Tutoriel Rust

Une version papier de ce tutoriel est disponible chez dunod.

Rust est un langage initié par Mozilla, désormais soutenu par la Rust Foundation et poussé par ses nombreux contributeurs sur GitHub. Ce tutoriel s'adresse à des développeurs ayant déjà programmé dans un autre langage. Ici, vous apprendrez toutes les bases afin que vous puissiez vous débrouiller tout seul par la suite.

Sommaire

I. Les bases de la programmation en Rust ..... p. 3
1. Présentation de Rust ..... p. 3
2. Mise en place des outils ..... p. 4
3. Premier programme ..... p. 6
4. Variables ..... p. 7
5. Conditions et pattern matching ..... p. 10
6. Les fonctions ..... p. 14
7. Les expressions ..... p. 16
8. Les boucles ..... p. 18
9. Les enums ..... p. 22
10. Les structures ..... p. 24
11. if let / while let ..... p. 29
12. Gestion des erreurs ..... p. 31
13. Cargo ..... p. 35
14. Utiliser des bibliothèques externes ..... p. 39
15. Jeu du plus ou moins ..... p. 40
II. Spécificités de Rust ..... p. 44
1. Le formatage des flux ..... p. 44
2. Les traits ..... p. 46
3. Les attributs ..... p. 51
4. Généricité ..... p. 54
5. Propriété (ou ownership) ..... p. 59
6. Durée de vie (ou lifetime) ..... p. 62
7. Déréférencement ..... p. 66
8. Sized et String vs str ..... p. 68
9. Unsafe ..... p. 70
10. Les unions ..... p. 73
11. Closure ..... p. 76
12. Multi-fichier ..... p. 79
13. Les macros ..... p. 82
14. Box ..... p. 87
15. Les itérateurs ..... p. 89
III. Aller plus loin ..... p. 92
1. Les macros procédurales (ou proc-macros) ..... p. 92
2. La compilation conditionnelle ..... p. 99
3. Utiliser du code compilé en C ..... p. 102
4. Documentation et rustdoc ..... p. 106
5. Ajouter des tests ..... p. 110
6. Rc et RefCell ..... p. 114
7. Le multi-threading ..... p. 118
8. Le réseau ..... p. 124

IV. Annexes ..... p. 130
1. Code des annexes ..... p. 130
2. Comparaison avec C++ ..... p. 132

I. Les bases de la programmation en Rust

1. Présentation de Rust

Rust est un langage de programmation système, compilé et multi-paradigme. C'est un croisement entre langage impératif (C), objet (C++), fonctionnel (Ocaml) et concurrent (Erlang). Il s'inspire des recherches en théories des langages de ces dernières années et des langages de programmation les plus populaires afin d'atteindre trois objectifs : rapidité, sécurité (en mémoire notamment) et concurrent (partage des données sécurisé entre tâches). Il est notamment utilisé pour de la programmation système, écrire des serveurs webs, faire des applications en ligne de commandes, des applications graphiques et des jeux vidéos.

Le développement de Rust a été initié par Graydon Hoare en 2006, notamment dans le but de résoudre les failles de sécurité dans Firefox sans que cela n'impacte négativement les performances. Sa première version stable, la 1.0, est sortie le 15 Mai 2015. En Août 2020, Mozilla a arrêté de soutenir le développement du langage, conduisant à la création de la fondation Rust le 8 Février 2021. Le but de cette fondation n'étant pas de diriger le développement du langage mais de le soutenir financièrement.

Depuis sa première version stable, Rust a été adopté par toutes les plus grosses entreprises de l'informatique telle que Google qui s'en sert pour Android ainsi que son cloud, Microsoft qui s'en sert dans Windows, Amazon, Facebook, Discord, Huawei, Dropbox, Mozilla...

Du côté des projets opensource, c'est devenu le troisième langage de programmation utilisé dans le développement du kernel Linux après le C et l'assembleur en 2022. Le projet GNOME a de plus en plus de projets internes utilisant Rust et a déjà réécrit certaines de ses bibliothèques telles que librsvg.

