Introduction

cf. README.md, s'il vous plaît.

Aborescence de la FAQ

  - A quoi sert le mot-clé `isize` ?
  - Existe-t-il des outils de build pour le langage Rust ?
  - Comment utiliser mes fonctions en dehors de mon module ?
  - Comment comparer deux objets avec Rust ?
  - Qu'est-ce que le shadowing ?
  - Qu'est-ce que la destructuration ?
  - Comment effectuer une déstructuration sur une liste ?
  - Comment effectuer une déstructuration sur une énumération ?
  - Comment effectuer une déstructuration sur une structure ?
  - Comment comparer deux objets d'une structure personnalisée avec Rust ?
  - Je n'arrive pas à utiliser les macros importées par ma bibliothèque ! Pourquoi ?
  - A quoi servent les mot-clés `if let` ?
  - A quoi servent les mot-clés `while let` ?
- Mécaniques et philosophies
  - Gestion de la mémoire
  - Le développeur doit-il gérer la mémoire seul ?
  - Qu'est-ce que "l'ownership" ?
  - Qu'est-ce que le concept de "borrowing" ?
  - Qu'est-ce que le concept de "lifetime" ?
- Outils de build
  - 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 mettre à jour mes bibliothèques ?
  - Comment créer ses benchmarks avec Cargo ?
  - A quoi sert benchmark_group!
  - A quoi sert benchmak_main!
- Gestion des erreurs
  - Comment s'effectue la gestion des erreurs avec Rust ?
  - 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 l'énumération `Result<T>` ?
    - Comment utiliser l'énumération `Result<T>` ?
    - Qu'est-ce que l'énumération `Option<T, E>` ?
    - Comment utiliser l'énumération `Option<T, E>` ?
- Metadonnées/Annotations
- I/O
    - Que puis-je trouver dans cette section ?
    - Comment créer un fichier ?
    - Comment lire le contenu d'un fichier ?
    - Comment écrire à l'intérieur d'un fichier ?
    - Comment différencier un fichier d'un répertoire ?
    - Comment lister les objets d'un répertoire ?
- Trucs & astuces
- Que puis-je trouver dans cette section ?
- Comment récupérer le vecteur d'une instance de la structure `Chars` ?

Langage

Questions générales

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 :

```
fn main() {
    let (foo, bar, baz) = (117, 42, "Hello world!");
    // ou
    let foo = 117;
    let bar = 42;
    let baz = "Hello world!";
}
```

Voir aussi : Rust possède-t-il un typage dynamique ?

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 :

```rust
fn main() {
    let foo: i32 = 117;
    let ref bar = foo;
    let baz = &foo; // idem
}
```

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 ce dernier désire faire abstraction des types. Cependant il perdra, en toute logique, l'avantage de choisir la quantité de mémoire que sa ressource consommera.

Par exemple, vous ne pouvez pas faire ceci :

```rust
fn main() {
    let mut foo = 1;
    foo = "Hello world !";
}
```

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 ?

Comment typer ses données/variables ?

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

```rust
fn main() {
    let foo: i32 = 117;
}
```

Ou :

```rust
fn main() {
    let bar = 117i32;
}
```

Quelle est la différence entre &str et String ?

`&str` est un type immuable représentant une chaîne de caractères tandis que `String` est un wrapper mutable au-dessus de ce dernier.

```rust
fn main() {
    let foo: &str = "Hello world!"; // ça fonctionne
    let bar: String = foo; // erreur
    let baz: String = foo.to_owned(); // Ok !
    let baz: String = foo.to_string(); // Ok ! (équivalent de la ligne du dessus)
}
```

Comment créer une chaîne de caractères ?

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

```rust
fn main() {
    let foo: String = "Hello world!";
}
```

Le compilateur vous a renvoyé cette 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()` :

```rust
fn main() {
    let foo: String = String::from("Hello world!");
    // ou
    let foo: &str = "Hello world!";
}
```

Ou encore avec les méthodes `to_owned` et `to_string` (à préférer à la méthode `from` qui est un peu plus générale et donc plus lente) :

```rust
fn main() {
    let foo = "Hello world!".to_owned();
    let foo = "Hello world!".to_string();
}
```

Quelle version de Rust est recommandée ?

Actuellement(**27 février 2017**), la version stable la plus récente est la **1.15.0**.

Les versions antérieures à la 1.13.0 contenant des régressions, je vous conseille d'utiliser la version la plus récente proposée.

Voir aussi : Page officielle du langage Rust

Rust est-il orienté objet ?

Rust propose l'encapsulation qui est un concept objet. On peut donc dire que Rust est orienté objet. Toutefois, l'encapsulation s'effectue à l'échelle d'un **module** et non d'une **classe/structure** comme on pourrait le remarquer en Java/C#.

