert un module ?](#)

II-A-19 - A quoi sert un module ?

Il vous permet de réunir plusieurs objets (`structures`, `traits`, fonctions, d'autres `modules`...) dans un même fichier puis de les réutiliser à plusieurs endroits dans votre programme.

Voir aussi :

- [A quoi sert le mot-clé pub ?](#)
- [A quoi servent les mot-clés `extern crate` ?](#)

II-A-20 - Comment créer un module ?

Voici comment créer un `module` :

Utilisation du mot-clé `mod`

```
1. mod A
2. {
3.     fn votre_fonction() {}
4.     fn une_autre_fonction() {}
5.     mod B
6.     {
7.         struct C {}
8.         trait D {}
9.     }
10. }
```

II-A-21 - A quoi sert le mot-clé type ?

Le mot-clé `type` permet de créer des *alias* et ainsi réduire la taille des types personnalisés (ou primitifs).

Voici un exemple :

Utilisation du mot-clé `type`

```
1. struct VeryLongTypeName;
2.
3. impl VeryLongTypeName
4. {
5.     pub fn new() -> VeryLongTypeName
6.     {
7.         println!("In new function");
8.         return VeryLongTypeName;
9.     }
10. }
11.
12. type ShortName = VeryLongTypeName;
13.
14. fn main()
15. {
16.     let foo = ShortName::new();
17. }
```



Les alias ne fonctionnent pas pour les « traits », si vous essayez de lier un alias à l'identifiant d'un trait, le compilateur vous renverra une erreur. (très explicite)

Liens :

Pour exécuter l'exemple de la Q/R, vous pouvez vous rendre [ici](#).

Retrouvez des explications [ici](#).

Explications de la documentation officielle.

II-A-22 - A quoi sert le mot-clé loop ?

Le mot-clé `loop` est un sucre syntaxique qui permet de remplacer le fameux :

```

1. while(true)
2. {
3.
4. }
5.
6. // ou
7.
8. for(;;)
9. {
10.
11. }
```

Préférez donc cette syntaxe :

```

1. loop
2. {
3.
4. }
```

Liens :

Documentation officielle.

II-A-23 - A quoi sert le mot-clé where ?

Le mot-clé `where` permet de filtrer les objets passés en paramètres dans une fonction génériques, par exemple :

Utilisation du mot-clé where

```

1. trait Soldier{}
2. trait Citizen{}
3. struct A;
4. struct B;
5. impl Soldier for A{}
6. fn foo<T>(test: T) -> T where T: Soldier
7. {
8.     return test;
9. }
10. fn main()
11. {
12.     let soldier : A = A;
13.     let citizen : B = B;
14.     foo(soldier);
15.     foo(citizen); //error: the trait bound `B: Soldier` is not satisfied
16. }
```

II-A-24 - A quoi sert le mot-clé unsafe ?

Le mot-clé `unsafe` permet, comme son nom l'indique, de casser certaines règles natives de Rust pour effectuer des opérations « à risque ».

`unsafe` peut être utilisé dans quatre contextes différents :

La déclaration d'une fonction :

```
1. unsafe fn dangerous_function() {}
```

La création d'un nouveau scope :

```
1. fn main() → ()  
2. {  
3.     unsafe { /*dangerous scope*/ }  
4. }
```

La déclaration d'un trait :

```
1. unsafe trait Dangerous_trait{}
```

L'implémentation d'un trait :

```
1. unsafe impl A for B {}
```

En pratique, le mot-clé `unsafe` permet une manipulation de la mémoire plus approfondie, plus directe, mais aussi plus compliquée, puisque le langage n'applique pas certaines règles.

Voir aussi :

Quelles sont les règles non-appliquées dans ces contextes ?

Quels comportements sont considérés « non-sûrs » par Rust ?

II-A-25 - Quelles sont les règles non-appliquées dans ces contextes ?

Trois règles, et seulement trois, sont brisées dans les blocs (et fonctions) `unsafe` :

- 1 L'accès et la modification d'une variable globale (statique) muable sont autorisés ;
- 2 Il est possible de déréférencer un pointeur (non-nul, donc) ;
- 3 Il est possible de faire à une fonction non-sûre.

II-A-26 - Quels comportements sont considérés « non-sûrs » par Rust ?

Pour en retrouver une liste exhaustive, rendez-vous à la [🇬🇧 section dédiée](#).

II-A-27 - A quoi sert le mot-clé fn ?

En rust, pour déclarer une fonction, il faut utiliser le mot-clé `fn` :

```
1. fn ma_fonction()  
2. {
```

```
3.  
4. }
```

II-A-28 - A quoi sert le mot-clé match ?

Le mot-clé `match` nous permet d'implémenter le *pattern matching*.

Ainsi, il est possible de comparer une entrée à plusieurs tokens constants et agir en conséquence. Le pattern matching est considéré comme un test *exhaustif*, car, quoi qu'il arrive, il fera en sorte de couvrir tous les cas de figure qu'on lui propose.

Exemple :

Implémentation du pattern matching

```
1. let foo : i32 = 117;  
2. match foo  
3. {  
4.     117 => println!("foo's value equals 117 !"),  
5.     _ => println!("You know nothing, John."), //s'efforcera de trouver une réponse  
6. }
```

Jusqu'ici, il semblerait que le mot-clé `match` ne soit pas capable de faire preuve de plus de souplesse qu'un `switch`, ce qui est bien entendu le contraire !

Vous pouvez assigner le résultat de vos tests directement dans une variable sans avoir à l'écrire dans votre `switch/match`.


Exemple :

Assignation automatique

```
1. fn main()  
2. {  
3.     let foo : i32 = 117;  
4.     let mut bar : String;  
5.     match foo  
6.     {  
7.         117 => println!("foo's value equals 117 !"),  
8.         _ => println!("You know nothing, John."),  
9.     }  
10.  
11.     bar = match foo  
12.     {  
13.         117 => "It's ok !".to_string(),  
14.         _ => "foo isn't equals to 117".to_string(),  
15.     };  
16.  
17.     println!("{}", &bar);  
18. }
```

Voir aussi :

Vous pouvez exécuter l'exemple [ici](#).

Vous pouvez retrouver  [une source](#) abordant le pattern matching. (avec plusieurs exemples)

 [Partie de la documentation officielle abordant l'implémentation du pattern matching.](#)

II-A-29 - A quoi sert le mot-clé ref ?

Le mot-clé `ref` est une alternative au caractère spécial `&` pour expliciter le renvoi d'une référence d'un objet :

Utilisation du mot-clé ref

```
1. struct A ;
2. fn main()
3. {
4.     let foo : A = A ;
5.     let bar : &A = &foo ; // ou let ref bar = foo ;
6. }
```

II-A-30 - A quoi sert le mot-clé mut ?

Le mot-clé `mut` permet de rendre l'une de vos variable muables lors de sa déclaration.

Utilisation du mot-clé mut

```
1. let mut foo : i32 = 0 ;
2. let bar : i32 = 1 ;
3. foo = 1 ;
4. bar = 2 ; //erreur
```

II-A-31 - Une erreur survient lorsque que je modifie le contenu de ma variable ! Que faire ?

Il se peut que vous ayez omis la particularité de Rust : tout est immuable par défaut.

Pour permettre à une variable de modifier son contenu, il vous faudra utiliser le mot-clé `mut`.

Voir aussi : [A quoi sert le mot-clé mut ?](#)

II-A-32 - Qu'est-ce qu'une macro ?

Une macro est ce que l'on peut appeler vulgairement : une fonction très puissante.

Grâce aux macros, nous pouvons capturer *plusieurs* groupes d'expressions et ainsi écrire les instructions désirées selon *chaque* cas.

En Rust, c'est ce qui se rapproche le plus de la *surcharge de méthodes* en Java.

Voir aussi : [Comment utiliser une macro ?](#)

II-A-33 - Comment utiliser une macro ?

Pour utiliser une macro, il faut d'abord la déclarer en utilisant le mot-clé `macro_rules!`.



Attention, il ne doit pas y avoir d'espace(s) entre `macro_rules!` et le point d'exclamation.