Pour suivre ce tutoriel, il est recommandé d'avoir déjà développé dans au moins un autre langage (C, C++, Java, JavaScript, Python, etc.). Ce n'est pas parce que Rust est un langage particulièrement difficile à apprendre, mais plutôt parce que les concepts abordés dans ce livre supposent une certaine familiarité avec la programmation. En d'autres termes, si vous êtes totalement novice en programmation, vous pouvez trouver certains passages de ce livre difficiles à suivre. Cela vous permettra notamment de vous concentrer sur les aspects spécifiques de Rust plutôt que de devoir assimiler à la fois les notions de base de la programmation et de Rust.

Les points forts de Rust sont :

- La gestion de "propriété" (ownership) des variables
- La gestion de la mémoire
- Le typage statique
- L'inférence de type
- Le filtrage par motif (pattern matching)
- La généricité

Nous reverrons tout cela plus en détails. Quelques liens utiles :

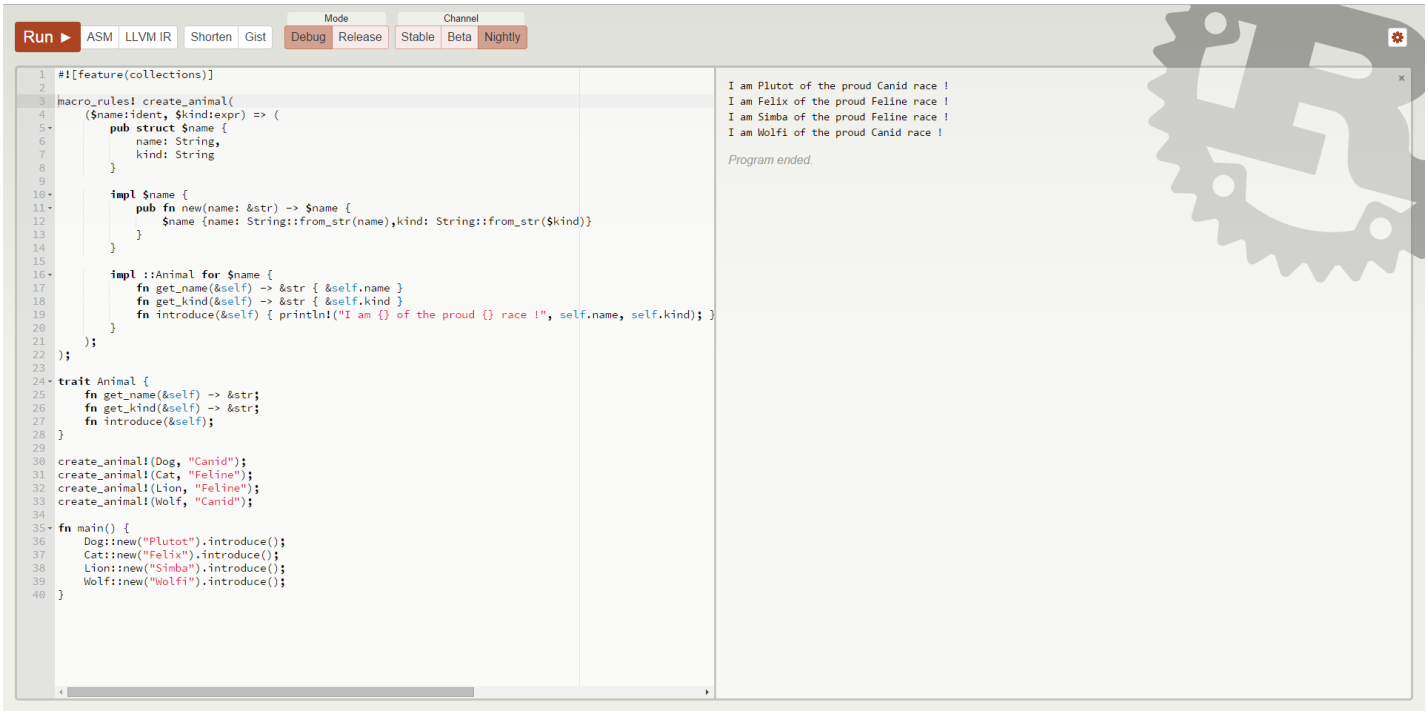
- Le site internet : rust-lang.org
- La documentation (toujours utile d'avoir ça sous la main !)
- Le dépôt Github (pour voir le code source)
- Le rustbook (le "cours" officiel, en anglais)
- Les rustlings (un programme d'exercices interactifs pour accompagner son apprentissage)
- rust by example (Une compilation d'exemples de rust)
- Le reddit (pour poser une question)