Il dispose d'un aspect de la POO, de prime abord, assez primitif; Rust permet 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, bien qu'il soit possible de le faire avec des traits.

Par conséquent, Rust est orienté objet, puisqu'il possède plusieurs parties de ce paradigme, mais n'est pas un langage objet *pur*.

Voir aussi : Qu'est-ce qu'un « trait » ?

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 :

```
trait Greeter {
    fn greetings(&self);
}

struct Person;

impl Greeter for Person {
    fn greetings(&self) {
        println!("Hello, my friends!");
    }
}

fn main() {
    let person = Person;
```

```
    person.greetings();
}
```

Pour aller au plus simple, un trait vous permet d'écrire un ensemble de fonctions qu'un objet est obligé d'implémenter lorsqu'il hérite de ce trait.

Rust supporte-t-il la surchage 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 ?

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 créer une macro ?

Comment créer un tableau ?

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

```
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.

À 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 le « scope » supérieur (dans un module).

Exemple :

```
mod mon_module {
    pub fn ma_fonction() {
        println!("Scope supérieur");
    }

    pub mod fils {
        pub fn fonction_enfant() {
            super::ma_fonction();
        }
    }

    pub mod fille {
        pub fn fonction_enfant() {
            super::ma_fonction();
        }
    }
}

fn main() {
    mon_module::fils::fonction_enfant();
    mon_module::fille::fonction_enfant();
}
```

A quoi sert le mot-clé self ?

Le mot-clé `self` est l'instance courante de votre type.

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 :

```
trait My_Trait {
    fn my_func(&self);
}

mod My_Mod {
    fn foo() {
        self::bar();
    }
```

```
    fn bar() {
    }
}
```

Il sert aussi à désigner le module courant lors d'un import. Par exemple :

```
use std::io::{self, File};

// maintenant on peut utiliser File mais aussi io !
```

A quoi sert le mot-clé use ?

Le mot-clé `use` permet de gérer les imports d'autres modules.

Exemple :

```
extern crate mon_package;

use mon_package::mon_module::ma_fonction;

fn main() {
    ma_fonction();
}
```

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

Exemple :

```
enum MonEnum {
    Arg1,
    Arg2,
}

fn main() {
    use MonEnum::Arg1; // On importe le variant Arg1 de l'enum MonEnum ici.

    let instance = Arg1; // Et on l'utilise directement ici.
    // Plus la peine d'expliciter d'où provient l'instance Arg1 comme ceci :
    let instance = MonEnum::Arg1;
}
```

Il permet aussi de réexporter des modules vers le scope supérieur. Prenons par exemple un projet possédant cette hiérarchie :

```
src
    fichier.rs
    video
 |          video.rs
```

```
|          mod.rs
|
   audio
          audio.rs
          mod.rs
```

Pour pouvoir accéder aux items présents dans `audio.rs` et `video.rs`, vous allez devoir les rendre visibles dans les niveaux supérieurs en les réexportant comme ceci :

```rust
// dans video/mod.rs
pub use self::video::{Video, une_fonction};

mod video;

// dans audio/mod.rs
pub use self::audio::{Audio, une_autre_fonction};

mod audio;
```

Dans `fichier.rs`, vous pourrez désormais faire :

```rust
use Audio;
use Video;
```

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 `mod`ules, `trait`s, ou `struct`ures, il aura toujours la même fonction : rendre publique l'objet concerné.

Exemple :

```
Cargo.lock
Cargo.toml
src
    lib.rs
    main.rs
target
    debug
        build
        deps
        examples
        libmon_projet.rlib
```

```
          mon_projet
          native

pub mod ma_lib { // le module représentant ma bibliothèque
    pub mod mon_module { // un module lambda
        pub fn ma_fonction() { //ma fonction
            println!("Hi there !");
        }
    }
}

extern crate mon_projet;

use mon_projet::ma_lib::mon_module::ma_fonction;

fn main() {
    ma_fonction();
}
```

Renvoie :

```
Hi there !
```

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

Pour cet exemple, voici le manifest rédigé :

```
[package]
name = "mon_projet"
version = "0.1.0"
authors = ["Songbird0 <chaacygg@gmail.com>"]

[dependencies]
```

Comment faire une méthode statique ?

Tout dépend de la présence de `self`/`&self`/`&mut self` en premier argument. Exemple :

```
struct A;

impl A {
    fn foo() { // ceci est une méthode statique
    }

    fn foo1(arg: i32) { // ceci est une méthode statique
    }

    fn foo2(&self) { // ceci n'est pas une méthode statique
    }
```

```
    fn foo3(self) { // ceci n'est pas une méthode statique non plus
    }

    fn foo4(&self, arg: i32) { // ceci n'est pas non plus une méthode statique
    }
}
```

À quoi servent les mot-clés “extern crate” ?

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

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 ?

À quoi sert le mot-clé mod ?

Le mot-clé `mod` vous permet d'importer ou de déclarer un module. Il est important de noter que les fichiers sont considérés comme des modules. Exemple :