Déclaration d'une macro

```
1. macro_rules! foo
2. {
3.     () => ();
4. }
```

Toutes les macros (y compris celle présentée ici) respectent une règle très importante : elles doivent toutes capturer au moins une expression pour être valide et compilées. (en l'occurrence, la regex `() => () ;`)

C'est donc cela, l'une des différences majeures entre une fonction/procédure et une macro : cette dernière est capable de capturer des expressions rationnelles, conserver en mémoire ce que désire le développeur, puis de les ré-utiliser dans le corps de l'une d'entre-elles.

Ces « super » fonctions demandent donc quelques notions liées aux expressions rationnelles pour vous permettre d'apprécier pleinement ce puissant mécanisme.

Voici un exemple très basique de macro :

Exemple d'utilisation d'une macro

```
1. /// **Attention**:  
2. ///  
3. ///Cette macro n'utilise qu'un seul type de spécificateur, mais il en existe beaucoup d'autres.  
4. macro_rules! foo  
5. {  
6.     ($your_name:expr, $your_last_name:expr, $carriage_return: expr) =>  
7.     {  
8.         if $carriage_return == true  
9.         {  
10.             println!("My name's {} {}.", $your_name, $your_last_name);  
11.         }  
12.         else { print!("My name's {} {}.", $your_name, $your_last_name); }  
13.     };  
14.  
15.     ($your_name:expr, $your_last_name:expr) =>  
16.     {  
17.         foo!($your_name, $your_last_name, false);  
18.     };  
19.  
20.     ($your_name:expr) =>  
21.     {  
22.         foo!($your_name, "", false);  
23.     };  
24. }  
25.  
26. fn main() -> ()  
27. {  
28.     foo!("Song", "Bird", true);  
29.     foo!("Song", "Bird"); //pas de retour à la ligne  
30.     foo!("Song"); //là non plus  
31. }
```

Vous aurez certainement remarqué que les paramètres passés sont assez spéciaux ; Au lieu d'avoir le nom de leur type après les deux points (« : »), il est écrit `expr`.

C'est ce que l'on appelle un « spécificateur » .

Liens :

Visionner le résultat de cet exemple.

Que sont les spécificateurs ?

II-A-34 - Que sont les spécificateurs ?

II-A-35 - A quoi sert le mot-clé `usize` ?

Le mot-clé `usize` permet de laisser le compilateur choisir la taille en mémoire d'un entier *non-signé*. (selon l'architecture de la machine sur laquelle le programme sera exécuté)

Voir aussi : [A quoi sert le mot-clé `isize` ?](#)

II-A-36 - A quoi sert le mot-clé `isize` ?

Le mot-clé `isize` permet de laisser le compilateur choisir la taille en mémoire d'un entier *signé*. (selon l'architecture de la machine sur laquelle le programme sera exécuté)

Voir aussi : [A quoi sert le mot-clé `usize` ?](#)

II-A-37 - Existe-t-il des outils de build pour le langage Rust ?

Rust dispose d'un outil de développement multifonction nommé Cargo.

Cargo est en premier lieu un gestionnaire de paquets (qui vous permet donc de télécharger des modules Rust développés par d'autres programmeurs) mais vous épaulé également dans la gestion, la construction de vos projets, la création de vos manifest, etc.

Un groupe de Q/R a été créé sur cette FAQ présentant une liste non-exhaustive de commandes supportées par Cargo suivie d'un exemple d'utilisation (vous pourrez également retrouver des exemples dans le manuel officiel de l'outil(`$ man cargo`)) :

- [Comment créer un projet avec Cargo ?](#)
- [Quel type de projet puis-je créer avec Cargo ?](#)
- [Comment compiler son projet ?](#)
- [Peut-on générer de la documentation avec Cargo ?](#)
- [Où trouver de nouvelles bibliothèques ?](#)
- [Comment installer de nouvelles bibliothèques ?](#)
- [Comment publier sa bibliothèque faite-maison ?](#)
- [Comment lancer des tests avec Cargo ?](#)
- [Comment créer ses benchmarks avec Cargo ?](#)
- [Comment mettre à jour mes bibliothèques ?](#)

II-A-38 - Comment utiliser mes fonctions en dehors de mon module ?

Pour utiliser vos fonctions en dehors de votre `module`, il vous faudra utiliser le mot-clé `pub`.

Voir aussi :

[A quoi sert le mot-clé `pub` ?](#)

[A quoi servent les mot-clés `extern crate` ?](#)

II-A-39 - Comment comparer deux objets avec Rust ?

Pour comparer deux objets avec Rust, vous pouvez utiliser la fonction `eq()` implémentée grâce au [trait](#) `PartialEq`.

Exemple :

Utilisation de la fonction `eq()`

```
1. fn main()
2. {
3.     let foo = 0;
4.     let bar = 0;
5.     let baz = foo.eq(&bar); //true
6.     let bazz = 'Hello world !';
7.     let bazzz = 'Hello world !'.to_string();
8.     let bazzzz = bazz.eq(&bazzz); //true
9. }
```

Voir aussi : [Comment comparer deux objets d'une structure personnalisée avec Rust ?](#)

II-A-40 - Qu'est-ce que le shadowing ?

Le shadowing consiste à faire abstraction des identificateurs qui pourraient être identiques à ceux se trouvant dans un scope (« champ ») plus petit, ou étranger à celui des autres identificateurs dans l'absolu.

Exemple :

```
1. fn main() -> ()
2. {
3.     let foo : &str = "Hello";
4.     {
5.         let foo : &str = "world!";
6.         println!("{}", &foo);
7.     }
8.     println!("{}", &foo);
9. }
```

La première déclaration de `foo` a été « éclipée » par celle se trouvant dans le deuxième scope. Lorsque cette dernière a été détruite (ou simplement mise hors d'accès, dans ce cas), la première déclaration de `foo` a été de nouveau opérationnelle.

Résultat :

Message sur la sortie standard

```
1. world!
2. Hello
```

II-A-41 - Qu'est-ce que la destructuration ?

Avec Rust, il est possible d'effectuer une « destructuration » sur certains types de données, mais qu'est-ce que cela signifie exactement ?

Grâce au pattern matching, il est possible de créer, donc, des « modèles » pour isoler une partie de la structure et ainsi vérifier si notre entrée correspond à nos attentes.

Une destrucuration peut se faire sur :

Les listes, les tuples ;

Les énumérations ;

Les structures.

Voir aussi :

- [Comment effectuer une destructuration sur une liste ?](#)
- [Comment effectuer une destructuration sur une énumération ?](#)
- [Comment effectuer une destructuration sur une structure ?](#)

II-A-42 - Comment effectuer une destructuration sur une liste ?

Ce n'est pas encore possible. (version 1.12.1)

Pour isoler une valeur comme nous allons le faire avec un tuple, ce n'est pas possible sans s'attirer les foudres du compilateur.



La fonctionnalité étant, pour le moment, à l'état expérimental, elle n'est pas tolérée par le compilateur pour une utilisation courante.

Nous ne parlerons donc ici que de la destructuration des tuples.

Pour isoler une valeur contenu dans un tuple, il faut d'abord écrire son modèle pour savoir où le chercher.

Par exemple, en assumant que nous cherchons une suite de chiffres dans un ordre croissant, il est simple de déterminer si cette suite est dans le bon ordre ou non.

Déstructuration tuple

```
1. let foo = ("one", "two", "three");
2. let bar = ("two", "one", "three");
3.
4. match bar
5. {
6.     ("one", x, "three") =>
7.     {
8.         if x == "two"
9.         {
10.            println!("tout est en ordre !");
11.        }
12.    },
13.    _ => println!("on dirait qu'il y a un problème dans votre tuple..."),
14. }
```

Lorsque vous construisez un modèle de ce type, gardez bien en tête que la valeur la plus à gauche représentera toujours la première valeur du tuple, et celle plus à droite représentera toujours la dernière valeur du tuple.