Il est maintenant temps de commencer.

2. Mise en place des outils

Pour pouvoir développer en Rust, il va déjà falloir les bons outils. Ici, je ne ferai qu'une présentation rapide de ceux que je **connais**. Pour écrire le code, vous pouvez utiliser soit :

- L'éditeur de code Rust en ligne :



- Soit un IDE. Par exemple visual studio.

J'utilise personnellement Sublime Text. Si vous souhaitez l'utiliser et que vous voulez avoir la coloration syntaxique pour Rust, je vous invite à vous rendre sur cette page. Au final ça donne ceci :

```rust
#![feature(collections)]

macro_rules! create_animal(
    ($name:ident, $kind:expr) => (
        pub struct $name {
            name: String,
            kind: String
        }

        impl $name {
            pub fn new(name: &str) -> $name {
                $name {name: String::from_str(name),kind: String::from_str($kind)}
            }
        }

        impl ::Animal for $name {
            fn get_name(&self) -> &str { &self.name }
            fn get_kind(&self) -> &str { &self.kind }
            fn introduce(&self) { println!("I am {} of the proud {} race !", self.name, self.kind); }
        }
    );
);

trait Animal {
    fn get_name(&self) -> &str;
    fn get_kind(&self) -> &str;
    fn introduce(&self);
}

create_animal!(Dog, "Canid");
create_animal!(Cat, "Feline");
create_animal!(Lion, "Feline");
create_animal!(Wolf, "Canid");

fn main() {
    Dog::new("Plutot").introduce();
    Cat::new("Felix").introduce();
    Lion::new("Simba").introduce();
    Wolf::new("Wolfi").introduce();
}
```

Après il vous suffit de suivre les instructions et vous aurez un éditeur de texte prêt à l'emploi ! Je tiens cependant à préciser que n'importe quel éditeur de texte fera l'affaire, Sublime Text n'est que ma préférence personnelle !

Les outils de Rust

L'installeur officiel pour le langage est disponible sur rustup.rs. Exécuter la commande et vous aurez le compilateur (rustc), le gestionnaire de paquet (cargo), la documentation du langage ainsi que rustup qui vous permettra de mettre tout cela à jour facilement.

Parmi les outils fournis, il y a notamment un formateur de code que vous pouvez lancer avec la commande `cargo fmt` et un linter de code que vous pouvez lancer avec `cargo clippy`.

Nous pouvons maintenant nous intéresser au langage Rust à proprement parler rédiger votre premier programme !

3. Premier programme

Après avoir installé les éléments nécessaires à l'emploi de rust (cf. chapitre 1), écrivons notre premier programme en Rust dans un fichier nommé **votre_fichier.rs**.

```
Run the following code:
fn main() {
    println!("Hello world!");
}
```

Maintenant que nous avons créé le fichier, compilons-le :

```
rustc votre_fichier.rs
```

Vous devriez maintenant avoir un exécutable **votre_fichier**. Lançons-le.

Sous Windows :

```
.\votre_fichier.exe
```

Sous Linux/macOS :

```
./votre_fichier
```

Et vous devriez obtenir :

```
Hello world!
```

Si jamais vous voulez changer le nom de l'exécutable généré, il vous faudra utiliser l'option **-o**. Exemple :

```
rustc votre_fichier.rs -o le_nom_de_l_executable
```

Vous savez maintenant comment compiler et exécuter vos programmes.