```
mod a { // Ici on crée un module.
    fn foo() {}
}

mod nom_du_fichier; // Ici on importe le fichier "nom_du_fichier.rs".
```

Voir aussi :

A quoi sert un module ?

À 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 ?

Comment créer un module ?

Voici comment créer un `module` :

```
mod A {
    fn votre_fonction() {}
    fn une_autre_fonction() {}

    mod B {
        struct C;
        trait D {}
    }
}
```

À 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 :

```
use std::collections::HashMap;

type MonDico = HashMap<String, i32>; // On met un alias sur la HashMap.

fn main() {
    let x = MonDico::new(); // Plus besoin de s'embêter avec le long nom et ses paramètres
}
```

Liens :

Pour exécuter l'exemple de la Q/R, vous pouvez vous rendre ici.

Retrouvez des explications ici.

Explications de la documentation officielle.

À quoi sert le mot-clé loop ?

Le mot-clé `loop` sert à créer des boucles infinies. Pour faire simple, c'est un sucre syntaxique qui permet de remplacer le fameux :

```rust
while(true) {

}

// ou

for(;;) {

}
```

Préférez donc cette syntaxe :

```rust
loop {

}
```

Liens :

Documentation officielle.

À quoi sert le mot-clé where ?

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

```rust
trait Soldier {}
trait Citizen {}

struct A;
struct B;

impl Soldier for A {}

fn foo<T>(test: T) -> T
where T: Soldier {
    return test;
}

fn main() {
    let soldier: A = A;
    let citizen: B = B;
    foo(soldier);
    foo(citizen); // error: the trait bound `B: Soldier` is not satisfied
}
```

À 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 ».

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.

Pour faire simple : utilisez `unsafe` aussi peu que possible.

Exemple d'utilisation d'`unsafe` :

```
let x: i32 = &0;
let ptr = x as *const i32;
unsafe { *ptr; } // on tente d'accéder à l'élément pointé par le pointeur, ce qui est haute
```

Voir aussi :

Quelles sont les règles non-appliquées dans ces contextes ?

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

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) mutable sont autorisés;
2. Il est possible de déréférencer un pointeur (non-nul, donc);
3. Il est possible de créer une fonction non-sûre.

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

Pour en retrouver une liste exhaustive, rendez-vous à la section dédiée.

À quoi sert le mot-clé fn ?

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

```
fn ma_fonction() {

}
```

Pour spécifier le type renvoyé par la fonction, vous devez utiliser le token `->` + le type de votre choix.

```rust
fn ma_fonction() -> Foo
{
    Foo::new()
}
```

À 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 :

```rust
let foo: i32 = 117;

match foo {
    117 => println!("foo vaut 117 !"),
    _ => println!("You know nothing, John."), // s'efforcera de trouver une réponse
}
```

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 par-exemple matcher sur un ensemble de valeurs :

```rust
let foo: i32 = 117;

match foo {
    100...120 => println!("foo est entre 100 et 120 !"),
    x => println!("You know nothing, John. (foo vaut {})", x), // s'efforcera de trouver une
}
```

Le pattern matching est très puissant, n'hésitez pas à en user et en abuser !

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.

À 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 :

```
struct A;

fn main() {
    let foo: A = A;
    let bar: &A = &foo; // ou let ref bar = foo;
}
```

Il permet aussi de dire explicitement qu'une valeur ne doit pas être "bougée"/move dans certains contextes.

À quoi sert le mot-clé mut ?

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

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

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 ?

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.

Pour grossir un peu le trait : *les macros sont une extension du compilateur de Rust. Elles sont interprétées au moment de la compilation, pas pendant l'exécution de votre programme.*

Voir aussi : Comment créer une macro ?

Comment créer une macro ?

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

```
macro_rules! foo
{
    () => ();
}
```

Toutes les macros (y compris celle présentée ici) respectent une règle très importante : elles doivent toutes capturer au moins une expression pour être valides 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 :

```
// **Attention**:
//
// Cette macro n'utilise qu'un seul type de spécificateur, mais il en existe beaucoup d'aut
macro_rules! foo
{
    ($your_name:expr, $your_last_name:expr, $carriage_return: expr) =>
    {
        if $carriage_return == true
        {
            println!("My name's {} {}.", $your_name, $your_last_name);
        }
        else { print!("My name's {} {}.", $your_name, $your_last_name); }
    };

    ($your_name:expr, $your_last_name:expr) =>
    {
        foo!($your_name, $your_last_name, false);
    };