Rien ne vous empêche donc de faire ceci :

Isolation de plusieurs valeurs

```
1. let foo = ("one", "two", "three");
2. let bar = ("two", "one", "three");
3.
4. match foo
5. {
6.     ("one", x, y) =>
7.     {
8.         if (x, y) == ("two", "three") //on surveille plusieurs valeurs
9.         {
```

Isolation de plusieurs valeurs

```
10.         println!("tout est en ordre !");
11.     }
12. },
13.     _ => println!("on dirait qu'il y a un problème dans votre tuple..."),
14. }
```

II-A-43 - Comment effectuer une destructuration sur une énumération ?

Le pattern matching vous donne la possibilité de « décortiquer » une énumération, vous permettant ainsi d'effectuer des tests complets.

Voici un exemple :

Utilisation du pattern matching sur une énumération

```
1. pub enum Enum
2. {
3.     One,
4.     Two,
5.     Three,
6.     Four,
7. }
8.
9. fn foo(arg: Enum) -> ()
10. {
11.     match arg
12.     {
13.         Enum::One =>
14.         {
15.             println!("One");
16.         },
17.         Enum::Two =>
18.         {
19.             println!("Two");
20.         },
21.         Enum::Three =>
22.         {
23.             println!("Three");
24.         },
25.         Enum::Four =>
26.         {
27.             println!("Four");
28.         },
29.     }
30. }
31.
32. fn main()
33. {
34.     let (bar, baz, bazz, bazzz) = (Enum::One, Enum::Two, Enum::Three, Enum::Four);
35.
36.     foo(bar);
37.     foo(baz);
38.     foo(bazz);
39.     foo(bazzz);
40. }
```

II-A-44 - Comment effectuer une destructuration sur une structure ?

Tout d'abord, la question que nous pourrions nous poser est : en quoi consiste la destructuration sur une structure ?

L'idée est d'isoler, encore une fois, les propriétés qui nous intéressent.

Destructuration

```
1. struct A
2. {
```

Destructuration

```
3.     x: String,
4.     y: String,
5.     z: String,
6. }
7.
8. fn main() -> ()
9. {
10.     let foo = A {
11.         x: "Hello".to_string(),
12.         y: " ".to_string(),
13.         z: "world!".to_string(),
14.     };
15.     let A {x: a, y: b, z: c} = foo; //on décortique les attributs de notre structure
16.     println!("{}", a, b, c); //puis on les utilise dans de nouvelles variables
17. }
```

Vous souhaiteriez omettre un attribut ? Pas de problèmes !

Destructuration

```
1.     let foo = A {
2.         x: "Hello".to_string(),
3.         y: " ".to_string(),
4.         z: "world!".to_string(),
5.     };
6.     let A {x: a, y: b, ..} = foo; //on décortique les attributs de notre structure
7.     println!("{}", a, b); //puis on les utilise dans de nouvelles variables
```

Vous pouvez également isoler ce style d'opération dans un scope plus petit (empêchant l'utilisation des variables temporaires en dehors de ce dernier) comme ceci :

```
1.     let foo = A {
2.         x: "Hello".to_string(),
3.         y: " ".to_string(),
4.         z: "world!".to_string(),
5.     };
6.     {
7.         let A {x: a, y: b, z: c} = foo; //on décortique les attributs de notre structure
8.         println!("{}", a, b, c); //puis on les utilise dans de nouvelles variables
9.     }
10.
11.     //a,b et c ne pourront plus être utilisés à partir d'ici
```

II-A-45 - Comment comparer deux objets d'une structure personnalisée avec Rust ?

La bibliothèque standard de Rust propose un(e) **trait**/ interface nommé(e) **PartialEq** composée de deux fonctions :

```
1  fn eq(&self, other : &instance_de_la_meme_structure) ;
2  fn ne(&self, other : &instance_de_la_meme_structure) ;
```



Comme il est stipulé dans la documentation officielle, vous n'êtes pas forcé d'implémenter les deux fonctions : **fn ne()** étant simplement le contraire de **fn eq()** et vice versa, il serait redondant de les implémenter dans la même structure.

Ci-dessous figure un exemple complet d'implémentation :

Implémentation du trait PartialEq

```
1. struct Spartan<'a>
2. {
3.
4.     sid: i32,
```

Implémentation du trait PartialEq

```
5.     name: &'a str
6.
7. }
8.
9. impl<'a> PartialEq for Spartan<'a>
10. {
11.     fn eq(&self, other: &Spartan) -> bool
12.     {
13.         self.sid == other.sid
14.     }
15. }
16.
17. impl<'a> Spartan<'a>
18. {
19.
20.     pub fn new(sid: i32, name: &str) -> Spartan
21.     {
22.         Spartan
23.         {
24.             sid: sid,
25.             name: name,
26.         }
27.     }
28. }
29. fn main()
30. {
31.     let (foo , bar) = (Spartan::new(117, "John"), Spartan::new(062, "Jorge"));
32.
33.     if foo == bar
34.     {
35.         println!("foo equals bar");
36.     }
37.     else
38.     {
39.         println!("foo not equals bar");
40.     }
41. }
```

II-A-46 - Je n'arrive pas à utiliser les macros importées par ma bibliothèque ! Pourquoi ?

Il se pourrait que vous ayez omis d'utiliser une annotation : `#[macro_use]`

Cette dernière permet d'exporter toutes les macros qui doivent être publiques pour être utilisées à l'extérieur de la bibliothèque.

Exportation des macros

```
1. #[macro_use]
2. extern crate votre_lib;
3.
4. fn main() -> ()
5. {
6.     votre_macro!();
7. }
```

Si vous ne parvenez toujours pas à les utiliser, il est possible que vous ayez omis l'annotation `#[macro_export]` dans les modules comportant vos macros.

Préparation de l'exportation des macros

```
1. // dans le fichier lib.rs
2. #[macro_use]//bien préciser que ce module utilise des macros
3. pub mod votre_conteneur
4. {
5.     #[macro_export]
6.     macro_rules! foo
7.     {
```


Préparation de l'exportation des macros

```
8.     () => ();
9.     }
10.    #[macro_export]
11.    macro_rules! bar
12.    {
13.        () => ();
14.    }
15.    #[macro_export]
16.    macro_rules! baz
17.    {
18.        () => ();
19.    }
20. }
```

Si votre problème persiste, je vous invite à vous rendre sur les forums figurant dans la rubrique programmation pour obtenir de l'aide. Présentez clairement l'erreur que le compilateur vous renvoi dans votre post.

II-A-47 - A quoi servent les mot-clés if let ?

La combinaison des deux mot-clés permet d'assigner, de manière concise, du contenu à une variable.

Assignation avec if let

```
1. fn main() -> ()
2. {
3.     let foo : Option<String> = Some("Hello world!".to_string());
4.     let mut bar : bool = false;
5.
6.     if let Some(content) = foo // si la variable foo contient quelque chose...
7.     {
8.         bar = true;
9.     }
10.    else
11.    {
12.        println!("foo's content is None");
13.    }
14. }
```

C'est un moyen simple et efficace d'assigner du contenu sans passer par le pattern matching.

II-A-48 - A quoi servent les mot-clés while let ?

La combinaison des deux mot-clés permet d'effectuer des tests de manière concise et ainsi nous éviter de passer par le pattern matching lorsque ça n'est pas nécessaire. (**while let** peuvent s'avérer très utiles lorsqu'il faut tester à chaque itération si le fichier contient toujours quelque chose)

[\[Exemple de la documentation officielle\]](#)

Cas d'utilisation de while let

```
1. let mut v = vec![1, 3, 5, 7, 11];
2. while let Some(x) = v.pop() {
3.     println!("{}", x);
4. }
```

II-B - Mécaniques et philosophies

II-B-1 - Gestion de la mémoire

II-B-1-a - Le développeur doit-il gérer la mémoire seul ?

Cette FAQ dispose de trois Q/R abordant trois concepts distincts (mais se complétant) gravitant autour de la gestion de la mémoire avec le langage Rust.

Par souci de concision, les Q/R ci-dessous ne retiennent que l'essentiel de chaque concepts :

- 1 [Qu'est-ce que « l'ownership » ?](#)
- 2 [Qu'est-ce que le concept de « borrowing » ?](#)
- 3 [Qu'est-ce que le concept de « lifetime » ?](#)

II-B-1-b - Qu'est-ce que « l'ownership » ?