4. Variables

La première chose à savoir en Rust est que toutes les variables sont constantes par défaut. Exemple :

```
Run the following code:
let i = 0;

i = 2; // Erreur !
```

Pour déclarer une variable mutable, il faut utiliser le mot-clé **mut** :

```
Run the following code:
let mut i = 0;

i = 2; // Ok !
```

Les types

Voyons maintenant comment fonctionnent les **types** en Rust. Ici, rien de nouveau par rapport à ce que vous avez pu voir dans d'autres langages, on a toujours des entiers, des flottants, des strings, etc. La seule différence viendra surtout de leur nommage. Par exemple, pour déclarer un entier de 32 bits, vous écrirez :

```
Run the following code:
let i: i32 = 0;
// ou :
let i = 0i32;
```

Vous devez également savoir que le compilateur de Rust utilise **l'inférence de type**. Cela signifie qu'il peut déduire le type d'une variable en fonction de sa valeur. Nous ne sommes donc pas obligés de déclarer le type d'une variable. Exemple :

```
Run the following code:
// On se met dans la peau de rustc :
// 0 est un entier donc i est un entier
let i = 0;
// 10 est un i32 alors max est un i32
let max = 10i32;

// < est capable de comparer deux nombres alors comme on sait que :
// max est un i32, donc le compilateur en déduit que i en est un aussi
if i < max {
    println!("i est inférieur à max !");
}
```

Voici une liste des différents types de base (aussi appelés "primitifs") disponibles :

- i8 : un entier signé de 8 bits
- i16 : un entier signé de 16 bits
- i32 : un entier signé de 32 bits
- i64 : un entier signé de 64 bits
- i128 : un entier signé de 128 bits
- u8 : un entier non signé de 8 bits
- u16 : un entier non signé de 16 bits
- u32 : un entier non signé de 32 bits
- u64 : un entier non signé de 64 bits
- u128 : un entier non signé de 128 bits
- f32 : un nombre flottant de 32 bits
- f64 : un nombre flottant de 64 bits
- str (on va y revenir plus loin dans ce chapitre)
- slice (on va y revenir plus loin dans ce chapitre)

Sachez cependant que les types **isize** et **usize** existent aussi et sont l'équivalent de **intptr_t** et de **uintptr_t** en C/C++. En gros, sur un système 32 bits, ils feront respectivement 32 bits tandis qu'ils feront 64 bits sur un système 64 bits.

Dernier petit point à aborder : il est courant de croiser ce genre de code en C/C++/Java/etc... :

Run the following code:
```
i++;
++i;
```

Cette syntaxe est invalide en Rust, il vous faudra donc utiliser :

Run the following code:
```
i += 1;
```

Autre détail qui peut avoir son importance : si on fait commencer le nom d'une variable par un '_', nous n'aurons pas de warning du compilateur si elle est inutilisée. Ça a son utilité dans certains cas, bien que cela reste assez restreint. Exemple :

Run the following code:
```
let _i = 0;
```

Les tableaux

On peut déclarer un tableau de cette façon :

Run the following code:
```
let tab = [0, 1, 2];
```

On ne peut pas modifier la taille d'un tableau, on peut seulement modifier son contenu. Si vous souhaitez avoir un tableau dont la taille peut être modifiée, il faudra utiliser le type Vec (prononcer "vecteur") :

Run the following code:
```
// Un vecteur vide.
let mut v = Vec::new();

// On ajoute les valeurs 2, 1 et 0.
v.push(2);
v.push(1);
v.push(0);

// Ça affichera "[2, 1, 0]".
println!("{:?}", v);
```

Il est maintenant temps de revenir sur les **slices**.