    ($your_name:expr) =>
    {
        foo!($your_name, "", false);
    };
}
```

```rust
fn main() {
    foo!("Song", "Bird", true);
    foo!("Song", "Bird"); // pas de retour à la ligne
    foo!("Song"); // là non plus
}
```

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 ?

Que sont les spécificateurs ?

Les spécificateurs sont des types d'entrées supportés par les macros; Il faut noter toutefois que ces fameux types font abstraction de la nature de l'entrée. (e.g. que l'entrée soit une châine de caractères ou un entier importe peu)

Vous pouvez retrouver, ci-dessous, une traduction de chaque description des spécificateurs proposés par le langage:

- `ident`: un identifiant. e.g. `x`; `foo`.
- `path`: une représentation d'un chemin "package-style"(e.g. `T::SpecialA`, `std::path::Path`).
- `expr`: une expression (e.g. `2 + 2`; `if true { 1 } else { 2 }`; `f(42)`).
- `ty`: Un type (e.g. `i32`, `&T`, `Vec<(char, String)>`).
- `pat`: Un modèle/pattern (e.g. `Some(t)`; `(17, 'a')`; `_`).
- `stmt`: Une (et une seule) déclaraction (e.g. `let some = 3`).
- `block`: Une suite de déclarations enveloppée par des crochets et/ou une expression (e.g. `{log(error, "hi"); return 12;}`).
- `item`: Un objet (e.g. `fn foo(){}`; `struct Bar;`).
- `meta`: Un "meta objet", plus connu sous le nom d'attributs (e.g. `cfg(target_os = "windows")`).
- `tt`: Un *token tree*. Certainement le type le plus puissant de la liste, puisqu'il vous permet de soumettre tout ce qui existe dans la syntaxe du langage.

Qu'est-ce qu'un item ?

À 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 compilé).

Voir aussi : À quoi sert le mot-clé isize ?

À 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 compilé).

Voir aussi : À quoi sert le mot-clé usize ?

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 épaule 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 ?

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 ?

Comment comparer deux objets avec Rust ?

Pour comparer deux objets avec Rust, vous pouvez implémenter le `trait PartialEq` que vous pourrez ensuite utiliser avec `==` ou la méthode `eq`.

Exemple :

```
fn main() {
    let foo = 0;
    let bar = 0;
    let baz = foo == bar; //true

    let bazz = "Hello world !";
    let bazzz = "Hello world !".to_string();
    let bazzzz = bazz == &bazzz; // true
    let bazzzz = bazz.eq(&bazzz); // équivalent de la ligne du dessus
}
```

Voir aussi : Comment comparer deux objets d'une structure personnalisée avec Rust ?

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 :

```
fn main() {
    let foo: &str = "Hello";
    {
        let foo: &str = "world!";
        println!("{}", &foo);
    }
    println!("{}", &foo);
}
```

La première déclaration de foo a été « éclipsée » par celle se trouvant dans le deuxième scope. Lorsque cette dernière a été détruite, la première déclaration de `foo` a été de nouveau opérationnelle.

Résultat :

```
world!
Hello
```

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 destructuration 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 ?

Comment effectuer une destructuration sur une liste ?

Pour isoler une valeur contenue 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.

```
let foo = ("one", "two", "three");
let bar = ("two", "one", "three");

match bar {
    ("one", x, "three") => {
        if x == "two" {
            println!("tout est en ordre !");
        }
    }
```

```
    _ => println!("on dirait qu'il y a un problème dans votre tuple..."),
}
```

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 :

```
let foo = ("one", "two", "three");
let bar = ("two", "one", "three");

match foo {
    ("one", x, y) => {
        if (x, y) == ("two", "three") { // on surveille plusieurs valeurs
            println!("tout est en ordre !");
        }
    },
    _ => println!("on dirait qu'il y a un problème dans votre tuple..."),
}
```

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 :

```
pub enum Enum {
    One,
    Two,
    Three,
    Four,
}

fn foo(arg: Enum) {
    match arg {
        Enum::One =>  println!("One"),
        Enum::Two => println!("Two"),
        Enum::Three => println!("Three"),
        Enum::Four => println!("Four"),
    }
}

fn main() {
    let (bar, baz, bazz, bazzz) = (Enum::One, Enum::Two, Enum::Three, Enum::Four);
```

```
    foo(bar);
    foo(baz);
    foo(bazz);
    foo(bazzz);
}
```

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.

```
struct A {
    x: String,
    y: String,
    z: String,
}

fn main() {
    let foo = A {
        x: "Hello".to_string(),
        y: " ".to_string(),
        z: "world!".to_string(),
    };
    let A { x: a, y: b, z: c } = foo; // on décortique les attributs de notre structure
    println!("{}{}{}", a, b, c); // puis on les utilise dans de nouvelles variables
}
```

Vous souhaiteriez omettre un attribut ? Pas de problèmes !