Cette Q/R abordant un concept propre au langage Rust, certains points pourraient encore vous paraître obscures après votre lecture. Si c'est le cas, vous pouvez vous reporter directement à la section, dédiée à ce sujet, de la [documentation officielle](#) du langage.

Si l'on fait abstraction du contexte dans lequel est employé ce terme (en l'occurrence, la programmation), nous pourrions le traduire de cette façon : « propriété », « possession ».

Nous verrons un peu plus bas que le fonctionnement de ce mécanisme n'est pas si étranger au sens littéral du terme.

Introduction

Rust est muni d'un système « d'appartenance » qui permet d'écartier les conflits les plus communs lorsqu'une ressource est utilisée à plusieurs endroits.

Bien que ce dernier soit très pratique, il demande d'avoir une certaine rigueur quant à la déclaration de nos ressources, sans quoi vous risqueriez de vous attirer les foudres du compilateur.

Pour cela, voici un exemple d'erreur typique lorsque l'on débute sans réellement connaître les tâches effectuées par le « ramasse-miette » :

Transfert par copie

```
1. fn main()
2. {
3.     let foo : String = String::from("Hello world!");
4.     let bar : String = foo;
5.     let baz : String = foo; //erreur la ressource a été « déplacée »
6. }
```

Renvoyant une erreur de ce style :

Message d'erreur

```
1. error: use of moved value: `foo`
```


C'est un exemple simple, mais qui (dans nos débuts) peut être une véritable plaie : on ne comprend pas d'où vient l'erreur - tout est syntaxiquement correct, mais le compilateur n'a pas l'air satisfait.

C'est simple :

La variable `foo` étant un pointeur contenant l'adresse mémoire d'un objet `String`, il est courant de dire qu'il possède « l'ownership », il est le seul à pouvoir utiliser cette ressource.

C'est en copiant les informations relatives à l'objet `String` (en « déplaçant » ces informations dans une nouvelle variable, donc) que le *garbage collector* va faire son travail : détruire le pointeur `foo` pour attribuer « l'ownership » au nouveau pointeur de la ressource : `bar`.

C'est lorsque la variable `baz` essaie de copier les informations de `foo` que l'erreur survient : `foo` a déjà été détruit par le *garbage collector*.


 *Lorsque la ressource est dite « déplacée » ce n'est pas l'objet lui-même qu'il l'est, rien n'est recréé lors de cette destruction de pointeurs. On se contente ici de « binder » (rattacher) la référence de l'objet à un nouveau pointeur : il n'y a donc pas d'effets de bord indésirables. (temps de création, consommation CPU)*

Pour remédier au problème, il aurait simplement suffi de copier `bar` de cette manière :

```
1. fn main()
2. {
3.     let foo : String = String::from("Hello world!");
4.     let bar : String = foo;
5.     let baz : &String = &bar; //on récupère une référence
6. }
```

Tout est en règle, le compilateur ne râle plus, et si vous souhaitez afficher votre chaîne de caractères sur la sortie standard, rien ne vous en empêche !

Attention cependant :

 Cette règle ne s'applique qu'aux ressources dynamiques ; Etant prompt à être référencé à plusieurs endroits en même temps, Rust s'assure de toujours fournir la référence la plus récente de la ressource.

Pour les ressources statiques, leurs places dans la mémoire sont déjà "prévues" donc pas de problèmes.

Vous pouvez très bien écrire ceci :

contre-exemple

```
1. fn main()
2. {
3.     let foo = 42;
4.     let bar = foo;
5.     let baz = foo;
6. }
```

Quid des fonctions ?

Les fonctions obéissent aux mêmes règles que les pointeurs :

Lorsqu'une ressource est passée en paramètre par copie, la fonction « possède » la ressource, même lorsqu'elle a terminé de s'exécuter.



Si la ressource en question a été créée dynamiquement, elle sera systématiquement détruite lorsque la fonction aura terminé de s'exécuter ; Sinon, elle devient simplement inaccessible pour le restant de l'exécution.

Exemple :

```
1. fn my_func(my_string: String)
2. {
3.     let chars = my_string.chars();
4.     for letter in chars
5.     {
6.         println!("{}", &letter);
7.     }
8. }
9. fn main()
10. {
11.     let foo : String = String::from("The cake is a lie!");
12.     my_func(foo);
13.
14.     let chars = foo.chars(); //error
15.
16. }
```

Vous remarquerez donc ici que le pointeur **foo** a été détruit, la copie de la chaîne de caractères appartient désormais à la fonction.

Ne vous attardez pas sur le contenu de la fonction myfunc() ;



Ce n'est qu'un exemple parmi tant d'autres, gardez simplement à l'esprit que si la ressource est passée en paramètre par copie, le pointeur vers cette dernière est détruit.

Voir aussi : [Qu'est-ce que le concept de « borrowing » ?](#)

II-B-1-c - Qu'est-ce que le concept de « borrowing » ?



Cette Q/R abordant un concept propre au langage Rust, certains points pourraient encore vous paraître obscures après votre lecture. Si c'est le cas, vous pouvez vous reporter directement à la section, dédiée à ce sujet, de la [documentation officielle](#) du langage.

Il est courant de devoir partager une ressource entre plusieurs pointeurs pour effectuer diverses tâches.

Toutefois, plus une ressource est sollicitée, plus il y a de chance qu'elle soit *désynchronisée/invalidée* à un moment ou un autre. (c'est encore plus fréquent lorsque cette dernière est sollicitée par plusieurs fils d'exécution)

Rust remédie à ce problème grâce au « borrow checking », un système d'emprunts créant en quelque sorte des *mutex* chargés de limiter l'accès à une ressource et ainsi éviter les risques d'écritures simultanées.

Le borrow checker fera respecter ces trois règles (que vous pouvez retrouver dans la documentation officielle) :

- 1 Une (ou plusieurs) variable peut emprunter la ressource en lecture. (référence immuable)
- 2 Un, et **seulement un**, pointeur peut disposer d'un accès en écriture sur la ressource.
- 3 Vous ne pouvez pas accéder à la ressource en lecture et en écriture en même temps, exemple :

Emprunt interdit

```
1. fn main()
2. {
3.     let mut foo = 117;
4.     let bar = &mut foo;
5.     let baz = &foo; //erreur
6.
7. }
```

Ou :

Deux accès en écriture

```
1. fn main()
2. {
3.     let mut foo = 117;
4.     let bar = &mut foo;
5.     let baz = &mut foo; //erreur
6. }
```

II-B-1-d - Qu'est-ce que le concept de « lifetime » ?



Cette Q/R abordant un concept propre au langage Rust, certains points pourraient encore vous paraître obscures après votre lecture. Si c'est le cas, vous pouvez vous reporter directement à la section, dédiée à ce sujet, de la [documentation officielle](#) du langage.

Introduction

Comme tous langages (sauf exception que nous pourrions ignorer), Rust dispose d'un système de durée de vie.

Toutefois, il fait preuve d'une grande rigourosité quant à la destruction des ressources dynamiques et à « l'isolement » des ressources statiques après utilisation.

Voici un exemple :

Démonstration des scopes

```
1. fn main()
2. {
3.     let mut foo : String = "Hello world!".to_string(); //Le scope A commence ici
4.     let bar : String = "Goodbye, friend !".to_string(); //Le scope B commence ici
5.     foo = bar; // bar détruit, le scope B s'arrête là
6.     println!("{}", &bar);
7. } // Le Scope A s'arrête ici
```

On remarque à la suite de cet exemple que le concept de « scope » (contexte) n'est pas à l'échelle d'une fonction, mais bien des variables, incitant le développeur à déclarer et initialiser sa ressource uniquement lorsqu'il en a besoin.

Quid des références ?

Le concept de durée de vie dédiée aux références peut parfois dérouter, surtout lorsqu'il faut expliciter certains tags (représentants des durées de vie) au compilateur lorsqu'il nous l'impose et que l'on ne comprend pas bien pourquoi.

Les références n'échappent pas à la règle, elles aussi ont des durées vie bien déterminées ; En règle générale, il n'est pas utile (voire interdit) au développeur d'expliquer les tags qui permettent au compilateur de « suivre » chaque référence durant son utilisation.