Les slices

Pour faire simple, une slice représente un morceau de mémoire ainsi que le nombre d'éléments qu'elle contient. Contrairement aux tableaux, leur taille n'a pas besoin d'être connue au moment de la compilation, ce qui les rend beaucoup plus facile à manipuler. Cela deviendra plus évident quand on abordera les fonctions. Une slice est créée quand on utilise & devant un tableau. Exemples :

Run the following code:
```
// tab est une slice contenant 0, 1 et 2.
let tab = &[0, 1, 2];
// Ça affichera "[0, 1, 2]".
println!("{:?}", tab);

// On crée une slice commençant à partir du 2e élément de tab.
let s = &tab[1..];
// Ça affichera "[1, 2]".
println!("{:?}", s);
```

De la même façon qu'il est possible d'obtenir une slice à partir d'un tableau ou d'une slice, on peut en obtenir à partir des

Vecs :

```
Run the following code:
let mut v = Vec::new();

v.push(0);
v.push(1);
v.push(2);

// Ça affichera "[0, 1, 2]".
println!("{:?}", v);
let s = &v[1..];

// Ça affichera "[1, 2]".
println!("{:?}", s);
```

Les types contenant des tableaux ont toujours une slice associée. Par exemple, String a &str, OsString a &OsStr, etc...

5. Conditions et pattern matching

Nous allons d'abord commencer par les conditions :

if / else if / else

Les if / else if / else fonctionnent de la même façon qu'en C/C++/Java :

Run the following code:
```
let age: i32 = 17;

if age >= 18 {
    println!("majeur !");
} else {
    println!("mineur !");
}
```

Notez que je n'ai pas mis de parenthèses, `(` et `)`, autour des conditions : elles sont superflues en Rust. Cependant, elles seront nécessaires si vous faites des "sous"-conditions :

Run the following code:
```
if age > 18 && (age == 20 || age == 24) {
    println!("ok");
}
```

Par contre, les accolades `{` et `}` sont **obligatoires**, même si le bloc de votre condition ne contient qu'une seule ligne de code !

Pour rappel : `&&` est la condition `et` tandis que `||` est la condition `ou`. Donc dans le code au-dessus, il faut le lire comme ceci :

`age` doit être supérieur à 18 ET `age` doit être égal à 20 OU 24.

Comparaison de booléens

Si vous souhaitez faire une comparaison avec un booléen (donc `true` ou `false`), sachez qu'il est possible de les écrire de façon plus courte :

Run the following code:
```
if x == true {}
// est équivalent à:
if x {}

if x == false {}
// est équivalent à:
if !x {}
```

Pattern matching

Rappelons dans un premier temps la définition du pattern matching ou "filtrage par motif" : c'est la vérification de la présence de constituants d'un motif par un programme informatique ou parfois par un matériel spécialisé.

Pour dire les choses plus simplement, c'est une condition permettant de faire les choses de manière différente. Grâce à lui, on peut comparer ce que l'on appelle des **expressions** de manière plus intuitive (nous aborderons les expressions un peu plus loin dans ce cours). Si vous avez déjà utilisé des langages fonctionnels, vous ne vous sentirez pas dépaysés.

Regardons un exemple pour faciliter les explications :

```
Run the following code:
let my_string = "hello";

match my_string {
    "bonjour" => println!("français"),
    "ciao" => println!("italien"),
    "hello" => println!("anglais"),
    "hola" => println!("espagnol"),
    _ => println!("je ne connais pas cette langue..."),
}
```

Ici ça affichera "anglais".

On peut s'en servir sur n'importe quel type. Après tout, il sert à comparer avec des **patterns** (**motifs** en français). Vous pouvez très bien matcher sur un **i32** ou sur un **f64** si vous voulez.

Concernant le _ utilisé dans la dernière branche du **match**, il s'agit d'une variable (nommée ainsi pour éviter un warning pour variable non utilisée, _a aurait eu le même résultat) qui contient "tous les autres patterns". C'est en quelque sorte le **else** du pattern matching. Il est obligatoire de l'utiliser si tous les patterns possibles n'ont pas été testés ! Dans le cas présent, il est impossible de tester toutes les strings existantes, on utilise donc _ à la fin. Si on teste un booléen, il n'y aura que deux valeurs possibles et il sera possible de les tester toutes les deux :

```
Run the following code:
let b = true;

match b {
    true => println!("true"),
    false => println!("false"),
}
```

La syntaxe du pattern matching suit ce modèle :

```
pattern => expression,
```

Le => sert à séparer le pattern de l'expression. La virgule sert à marquer la fin de l'expression.