```
let foo = A {
    x: "Hello".to_string(),
    y: " ".to_string(),
    z: "world!".to_string(),
};

let A { x: a, y: b, .. } = foo; // on décortique les attributs de notre structure
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 :

```
let foo = A {
    x: "Hello".to_string(),
    y: " ".to_string(),
    z: "world!".to_string(),
```

```
};
{
    let A { x: a, y: b, z: c } = foo; // on décortique les attributs de notre structure
    println!("{}{}{}", a, b, c); // puis on les utilise dans de nouvelles variables
}

// a, b et c ne pourront plus être utilisés à partir d'ici
```

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)` .

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

Ci-dessous figure un exemple complet d'implémentation :

```
struct Spartan<'a> {
    sid: i32,
    name: &'a str,
}

impl<'a> PartialEq for Spartan<'a> {
    fn eq(&self, other: &Spartan) -> bool {
        self.sid == other.sid
    }
}

impl<'a> Spartan<'a> {
    pub fn new(sid: i32, name: &str) -> Spartan {
        Spartan {
            sid: sid,
            name: name,
        }
    }
}

fn main() {
    let (foo , bar) = (Spartan::new(117, "John"), Spartan::new(062, "Jorge"));
```

```
    if foo == bar {
        println!("foo equals bar");
    } else {
        println!("foo not equals bar");
    }
}
```

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.

```
#[macro_use]
extern crate votre_lib;

fn main() {
    votre_macro!();
}
```

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.

```
// dans le fichier lib.rs
#[macro_use] // bien préciser que ce module utilise des macros
pub mod votre_conteneur {
    #[macro_export]
    macro_rules! foo
    {
        () => ();
    }
    #[macro_export]
    macro_rules! bar
    {
        () => ();
    }
    #[macro_export]
    macro_rules! baz
    {
        () => ();
    }
}
```

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.

À quoi servent les mot-clés `if let` ?

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

```
fn main() {
    let foo : Option<String> = Some("Hello world!".to_string());
    let mut bar : bool = false;

    if let Some(content) = foo { // si la variable foo contient quelque chose...
        bar = true;
    } else {
        println!("foo's content is None");
    }
}
```

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

À quoi servent les mot-clés `while let` ?

La combinaison des deux mot-clés permet d'effectuer des tests concis et ainsi nous éviter de passer par le pattern matching lorsque cela 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]

```
let mut v = vec![1, 3, 5, 7, 11];

while let Some(x) = v.pop() {
    println!("{}", x);
}
```

Comment étendre un trait sur un autre trait ?

Mécaniques et philosophies

Gestion de la mémoire

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 » ?

Qu'est-ce que « l'ownership » ?

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'écarter 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 » :

```
fn main() {
    let foo: String = String::from("Hello world!");
    let bar: String = foo;
    let baz: String = foo; // erreur, la ressource a été « déplacée »
}
```

Renvoyant une erreur de ce style :

```
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 « déplacant » 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*.

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

```
fn main() {
    let foo: String = "Hello world!".to_owned();
    let bar: String = foo;
    let bar2: String = bar.clone(); // on clone bar
    let baz: &String = &bar; // on récupère une référence
}
```

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 !

Vous pouvez très bien écrire ceci :

```
fn main() {
    let foo = 42;
    let bar = foo;
    let baz = foo;
}
```

Car les types primitifs tels que les `i8`, `i16`, `i32`, `i64`, `u8`, … implémentent le trait `Copy`.

Il est cependant important de noter que les appels à `clone` sont très coûteux et ne devraient être utilisés qu'en derniers recours. En général (sauf si vous souhaitez vraiment copier les données), on peut toujours faire autrement.

Quid des fonctions ?

Les fonctions obéissent aux mêmes règles que les pointeurs :

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

Exemple :

```
fn my_func(my_string: String) {
    for letter in my_string.chars() {
        println!("{}", &letter);
    }
}

fn main() {
    let foo: String = String::from("The cake is a lie!");

    my_func(foo);
    let chars = foo.chars(); // error
}
```

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

Voir aussi : Qu'est-ce que le concept de « borrowing » ?

Qu'est-ce que le concept de « borrowing » ?

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 :

```
fn main()  {
    let mut foo = 117;
    let bar = &mut foo;
    let baz = &foo; // erreur
}
```

Ou :

```
fn main() {
    let mut foo = 117;
    let bar = &mut foo;
    let baz = &mut foo; //erreur
}
```

Qu'est-ce que le concept de « lifetime » ?

Introduction

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

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

Voici un exemple :