Cependant, lorsque l'une d'elles est passée en paramètre à une fonction, il peut parfois être nécessaire de tagger celles qui survivront au moins à l'exécution de la fonction. (ne serait-ce que par souci de clareté)

Voici un exemple qui pourrait vous épauler : (attention à bien lire les commentaires)

```
1. fn foo(phrase: &str) -> () //aucune référence ne survi, donc pas la peine de l'annoter
2. {
3.     println!("{}", &phrase);
4. }
5.
6. fn bar<'a>(phrase: &'a mut String, word: &str) -> &'a
   String //une référence va survivre il faut maintenant savoir laquelle
7. {
8.     phrase.push_str(word);
9.     return phrase;
10. } //La référence qui survivra sera donc « phrase », elle dispose donc de la durée de vie 'a.
11.
12. fn main()
13. {
14.     let mut baz : String = "Hello ".to_string();
15.     let word : &str = "world!";
16.     let bazz = bar(&mut baz,
17. word); //ce que contient la variable bazz ne peut être accédé qu'en lecture
17.     println!("{}", &bazz); //nous affichons nos caractères sur la sortie standard
18. }
```

En revanche, ce n'est pas un cas commun, nous vous invitons donc à vous tourner vers la documentation officielle ou à expérimenter par vous-même.

Que faut-il retenir ?

Pour faire simple, il faut retenir que :

- Chaque variable crée un nouveau scope lors de sa déclaration ;
- Toutes variables retrouvées dans le scope d'une autre verra sa durée de vie plus courte que cette dernière ;
- A propos des références passées en paramètres, seules les références survivant au moins à la fin de l'exécution de la fonction devraient être annotées.

Voir aussi :

Le Rustonomicon

La section dédiée du livre

II-B-1-e - Comment étendre un trait sur un autre trait ?

II-C - Outils de build

II-C-1 - Comment créer un projet avec Cargo ?

Pour créer un nouveau projet avec Cargo, vérifiez d'abord qu'il est *installé* sur votre machine :

```
$ cargo -V
```

Puis :

```
$ cargo new nom_de_votre_repertoire
```

Vous devriez voir se générer un dossier avec le nom assigné dans lequel se trouvera un répertoire nommé `src` et un manifest nommé `Cargo.toml`.

II-C-2 - Quel type de projet puis-je créer avec Cargo ?

Lorsque vous lancez la commande de génération (telle qu'elle), votre projet est généré en mode « bibliothèque », et n'est donc pas destiné à être directement exécuté.

Si vous souhaitez générer un projet en mode « exécutable », il suffit de le préciser dans la commande :

```
$ cargo new folder_name --bin
```

Par défaut, le nom du répertoire racine sera également le nom de votre bibliothèque si elle devait être identifiée par d'autres utilisateurs dans le but de la télécharger. Si vous souhaitez lui attribuer un autre nom, vous pouvez également le spécifier dans la commande :

```
$ cargo new folder_name --name another_name --bin
```

Le manifest sera modifié en conséquence.

II-C-3 - Comment compiler son projet ?

Pour compiler votre projet, vous devez vous trouver à la racine de ce dernier.

Une fois que c'est fait, il vous suffit de lancer la commande suivante :

```
$ cargo build
```

Par défaut, cargo compile votre projet en mode « debug », empêchant le compilateur d'effectuer des optimisations trop agressives.



Lorsque vous souhaitez envoyer votre binaire en production, vous pouvez utiliser l'option « --release » comme ceci :

```
$ cargo build --release
```

II-C-4 - Peut-on générer de la documentation avec Cargo ?

Bien sûr !

Il suffit de lancer la commande `$ cargo doc` à la racine de votre projet.

La documentation se trouvera dans le dossier `./target/doc/...`

Où est l'index de mon site ?

Il se trouve dans le répertoire portant le nom de votre projet.



Notez que si vous avez ajouté des dépendences à votre projet, cargo générera également la documentation de celles-ci. (assurant alors un site uniforme et complet)

II-C-5 - Où trouver de nouvelles bibliothèques ?

Vous pouvez trouver d'autres bibliothèques sur le [site officiel](#) de Cargo.

Voir aussi : [Comment installer de nouvelles bibliothèques ?](#)

II-C-6 - Comment installer de nouvelles bibliothèques ?

Il y a deux manières de faire :

- 1 Les télécharger à partir de [crate.io](#) ;
- 2 Les télécharger directement à partir de leur dépôt github.

C'est selon vos préférences. (et surtout selon la disponibilité de la ressource)

Donc pour la première façon, rien de plus simple :

- Vous cherchez la bibliothèque que vous désirez sur le site ;
- Vous renseignez son nom dans votre manifest ;
- Compilez ;
- C'est prêt !

Pour la seconde :

- Cherchez le dépôt github de la bibliothèque désirée ;
- Notez le nom que porte cette bibliothèque dans son manifest ;
- Puis ajoutez cette ligne dans vos dépendences : `lib_name = {git = "url du dépôt"}` ;
- Compilez ;
- C'est prêt !

II-C-7 - Comment publier sa bibliothèque faite-maison ?

Les procédures étant très bien expliquées sur le site de [crates.io](#), nous vous invitons à vous rendre dans la [section dédiée](#).

Si vous souhaitez malgré tout lire les procédures sur la FAQ, en voici une traduction :

Une fois que vous avez une bibliothèque que vous souhaiteriez partager avec le reste du monde, il est temps de la publier sur [crates.io](#) !

La publication d'un paquet est effective lorsqu'il est uploadé pour être hébergé par [crates.io](#).

*Réfléchissez avant de publier votre paquet, car sa publication est **permanente**.*



*La version publiée ne pourra **jamais** être écrasée par une autre, et le code ne pourra être supprimé.*

En revanche, le nombre de versions publiées n'est pas limité.

Avant votre première publication

Premièrement, vous allez avoir besoin d'un compte sur crates.io pour recevoir un « token » (jeton) provenant de l'API. Pour faire ceci, visitez la page d'accueil et enregistrez-vous via votre compte Github. Ensuite, rendez-vous dans vos options de compte, et lancez la commande \$ cargo login suivi de votre token.

```
1. $ cargo login abcdefghijklmnopqrstuvwxyz012345
```

Cette commande va informer Cargo que vous détenez un token provenant de l'API du site. (il est enregistré dans le chemin suivant : ~/.cargo/config.)

Ce token doit rester secret et ne devrait être partagé avec personne. Si vous le perdez d'une quelconque manière, régénérez-le immédiatement.

Avant la publication du paquet

Gardez en tête que le nom de chaque paquet est alloué en respectant la règle du « premier arrivé, premier servi ». Une fois que vous avez choisi un nom, il ne pourra plus être utilisé pour un autre paquet.

Empaqueter le projet

La prochaine étape consiste à emballer votre projet de manière à être intelligible pour crates.io. Pour remédier à cela, nous allons utiliser la commande cargo package. Votre projet sera donc emballé sous la format *.crate et se trouvera dans le répertoire target/package/.

```
1. $ cargo package
```

En plus de cela, la commande package est capable de vérifier l'intégrité de votre projet en dépaquetant votre *.crate et le recompiler. Si la phase de vérification se passe sans problème, rien ne devrait être affiché dans votre terminal.

Toutefois, si vous souhaitez désactiver cette vérification avant l'envoi, il vous suffit d'ajouter le flag --no-verify.

Cargo va ignorer automatiquement tous les fichiers ignorés par votre système de versionning, mais si vous voulez spécifier un type de fichiers en particulier, vous pouvez utiliser le mot-clé exclude dans votre manifest :

[Exemple tiré de la [documentation officielle](#) de l'outil]

```
1. [package]
2. # ...
3. exclude = [
4.     "public/assets/*",
5.     "videos/*",
6. ]
```

La syntaxe de chaque élément dans ce tableau est ce que glob accepte. Si vous souhaitez créer une whitelist au lieu d'une blacklist, vous pouvez utiliser le mot-clé include.