Si vous souhaitez faire plus d'une chose dans une branche du pattern matching, on peut ajouter des accolades :

```
Run the following code:
let b = true;
let mut x = 0;

match b {
    true => {
        println!("true");
        x += 1;
    }
    false => println!("false"),
}
```

Si vous utilisez des accolades, la virgule devient optionnelle (c'est pourquoi elle n'est en pratique jamais présente dans ce cas).

Un autre exemple en utilisant un **i32** :

Run the following code:
```
let age: i32 = 18;

match age {
    17 => {
        println!("mineur !");
    }
    18 => {
        println!("majeur !");
    }
    x => {
        println!("ni 17, ni 18 mais {} !", x);
    }
}
```

Il est bien évidemment possible de matcher plus d'une valeur à la fois, pas besoin d'écrire toutes les valeurs en dessous de 18 pour voir s'il est mineur. Si on reprend le précédent exemple, le mieux serait d'écrire :

Run the following code:
```
let age: i32 = 17;

match age {
    tmp if tmp > 60 => {
        println!("plus tout jeune...");
    }
    tmp if tmp > 17 => {
        println!("majeur !");
    }
    _ => {
        println!("mineur !");
    }
}
```

Comme vous l'avez sans doute compris, il est possible d'ajouter une condition à un pattern matching. Elle nous permet d'affiner les résultats ! On peut donc ajouter des && ou des || selon les besoins.

Toujours plus loin !

Il est aussi possible de matcher directement sur un ensemble de valeurs de cette façon :

Run the following code:
```
let i = 0i32;

match i {
    10..=100 => println!("La variable est entre 10 et 100 (inclus)"),
    x => println!("{} n'est pas entre 10 et 100 (inclus)", x),
}
```

Vous pouvez aussi "binder" (ou matcher sur un ensemble de valeurs) la variable avec le symbole "@" :

Run the following code:
```
let i = 0i32;

match i {
    x @ 10..=100 => println!("{} est entre 10 et 100 (inclus)", x),
    x => println!("{} n'est pas entre 10 et 100 (inclus)", x),
}
```

Cela permet de stocker la valeur sur laquelle on matche directement dans x, ce qui est pratique si vous avez besoin de cette valeur (pour l'afficher comme dans le code au-dessus par exemple).

Il ne nous reste maintenant plus qu'un dernier point à aborder :

Run the following code:

```
match une_variable {
    "jambon" | "poisson" | "œuf" => println!("Des protéines !"),
    "bonbon" => println!("Des bonbons !"),
    "salade" | "épinards" | "fenouil" => println!("Beurk ! Des légumes !"),
    _ => println!("ça, je sais pas ce que c'est..."),
}
```

Vous l'aurez sans doute deviné : ici, le '|' sert de condition "ou" sur le pattern. Dans le premier cas, si `une_variable` vaut "jambon", "poisson" ou "œuf", le match rentrera dans cette condition, et ainsi de suite.

Voilà qui clôt ce chapitre sur les conditions et le pattern matching. N'hésitez pas à revenir sur les points que vous n'êtes pas sûr d'avoir parfaitement compris, car il s'agit vraiment de la base de ce langage. Si quelque chose n'est pas parfaitement maîtrisé, vous risquez d'avoir du mal à comprendre la suite.

6. Les fonctions

Jusqu'à présent, nous n'utilisions qu'une seule fonction : **main**. Pour le moment c'est amplement suffisant, mais quand vous voudrez faire des programmes plus gros, ça deviendra vite ingérable. Je vais donc vous montrer comment créer des fonctions en Rust.

Commençons avec un exemple :

Run the following code:
```
fn addition(nb1: i32, nb2: i32) -> i32
```

Ceci est donc une fonction appelée **addition** qui prend 2 arguments de type `i32` en paramètre et retourne un `i32`. Rien de très différent de ce que vous connaissez déjà donc.