```
fn main() {
    let mut foo: String = "Hello world!".to_string(); // Le scope A commence ici
    let bar: String = "Goodbye, friend !".to_string(); // Le scope B commence ici
    foo = bar; // bar détruit, le scope B s'arrête là
    println!("{}", &bar); // erreur du compilateur
} // 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'expliciter 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) :

```
fn foo(phrase: &str) { //aucune référence ne survit, donc pas la peine de l'annoter
    println!("{}", &phrase);
}

fn bar<'a>(phrase: &'a mut String, word: &str) -> &'a String { // une référence va survivre
    phrase.push_str(word);
    phrase
} // La référence qui survivra sera donc « phrase », elle dispose donc de la durée de vie '

fn main() {
    let mut baz: String = "Hello ".to_string();
    let word: &str   = "world!";
    let bazz = bar(&mut baz, word); // ce que contient la varialbe bazz ne peut être accédé
```

```
    println!("{}", &bazz); // nous affichons nos caractères sur la sortie standard
}
```

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 créé 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

Outils de build

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*.

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 project_name  --bin
```

Le manifest sera modifié en conséquence.

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
```

Il est également possible de laisser le compilateur effectuer ses optimisations.

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

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 (donc `./doc/votre_projet/index.html`).

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

Où trouver de nouvelles bibliothèques ?

Vous pouvez trouver d'autres bibliothèques sur le site officiel de Cargo.

Voir aussi : Comment installer de nouvelles bibliothèques ?

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: `lib_name = "lib.version"`;
- 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 dépendances : `lib_name = {git = "url du dépôt" }` ;
- Compilez ;
- C'est prêt !

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.

Attention:

> 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.

```
$ 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 par un autre paquet.

Empaqueter le projet

La prochaine étape consiste à empaqueter 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 empaqueté sous le format *.crate* et se trouvera dans le répertoire target/package/.

```
$ 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]

```
[package]
# ...
exclude = [
    "public/assets/*",
    "videos/*",
]
```

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]

```
[package]
# ...
include = [
    "**/*.rs",
    "Cargo.toml",
]
```

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 !

```
$ 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 :

```
[package]
name = "verbose_bird"
version = "0.3.2"
authors = ["Songbird0 <chaacygg@gmail.com>"]
description = "An awesome homemade loggers pack."
documentation = "https://github.com/Songbird0/Verbose_Bird/blob/master/src/README.md"
homepage = "https://github.com/Songbird0/Verbose_Bird"
repository = "https://github.com/Songbird0/Verbose_Bird"

readme = "README.md"

keywords = ["Rust", "log", "loggers", "pack"]

license = "GPL-3.0"

license-file = "LICENSE.md"

[dependencies]
```

Attention:

> 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 visionnez les noms raccourcis entre parenthèses de chaque licence.

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 :

```
#[cfg(test)]
mod oo_tests {
    struct Alice;
    use loggers_pack::oop::Logger;
    impl Logger for Alice{/*...*/}

    #[test]
    fn pack_logger_oop_info() {
        Alice::info("@Alice", "Hello, I'm Alice ", "Peterson !");
    }

    #[test]
    fn pack_logger_oop_wan() {
        Alice::warn("@Alice", "Hello, I'm Alice ", "Peterson !");
    }

    #[test]
    fn pack_logger_oop_error() {
        Alice::error("@Alice", "Hello, I'm Alice ", "Peterson !");
    }

    #[test]
    fn pack_logger_oop_success() {
        Alice::success("@Alice", "Hello, I'm Alice ", "Peterson !");
    }
}
```

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

Attention:

> 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 **1.12.1**. Si vous rencontrez ce problème, nous vous conseillons de mettre à jour votre compilateur jusqu'à la version stable la plus récente possible.

Comment mettre à jour mes bibliothèques ?

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

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

Comment créer ses benchmarks avec Cargo ?

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

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

```
[package]
name = "awesome_tests"
version = "0.1.0"
authors = ["Songbird0 <chaacygg@gmail.com>"]

[dependencies]

bencher = "0.1.1"

[[bench]]
name = "my_bench"
harness = false
```

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

```
#[macro_use]
extern crate bencher;
use bencher::Bencher;

fn find_short(s: &str) -> usize {
    let splitting: Vec<&str> = s.split_whitespace().collect();
    let mut shortest_len: usize = 0;
    let mut i: usize = 0;

    while i < splitting.len() {
        if i == 0 {
            shortest_len = splitting[0].len();
        } else {
            if splitting[i].len() < shortest_len {
                shortest_len = splitting[i].len();
```

```
            }
        }
        i += 1;
    }
    shortest_len
}

fn bench_find_short(b: &mut Bencher) {
    b.iter(|| find_short("Hello darkness my old friend"));
}

benchmark_group!(my_bench, bench_find_short);
benchmark_main!(my_bench);
```

À quoi sert benchmark_group! ?

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

À quoi sert benchmark_main! ?

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

Gestion des erreurs

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 :

- 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 l'énumération Option ?
- Comment utiliser l'énumération Option ?
- Qu'est-ce que l'énumération Result ?
- Comment utiliser l'énumération Result ?

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 erreurs de cette manière.

Voir aussi :

Comment s'effectue la gestion des erreurs avec Rust ?

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! ?

À quoi sert la macro panic! ?

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

```
public class MyClass
{
        public static void main(String[] args)
        {
            throw new RuntimeException("Error !");
```

```
            System.out.println("Dead code.");
        }
}
```

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.