[Exemple tiré de la [documentation officielle](#) de l'outil]

```
1. [package]
2. # ...
3. include = [
4.     "**/*.rs",
5.     "Cargo.toml",
6. ]
```

Maintenant que nous avons un fichier *.crate prêt à y aller, il peut être uploadé sur crates.io grâce à la commande cargo publish. C'est tout, vous venez de publier votre premier paquet !

```
1. $ cargo publish
```

Si vous venez à oublier de lancer la commande `cargo package`, `cargo publish` le fera à votre place et vérifiera l'intégrité de votre projet avant de lancer l'étape de publication.

Il se pourrait que la commande `publish` vous refuse votre première publication. Pas de panique, ce n'est pas très grave.

Votre paquet, pour être différencié des autres, doit compter un certain nombre de métadonnées pour renseigner vos futurs utilisateurs sur les tenants et aboutissants de votre projet, comme la licence par exemple.



Pour ceci, vous pouvez vous rendre [ici](#), et ainsi visionner un exemple simple des métadonnées à renseigner.

Relancez votre procédure `cargo publish`, vous ne devriez plus avoir de problème.

Un problème pour accéder à l'exemple ? En voici un autre :

Archétype de manifest valide

```
1. [package]
2. name = "verbose_bird"
3. version = "0.3.2"
4. authors = ["Songbird0 <chaacygg@gmail.com>"]
5. description = "An awesome homemade loggers pack."
6. documentation = "https://github.com/Songbird0/Verbose_Bird/blob/master/src/README.md"
7. homepage = "https://github.com/Songbird0/Verbose_Bird"
8. repository = "https://github.com/Songbird0/Verbose_Bird"
9.
10. readme = "README.md"
11.
12. keywords = ["Rust", "log", "loggers", "pack"]
13.
14. license = "GPL-3.0"
15.
16. license-file = "LICENSE.md"
17.
18. [dependencies]
```

Il se peut que vous rencontriez également des problèmes avec l'entrée « `license = ...` » vous informant que le nom de licence entré n'est pas valide.



Pour régler le souci rendez-vous sur [opensource.org](#) et visionner les noms raccourcis entre parenthèses de chaque licence.

II-C-8 - Comment lancer des tests avec Cargo ?

Pour lancer un test avec cargo, il vous faudra utiliser l'attribut `#[test]` et, évidemment, la commande `$ cargo test`.

Voici un exemple simple de tests :

```
1. #[cfg(test)]
2. mod oo_tests
3. {
4.     struct Alice;
5.     use loggers_pack::oop::Logger;
```

```

6.     impl Logger for Alice{ /*...*/ }
7.
8.     #[test]
9.     fn pack_logger_oop_info()
10.    {
11.        Alice::info("@Alice", "Hello, I'm Alice ", "Peterson !");
12.    }
13.
14.    #[test]
15.    fn pack_logger_oop_wan()
16.    {
17.        Alice::warn("@Alice", "Hello, I'm Alice ", "Peterson !");
18.    }
19.
20.    #[test]
21.    fn pack_logger_oop_error()
22.    {
23.        Alice::error("@Alice", "Hello, I'm Alice ", "Peterson !");
24.    }
25.
26.    #[test]
27.    fn pack_logger_oop_success()
28.    {
29.        Alice::success("@Alice", "Hello, I'm Alice ", "Peterson !");
30.    }
31. }

```

Chaque fonction annotée par l'attribut `#[test]` sera compilée durant la phase de test.



La version 1.9.0 de Rust comporte un bogue au niveau des tests. Dans cette version, toutes les fonctions annotées `#[test]` doivent être encapsulées dans un module. Ce n'est bien entendu plus le cas en 1.12.0.

Si vous rencontrez ce problème, nous vous conseillons de mettre à jour votre SDK.

II-C-9 - Comment mettre à jour mes bibliothèques ?

Pour mettre à jour vos dépendences, il vous suffit d'utiliser la commande : `$ cargo update`.

Vous pouvez également préciser quelle bibliothèque mettre à jour séparément en utilisation l'option `$ cargo update --precise nom_dep`



Faites tout de même attention avant de mettre à jour vos dépendences. Il se pourrait que les nouvelles versions cassent la compatibilité ascendante.

II-C-10 - Comment créer ses benchmarks avec Cargo ?



Avant toute chose, il est bon de savoir que la stabilité du module supportant les benchmarks est encore discutée au sein même de l'équipe chargée du développement de Rust.

N'étant pas disposé à fonctionner en production, vous devrez avoir recours à un paquet qui ne se trouve pas dans la bibliothèque standard, mais sur crates.io.

Pour créer nos benchmark, donc, nous allons utiliser le paquet **bencher**.

Ce module était premièrement connu sous le nom `test` puis `bencher` qui sera porté en tant que dépendance externe pour éviter les effets de bord dans les versions stables du langage.

Fichier Cargo.toml

```
1. [package]
2. name = "awesome_tests"
3. version = "0.1.0"
4. authors = ["Songbird0 <chaacygg@gmail.com>"]
5.
6. [dependencies]
7.
8. bencher = "0.1.1"
9.
10. [[bench]]
11. name = "my_bench"
12. harness = false
```

Voici un exemple basique de benchmark pour une fonction qui recherche le mot le plus court d'une phrase :

find_short function benchmark

```
1. #[macro_use]
2. extern crate bencher;
3. use bencher::Bencher;
4.
5. fn find_short(s: &str) -> usize {
6.     let splitting : Vec<&str> = s.split_whitespace().collect();
7.     let mut shortest_len : usize = 0;
8.     let mut i : usize = 0;
9.     while(i < splitting.len())
10.    {
11.        if(i == 0)
12.        {
13.            shortest_len = splitting[0].len();
14.        }
15.        else
16.        {
17.            if(splitting[i].len() < shortest_len)
18.            {
19.                shortest_len = splitting[i].len();
20.            }
21.        }
22.        i += 1;
23.    }
24.    return shortest_len;
25. }
26.
27. fn bench_find_short(b: &mut Bencher) {
28.     b.iter(|| find_short("Hello darkness my old friend"));
29. }
30.
31. benchmark_group!(my_bench, bench_find_short);
32. benchmark_main!(my_bench);
```

II-C-11 - A quoi sert `benchmark_group!` ?

La macro `benchmark_group!` sert à créer des « groupes » de fonctions à mesurer lors de l'exécution de la commande `cargo bench`.

II-C-12 - A quoi sert `benchmark_main!` ?

La macro `benchmark_main!` permet de créer une fonction `main` contenant toutes les fonctions à « benmarker ».



II-D - Gestion des erreurs







II-D-1 - Comment s'effectue la gestion des erreurs avec Rust ?

Tout comme les langages impératifs classiques (e.g. C), Rust ne gère pas les erreurs grâce à un système « d'exceptions » comme nous pourrions retrouver dans des langages plus orientés objets, mais grâce au contenu renvoyé en sortie de fonction.

Plusieurs fonctions (et macros) sont d'ailleurs dédiées à cette gestion (e.g. `panic!`, `unwrap()` (et ses dérivés), `and_then()`) permettant ainsi de rattraper (d'une manière plus ou moins fine) la situation lorsque les conditions imposées par vos soins ne sont pas respectées.

Cette section regroupe donc un certain nombre de Q/R qui pourrait vous aider à mieux cerner ce système de gestion :

 Toutes les Q/R taggées avec l'icône  sont considérées comme « en cours de rédaction ». Si vous rencontrez un problème de lecture avec ces Q/R inachevées, ne remontez pas le problème tant qu'elles disposent de ce tag. Merci.

- [A quoi sert la macro `panic!` ?](#)
- [A quoi sert la méthode `unwrap` ?](#)
- [A quoi sert la méthode `unwrap_or` ?](#)
- [A quoi sert la méthode `unwrap_or_else` ?](#)
- [A quoi sert la méthode `map` ?](#)
- 
- [A quoi sert la méthode `and_then` ?](#)
- [A quoi sert la macro `try!` ?](#)
- 
- [Comment utiliser la macro `assert!` ?](#)
- [Comment utiliser la macro `assert_eq!` ?](#)
- [Comment utiliser la macro `debug_assert!` ?](#)
- [Qu'est-ce que la structure `Option<T>` ?](#)
- 
- [Comment utiliser la structure `Option<T>` ?](#)
- 
- [Qu'est-ce que la structure `Result<T, E>` ?](#)
- 
- [Comment utiliser la structure `Result<T, E>` ?](#)
- 

II-D-2 - Comment créer un type spécifique d'exceptions ?