Si vous souhaitez déclarer une fonction qui ne retourne rien (parce qu'elle ne fait qu'afficher du texte par exemple), vous pouvez la déclarer des façons suivantes :

Run the following code:
```
fn fait_quelque_chose() {
    println!("Je fais quelque chose !");
}
// ou bien :
fn fait_quelque_chose() -> () {
    println!("Je fais quelque chose !");
}
```

Expliquons rapidement ce qu'est ce **()** : c'est un **tuple** vide. **tuple** est une structure de donnée qui a un nombre d'éléments définis et dont les types peuvent différer. Ils sont utiles si l'on souhaite retourner plusieurs valeurs d'un coup :

Run the following code:
```
fn entier_et_float() -> (usize, f32) {
    (12, 0.1)
}

fn main() {
    let tuple = entier_et_float();
    // On accède aux champs avec leur position dans le tuple.
    println!("entier : {}, float : {}", tuple.0, tuple.1);
}
```

Pour en revenir au **tuple** vide, son équivalent le plus proche en C/C++ est le type **void** (et non pas la valeur **NULL**). Prenons un exemple :

Run the following code:
```
fn main() {
    println!("1 + 2 = {}", addition(1, 2));
}

fn addition(nb1: i32, nb2: i32) -> i32 {
    return nb1 + nb2;
}
```

Ce qui affiche :

```
1 + 2 = 3
```

Le mot-clé **return** retourne l'expression qui le suit. Donc ici, le résultat de l'addition `nb1 + nb2`. À noter qu'il est aussi possible de se passer de **return** si c'est la dernière **expression** de la fonction. Par exemple on pourrait réécrire la fonction `addition` de cette façon :

Run the following code:
```
fn addition(nb1: i32, nb2: i32) ->  i32 {
    nb1 + nb2
}
```

Par défaut, tout est expression en Rust, le point-virgule permettant simplement de marquer la fin de l'expression courante.

Ne vous inquiétez pas si vous ne comprenez pas tout parfaitement, nous verrons les expressions dans le chapitre suivant. Un autre exemple pour illustrer cette différence :

Run the following code:
```
fn get_bigger(nb1: i32, nb2: i32) ->  i32 {
    if nb1 > nb2 {
        return nb1;
    }
    nb2
}
```

Cette façon de faire n'est cependant pas recommandée en Rust, il aurait mieux valu écrire :

Run the following code:
```
fn get_bigger(nb1: i32, nb2: i32) ->  i32 {
    if nb1 > nb2 {
        nb1
    } else {
        nb2
    }
}
```

Une autre différence que certains d'entre vous auront peut-être noté (surtout ceux ayant déjà codé en C/C++) : je n'ai pas "déclaré" ma fonction addition et pourtant la fonction main l'a trouvé sans problème. Sachez juste que les déclarations de fonctions ne sont pas nécessaires en Rust (contrairement au C ou au C++ qui ont besoin de fichiers "header" par exemple).

Voilà pour les **fonctions**, rien de bien nouveau par rapport aux autres langages que vous pourriez déjà connaître.

Il reste cependant un dernier point à éclaircir : **println!** et tous les appels ayant un '!' ne sont pas des fonctions, ce sont des **macros**.

Si vous pensez qu'elles ont quelque chose à voir avec celles que l'on peut trouver en C ou en C++, détrompez-vous ! Elles sont l'une des plus grandes forces de Rust, elles sont aussi très complètes et permettent d'étendre les possibilités du langage. Par contre, elles sont très complexes et seront le sujet d'un autre chapitre.

Pour le moment, sachez juste que :

Run the following code:
```
fonction!(); // c'est une macro
fonction(); // c'est une fonction
```

7. Les expressions

Il faut bien comprendre que Rust est un langage basé sur les expressions. Avant de bien pouvoir vous les expliquer, il faut savoir qu'il y a les expressions et les déclarations. Leur différence fondamentale est que la première retourne une valeur alors que la seconde non. C'est pourquoi il est possible de faire ceci :

Run the following code:
```
let var = if true {
    1u32
} else {
    2u32
};
```

Mais pas ça :

Run the following code:
```
let var = (let var2 = 1u32);
```

C'est tout simplement parce que le mot-clé **let** introduit une assignation et ne peut donc être considéré comme une expression. C'est donc une déclaration. Ainsi, il est possible de faire :

Run the following code:
```
let mut var = 0i32;
let var2 = (var = 1i32);
```

Car **(var = 1i32)** est considéré comme une expression.