```
fn main() {
    panic!("Error !");
    println!("Dead code");
}
```

Voir aussi :

- À quoi sert la méthode unwrap() ?
- À quoi sert la méthode and_then ?
- À quoi sert la macro try! ?

À 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`().

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

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

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 l'énumération Option ?
- Qu'est-ce que l'énumération Result ?

À quoi sert la méthode unwrap_or ?

La méthode `unwrap_or()` fonctionne exactement comme la méthode originelle unwrap, 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.

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

    println!("{} {} {} {}", foo.unwrap(), bar.unwrap_or(String::from("ça passe, mais de just
    /*
        Pensez à isoler les appels de la méthode si vous ne souhaitez pas faire planter vot
    */
}
```

Voir aussi :

Tester l'exemple

À 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.

```
fn bang(arg: String) -> String {
    return "Chef, on a eu une erreur: ".to_string() + arg.as_str();
}

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

    bar.unwrap_or_else(|| { return "On évite la casse !".to_string(); });
    println!("{}", bing.unwrap_or_else(bang));
}
```

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

À quoi sert la méthode map ?

Elle permet de modifier la donnée contenue. Exemple :

```rust
fn main() {
    let foo = vec![1, 2, 3];
    let foo2: Vec<_> = foo.iter().map(|entry| format!("> {}", entry)).collect();

    for entry in foo2 {
        println!("-> \"{}\"", entry);
    }
}
```

Ce qui affichera :

```
-> "> 1"
-> "> 2"
-> "> 3"
```

À 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.

```rust
fn concat(arg: &str) -> Option<String> {
    Some(arg.to_string() + "world!")
}

fn main() {
    let foo = Some("Hello ");
    println!("{}", foo.and_then(concat).unwrap());
}
```

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

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

Voir aussi :

- À quoi sert la méthode unwrap() ?
- Qu'est-ce que l'énumération Result ?
- Qu'est-ce que l'énumération Option ?

À quoi sert la macro `try!` ?

La macro `try!` permet de s'assurer de l'intégrité de la ressource. Si la ressource *enveloppée* par la macro `try!` est intègre, elle sera *bindée* à l'identificateur qui lui est assigné. Sinon, `try!` effectue un retour, renvoi prématuré.

Note

Attention toutefois à ne pas oublier qu'une fonction usant de cette macro doit forcément renvoyer une instance de `Result<(), io::Error>` (le type de la valeur renvoyée en cas de succès est arbitaire).

```
fn foo(string: &String) -> Result<(), std::io::Error>
{
    try!(std::fs::File::create("my_file.txt"));
    println!("Une chance sur deux pour que je sois du code mort !");
    Ok(())
}

fn bar(string: &String) -> std::io::Result<()>
{ // fonctionne également avec l'alias de Result<T, E>
    try!(std::fs::File::create("my_file.txt"));
    println!("Une chance sur deux pour que je sois du code mort !");
    Ok(())
}
```

Note(bis)

Depuis la version `1.13`, la macro `try!` a été plus ou moins remplacée par l'opérateur `?`. Elle peut toujours être utilisée, toutefois, privilégiez cet opérateur autant que possible.

L'exemple ci-dessus peut donc être transposé de cette manière:

```
use std::io::Error;
fn foo(string: &String) -> Result<(), Error>
{
    std::fs::File::create("my_file.txt")?;
    println!("Une chance sur deux pour que je sois du code mort !");
    Ok(())
}

fn bar(string: &String) -> std::io::Result<()>
{ // fonctionne également avec l'alias de Result<T, E>
    std::fs::File::create("my_file.txt")?;
    println!("Une chance sur deux pour que je sois du code mort !");
    Ok(())
}
```

Voir aussi:

- Le document abordant la gestion d'erreur avec les nouvelles fonctionnalités du langage.

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 :

```
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 :

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

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.