Il n'est pas possible de créer une structure censée représenter un type d'erreur, comme nous pourrions le faire en Java ; Rust ne gère pas les potentielles de cette manière.

Voir aussi :

Comment s'effectue la gestion des erreurs avec Rust ?

II-D-3 - Est-il possible de créer des assertions ?

Oui, bien entendu.

Il existe trois assertions différentes en Rust (toutes encapsulées par une macro) :

```
1 assert!;  
2 assert_eq!;  
3 debug_assert!.
```

Voir aussi :

- [Comment utiliser la macro `assert!` ?](#)
- [Comment utiliser la macro `assert_eq!` ?](#)
- [Comment utiliser la macro `debug_assert!` ?](#)

II-D-4 - A quoi sert la macro `panic!` ?

La macro `panic!` pourrait être comparée aux exceptions `RuntimeException` en Java qui sont, à coup sûr, des erreurs bloquantes.

Exemple de dead code

```
1. public class MyClass  
2. {  
3.     public static void main(String[] args)  
4.     {  
5.         throw new RuntimeException("Error !");  
6.         System.out.println("Dead code.");  
7.     }  
8. }
```

Elle est donc la macro la plus bas niveau que l'on peut retrouver parmi les macros et/ou fonctions proposées par la bibliothèque standard; Elle ne prend rien en compte mis à part l'arrêt du programme et l'affichage de la trace de la pile.

Exemple de dead code en Rust

```
1. fn main()  
2. {  
3.     panic!("Error !");  
4.     println!("Dead code");  
5. }
```

Voir aussi :

- [A quoi sert la méthode `unwrap` ?](#)
- [A quoi sert la méthode `and_then` ?](#)
- [A quoi sert la macro `try!` ?](#)

II-D-5 - A quoi sert la méthode `unwrap` ?

La méthode `unwrap()` permet de récupérer la donnée contenue par son wrapper et de faire abstraction des « cas d'analyse » avant de la délivrer.

Autrement dit, la méthode `unwrap()` délivre la donnée enveloppée si l'instance vaut `Some()` ou `Ok()`, sinon plante le programme si elle vaut `None` ou `Err()`.

Utilisation de la méthode `unwrap()`

```
1. fn main()  
2. {  
3.     let foo : Option<String> = Some("ça passe!".to_string());  
4.     let bar : Option<String> = None;  
5.     let baz : Result<String, String> = Ok("ça passe!".to_string());  
6.     let bing : Result<String, String> = Err("ça casse!".to_string());
```

Utilisation de la méthode unwrap()

```
7.
8.     println!("{}", foo.unwrap(), bar.unwrap(), baz.unwrap(), bing.unwrap());
9. }
```

Voir aussi :

- **Tester l'exemple** (Pensez à isoler les appels de la méthode si vous ne souhaitez pas faire planter votre programme.)
- Qu'est-ce que la structure `Option<T>` ?
- Qu'est-ce que la structure `Result<T, E>` ?

II-D-6 - A quoi sert la méthode unwrap_or ?

La méthode `unwrap_or()` fonctionne exactement comme la méthode originelle (2) mais permet d'éviter de faire « paniquer » le programme, et donc l'arrêt de l'exécution, en nous permettant de passer une valeur par défaut à renvoyer si le wrapper visé ne contient rien initialement.

Utilisation de la méthode unwrap_or()

```
1. fn main()
2. {
3.     let foo : Option<String> = Some("ça passe!".to_string());
4.     let bar : Option<String> = None;
5.     let baz : Result<String, String> = Ok("ça passe!".to_string());
6.     let bing : Result<String, String> = Err("ça casse!".to_string());
7.
8.     println!("{}", foo.unwrap(),
9.         bar.unwrap_or(String::from("ça passe, mais de justesse !")), baz.unwrap(),
10.        bing.unwrap_or(String::from("On évite de faire planter le programme.")));
11.     /*
12.     Pensez à isoler les appels de la méthode si vous ne souhaitez pas faire planter votre
13.     programme.
14.     */
15. }
```

Voir aussi :

Tester l'exemple

II-D-7 - A quoi sert la méthode unwrap_or_else ?

La méthode `unwrap_or_else` fonctionne exactement comme `unwrap_or`, mais proposera de passer en paramètre une fonction à la place d'une simple donnée.

Attention, `Option<T>` et `Result<T, E>` l'implémentent toutes les deux, mais pas de la même manière.



L'une l'implémente de manière à passer en paramètre une closure, l'autre à passer une fonction déclarée et identifiée dans un premier temps.

Utilisation de la méthode unwrap_or_else

```
1. fn bang(arg: String) -> String
2. {
3.     return "Chef, on a eu une erreur: ".to_string() + arg.as_str();
4. }
5. fn main()
6. {
7.     let foo : Option<String> = Some("ça passe!".to_string());
```

Utilisation de la méthode unwrap_or_else

```
8. let bar : Option<String> = None;
9. let baz : Result<String, String> = Ok("ça passe!".to_string());
10. let bing : Result<String, String> = Err("ça casse!".to_string());
11.
12. bar.unwrap_or_else(|| { return "On évite la casse !".to_string(); });
13. println!("{}", bing.unwrap_or_else(bang));
14. }
```

Note : le paramètre que reçoit la fonction `bang` n'est ni plus ni moins ce que vous avez renseigné dans le constructeur de l'instance `Err()` `bing`. Gardez ceci en tête lorsque vous souhaitez effectuer des opérations sur ce paramètre dans le corps de votre fonction.

II-D-8 - A quoi sert la méthode map ?

II-D-9 - A quoi sert la méthode and_then ?

La méthode `and_then()` permet d'effectuer des opérations sur la structure qui l'implémente, puis renvoie une nouvelle instance de cette dernière.

Utilisation de la méthode and_then()

```
1. fn concat(arg: &str) -> Option<String>
2. {
3.     Some(arg.to_string() + "world!")
4. }
5. fn main()
6. {
7.     let foo = Some("Hello ");
8.     println!("{}", foo.and_then(concat).unwrap());
9. }
```

Actuellement, les structures qui implémentent la méthode `and_then()` sont :

- `Option<T>`;
- `Result<T, E>`;

Voir aussi :

- A quoi sert la méthode `unwrap()` ?
- Qu'est-ce que la structure `Result<T, E>` ?
- Qu'est-ce que la structure `Option<T>` ?

II-D-10 - A quoi sert la macro try! ?

II-D-11 - Comment utiliser la macro assert! ?

La macro `assert!` capture deux types « d'expressions » différents :

Les expressions à proprement parler, qui pourraient être illustrées par les exemples suivants :

```
1. 2 * 2, if ... else ..., foo() ;
```

Les « tokens tree » qui pourraient être illustrés par n'importe quoi d'autres figurant dans la syntaxe du langage. (puisque, dans l'absolu, le compilateur représente tout ce qui est rédigé dans les fichiers sources grâce à une nomenclature bien à lui)

Donc si nous récupérons le code source raccourci de la documentation, cela donne ceci :

Source de la macro `assert!`

```
1. macro_rules! assert {
2.     ( $ cond : expr ) => { ... };
3.     (
4.     $ cond : expr , $ ( $ arg : tt ) + ) => { ... };
5. }
```

Si certaines choses vous échappent, n'hésitez pas à vous rendre sur les liens proposés en bas de cette Q/R.

A quoi sert le second paramètre ?

Le second peut, par exemple, accueillir un message personnalisé pour la macro `panic!` facilitant ainsi le débogage.

Utiliser la macro `assert!`

```
1. fn foo(arg: Option<String>) -> ()
2. {
3.     let bar : String = String::from("Hello world!");
4.     let mut some : Option<String> = None;
5.     assert!(!arg.is_none(), "Arg is None");
6.     assert!(arg.unwrap().eq(&bar), "arg n'est pas égal à bar");
7. }
8.
9. fn main() -> ()
10. {
11.     foo(Some("Ok".to_string()));
12.     foo(None);
13. }
```

Voir aussi :

- [Visionner le résultat de l'exemple](#) (requiert une connexion internet)
- Comment utiliser une macro ?
- [macro]Antisèche des sous-types

II-D-12 - Comment utiliser la macro `assert_eq!` ?

`assert_eq!` est un dérivé de la macro `assert!` et permet de tester directement l'égalité de deux objets (3) .

Bien entendu, elle hérite également du message personnalisé pour la macro `panic!`.

Utiliser la macro `assert_eq!`

```
1. fn foo(arg: Option<String>) -> ()
2. {
3.     let bar : String = String::from("Hello world!");
4.     let mut some : Option<String> = None;
5.     assert!(!arg.is_none(), "Arg is None");
6.     assert_eq!(arg.unwrap(), bar, "arg n'est pas égal à bar");
7. }
8.
9. fn main() -> ()
10. {
11.     foo(Some("Ok".to_string()));
12.     foo(None);
13. }
```

Voir aussi :

- [Visionner le résultat de l'exemple](#) (requiert une connexion internet)
- [Comment utiliser une macro ?](#)

II-D-13 - Comment utiliser la macro `debug_assert!` ?

Où puis-je l'utiliser ?

`debug_assert!` ainsi que ses dérivés (`debug_assert_eq!`) ne sont compilées que lorsque le code source est compilé en mode debug. (mode par défaut de **rustc**)

Vous ne devez pas compter sur ces assertions pour contrôler le flux de votre programme en production, assurez-vous toujours d'avoir une assertion compilée en mode release.



Si vous souhaitez toutefois les utiliser dans un binaire optimisé, vous devez passer l'argument `-C debug-assertions` au compilateur.

Comment l'utiliser ?

En dehors du contexte dans lequel ces assertions doivent être déclarées, la manière dont elles sont utilisées ne changent pas.

Voir aussi :

- [Comment utiliser la macro `assert!` ?](#)
- [Comment utiliser la macro `assert_eq!` ?](#)
- [Comment utiliser une macro ?](#)

II-D-14 - Qu'est-ce que l'énumération `Result<T, E>` ?

`Result<T, E>` est une énumération contenant deux constructeurs :

- 1 `Ok(T),`
- 2 `Err(E).`

Elle permet de gérer convenablement les cas où l'entrée `T` ne correspond pas à nos attentes et ainsi le communiquer au reste du programme pour que l'incident soit rattrapé plus loin si besoin.

Voir aussi : Comment utiliser la structure `Result<T, E>` ?

II-D-15 - Comment utiliser l'énumération `Result<T, E>` ?

L'utilisation de cette énumération requiert quelques notions quant à la gestion des erreurs avec Rust ; Ce dernier ne permettant pas l'utilisation des exceptions, cette structure vous permettra de conserver l'entrée si elle correspond à vos attentes, ou le message d'erreur si quelque chose ne s'est pas passé correctement.

Voici un exemple simple de gestion d'erreur :

Gestion d'erreur avec `unwrap()` seulement

```
1. fn foo<'a, 'b>(arg: Option<&'a str>) -> Result<String, &'b str>
2. {
3.     if let Some(content) = arg
4.     {
5.         let unwrapping = arg.unwrap();
6.         return Ok(unwrapping.to_string());
7.     }
8. }
```

Gestion d'erreur avec unwrap() seulement

```
9.     return Err("L'argument ne contient rien.");
10. }
11.
12. fn main()
13. {
14.     match foo(None)
15.     {
16.         Ok(content) => println!("Ok: {}", content),
17.         Err(err)   => println!("Error: {}", err.to_string()),
18.     }
19. }
```

Attention, cet exemple fonctionne mais il n'est pas recommandé d'opérer de cette manière pour gérer vos erreurs.



Dans un cadre autre que pédagogique, il est conseillé d'utiliser la macro « try! » pour envelopper les parties de notre code susceptible de lever une erreur.

Voir aussi :

A quoi sert la macro « try! » ?

A quoi sert la macro « panic! » ?

[Source](#) **Le résultat de cet exemple**

II-D-16 - Qu'est-ce que l'énumération Option<T> ?

Option est une énumération contenant deux constructeurs différents : Some(T) et None.

Option est en quelque sorte un wrapper, conteneur permettant de vérifier l'intégrité des données contenues.

II-D-17 - Comment utiliser l'énumération Option<T> ?

Pour utiliser les variantes de l'énumération, il faut savoir à quoi elles correspondent.

- Some(T) représente un binding valide ;
- None représente un binding invalide.

```
1. fn main()
2. {
3.     let foo : Option<String> = Some(String::from("Binding valide"));
4.     let bar : Option<String> = None; //binding invalide, ne contient rien
5. }
```

II-E - Meta-données



WIP

II-F - I/O



WIP

II-G - Antisèches Rust



WIP

II-H - Trucs & astuces

II-H-1 - Que puis-je trouver dans cette section ?

Vous pourrez retrouver des « trucs et astuces » pour résoudre un problème plus ou moins commun et complexe.

Ce qui signifie que si vous souhaitez ne serait-ce que conserver des notes quant aux manipulations requises pour se sortir d'un mauvais pas, d'un contexte qui prête à confusion, vos contributions sont les bienvenues dans cette section. :)



Les contributions publiées (au sein de cette section, comme dans la FAQ toute entière) sont relues, mais peuvent toujours contenir des erreurs ; N'hésitez pas à les signaler (en contactant le responsable de la FAQ ou un mainteneur, par exemple), ou à renvoyer une version corrigée de la Q/R en question. Merci.

II-H-2 - Comment récupérer le vecteur d'une instance de la structure Chars ?

Il est parfois nécessaire d'éclater une chaîne pour traiter ses caractères au cas par cas ; Jusqu'ici, Rust vous propose une méthode plutôt intuitive nommée `chars()`.

Après avoir éclatée la chaîne, vous souhaiteriez peut-être itérer plusieurs fois sur celle-ci, sans succès.

```
1. fn main()
2. {
3.     let foo = String::from("Hello");
4.     let bar = foo.chars();
5.
6.     for letter in bar {}
7.     for letter in bar {}
8. }
```

Erreur :

```
1. error[E0382]: use of moved value: `bar`
2. --> <anon>:7:19
3. |
4. 6 |     for letter in bar {}
5. |               --- value moved here
6. 7 |     for letter in bar {}
7. |               ^^^ value used here after move
8. |
```

La solution pourrait être la suivante :

Accès par référence

```
1. let foo = String::from("Hello");
2. let bar = foo.chars();
3.
4. for letter in &bar {}
5. for letter in &bar {}
```

```
1. error[E0277]: the trait bound `&std::str::Chars<'>: std::iter::Iterator` is not satisfied
2. --> <anon>:6:5
3. |
4. 6 |         for letter in &bar {}
5. |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
6. |
7. = note: `&std::str::Chars<'>` is not an iterator; maybe try calling `.iter()` or a similar
      method
8. = note: required by `std::iter::IntoIterator::into_iter`
```

Mais récoltez encore une erreur...

Le compilateur vous invite alors à essayer d'appeler la méthode `.iter()` qui est censée être implémentée par toutes les structures implémentant l'interface `Iterator`; Ce n'est malheureusement pas le cas pour la structure `Chars`.

Que faire alors ?

La méthode remplaçant `.iter()` est `.collect()`; Cette dernière vous permet de récupérer un vecteur contenant toutes les entités (4) de l'ancien itérateur.

Vous pouvez désormais accéder à votre ressource par référence et ainsi la parcourir autant de fois que vous le souhaitez.

Utilisation de la méthode `.collect()`

```
1. fn main()
2. {
3.     let foo = String::from("Hello");
4.     let bar = foo.chars();
5.     let baz : Vec<char> = bar.collect();
6.     for letter in &baz {}
7.     for letter in &baz {}
8. }
```

1 : **25 septembre 2016**

2 : unwrap

3 : Le terme « objet » est ici utilisé pour désigner toutes les entités pouvant être comparées à d'autres. (cela ne concerne donc pas que les instances des structures)

4 : En l'occurrence, ici, les caractères.