```
fn foo(arg: Option<String>) {
    let bar: String = String::from("Hello world!");
    let mut some: Option<String> = None;
    assert!(!arg.is_none(), "Arg is None");
    assert!(arg.unwrap().eq(&bar), "arg n'est pas égal à bar");
}

fn main() {
    foo(Some("Ok".to_string()));
    foo(None);
}
```

Voir aussi :

- Visionner le résultat de l'exemple (requiert une connexion internet)
- Comment créer une macro ?

Comment utiliser la macro assert_eq! ?

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

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

```
fn foo(arg: Option<String>) {
    let bar: String = String::from("Hello world!");
    let mut some: Option<String> = None;
    assert!(!arg.is_none(), "Arg is None");
    assert_eq!(arg.unwrap(), bar, "arg n'est pas égal à bar");
}

fn main() {
    foo(Some("Ok".to_string()));
    foo(None);
}
```

Voir aussi :

- Visionner le résultat de l'exemple (requiert une connexion internet)
- Comment créer une macro ?

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 débug. (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.

Note: 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 créer une macro ?

Qu'est-ce que l'énumération Result ?

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

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 l'énumération `Result<T, E>` ?

Comment utiliser l'énumération Result ?

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 :

```
fn foo<'a, 'b>(arg: Option<&'a str>) -> Result<String, &'b str> {
    if let Some(content) = arg {
        let unwrapping = arg.unwrap();
        Ok(unwrapping.to_string())
    } else {
        Err("L'argument ne contient rien.")
    }
}

fn main() {
    match foo(None) {
        Ok(content) => println!("Ok: {}", content),
        Err(err) => println!("Error: {}", err.to_string()),
    }
}
```

Voir aussi :

À quoi sert la macro `try!` ?

À quoi sert la macro `panic!` ?

Le résultat de cet exemple

Qu'est-ce que l'énumération Option ?

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.

Comment utiliser l'énumération Option ?

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.

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

Meta-données

I/O

Que puis-je trouver dans cette section ?

Dans cette section, vous retrouverez toutes les questions couramment posées concernant l'utilisation des outils dédiés à la gestion des flux.

Comment créer un fichier ?

Pour créer un fichier, rien de plus simple, il vous faudra utiliser la structure `File`.

```
use std::io;
use std::fs::File;

fn foo() -> io::Result<()> //vous pouvez mettre ce que vous voulez dans le diamant pour que
{
    let mut file = File::create("thecakeisalie.txt")?;
    Ok(())
}
```

Comment lire le contenu d'un fichier ?

Pour lire un fichier, il vous faudra utiliser la structure `File` et le trait `Read`. La procédure est presque identique à celle qui vous permet de créer votre fichier.

```
use std::io;
use std::fs::File;

fn foo() -> io::Result<()> //vous pouvez mettre ce que vous voulez dans le diamant pour que
{
    let mut file = File::open("thecakeisalie.txt")?;
    let mut string = String::new();
    file.read_to_string(&mut string)?;
    Ok(())
}
```

Comment écrire à l'intérieur d'un fichier ?

Pour écrire dans en fichier, vous devrez importer trois ressources:

- std::io::Result; (un alias de l'`enum std::result::Result`)
- std::io::Write; (Le trait qui permet d'implémenter la méthode `write_all()`)
- std::fs::File. (La structure censée représenter votre fichier)

```
use std::io::Result;
use std::io::Write;
use std::fs::File;

fn foo() -> Result<()>
{

    let mut f = File::create("foo.txt")?;
    f.write_all(b"Hello, world!")?;
    Ok(())
}

fn main() -> ()
{
    /*...*/
}
```

A quoi sert le 'b' qui préfixe la chaîne de caractères ?

La méthode `write_all()` ne traite l'information que sous forme d'octets, et pour convertir une chaîne caractères en octets, elle doit être précédée par la lettre 'b'.

Comment différencier un fichier d'un répertoire ?

Pour cela, vous aurez besoin d'utiliser une ressource:

- std::path::Path;

Vous pouvez récupérer une instance de la structure `Path` en lui soumettant un chemin hypothètique, pour enfin effectuer vos tests.

```
use std::path::Path;

fn main() -> ()
{
   let my_path : &Path = Path::new("/your/directory/"); /* ne fonctionnera bien évidemment p
   if my_path.is_dir()
   {

       println!("He's a directory !");

   }
   else{ println!("He isn't a directory !"); }
}
```

Comment lister les objets d'un répertoire ?

Antisèches Rust

Trucs & astuces

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. :)

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

```
fn main() {
    let foo = String::from("Hello");
    let bar = foo.chars();

    for letter in bar {}
    for letter in bar {}
}
```

Erreur :

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

La solution pourrait être la suivante :

```
    let foo = String::from("Hello");
    let bar = foo.chars();

    for letter in &bar {}
    for letter in &bar {}
```

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

Mais vous récolterez 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`.

Que faire alors ?

La méthode remplaçant `.iter()` est `.collect()`; Cette dernière vous permet de récupérer un vecteur contenant toutes les entités de l'ancien itérateur. Vous pouvez désormais accéder à votre ressource par référence et ainsi la parcourir autant de fois que vous le souhaitez.

```
fn main()
{
    let foo = String::from("Hello");
    let bar = foo.chars();
    let baz : Vec<char> = bar.collect();
    for letter in &baz {}
    for letter in &baz {}
}
```