Attention cependant, une assignation de valeur retourne le type **()** (qui est un tuple vide. Pour rappel, son équivalent le plus proche en C/C++ est le type **void** comme expliqué dans le chapitre précédent) et non la valeur assignée contrairement à un langage comme le C par exemple.

Un autre point important d'une expression est qu'un point-virgule marquera toujours sa fin. Démonstration :

Run the following code:
```
let var: u32 = if true {
    1u32;
} else {
    2u32;
};
```

Il vous dira à ce moment-là que le if else renvoie '()' et donc qu'il ne peut pas compiler car il attendait un entier car j'ai explicitement demandé au compilateur de créer une variable **var** de type `u32`.

Je pense à présent que vous avez un bon aperçu de ce que sont les expressions. Il est très important que vous compreniez bien ce concept pour pouvoir aborder la suite de ce cours sereinement. Voici un exemple d'une fonction qui n'est composée que d'une expression :

Run the following code:
```
fn test_expression(x: i32) -> i32 {
    if x < 0 {
        println!("{} < 0", x);
        -1
    } else if x == 0 {
        println!("{} == 0", x);
        0
    } else {
        println!("{} > 0", x);
        1
    }
}
```

Tout comme pour les if/else, il est possible de retourner une valeur d'un pattern matching et donc de la mettre directement

dans une variable. Du coup, le ';' est nécessaire pour terminer ce bloc :

Run the following code:
```
let my_string = "hello";

let s = match my_string {
    "bonjour" => "français",
    "ciao" => "italien",
    "hello" => "anglais",
    "hola" => "espagnol",
    _ => "je ne connais pas cette langue..."
}; // on met un ';' ici car ce match retourne un type

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

Essayez le code suivant si vous avez encore un peu de mal à comprendre :

Run the following code:
```
fn main() {
    if 1 == 2 {
        "2"
    } else {
        "1"
    }
    println!("fini");
}
```

8. Les boucles

Les boucles sont l'une des bases de la programmation, il est donc impératif de regarder comment elles fonctionnent en Rust.

while

Comme dans les autres langages, la boucle **while** continue tant que sa condition est respectée. Exemple :

```
Run the following code:
let mut i: i32 = 0;

while i < 10 {
    println!("bonjour !");
    i += 1;
}
```

Ici, le programme affichera bonjour tant que `i` sera inférieur à 10.

Il faut cependant faire attention à ces deux éléments :

- Notez bien qu'il n'y a pas de parenthèse autour de la condition (`i < 10`).
- Les accolades sont **obligatoires** !

loop

Il existe aussi la possibilité d'écrire des boucles infinies avec le mot clé **loop** (plutôt qu'un `while true`) :

Il est assez courant d'écrire des boucles infinies mais prenons un cas pratique de leur utilisation : un jeu vidéo. L'affichage doit alors continuer en permanence jusqu'à ce que l'on quitte. Donc plutôt que d'écrire :

```
Run the following code:
while true {
    //...
}

// ou

let mut end = false;

while !end {
    //...
}
```

On écrira :

```
Run the following code:
loop {
    //...
}
```

Pour sortir d'une boucle infinie, il y a deux solutions :

- Utiliser le mot-clé **break**.
- Utiliser le mot-clé **return**.

Reprenons notre exemple du début et modifions-le un peu pour utiliser **loop** à la place :

Run the following code:
```
let mut i: i32 = 0;

loop {
    println!("bonjour !");
    i += 1;
    if i > 10 {
        break; // On arrête la boucle.
    }
}
```

Petit rappel concernant les mots-clés **break** et **return** : le mot-clé **break** permet seulement de quitter la **boucle** courante :

Run the following code:
```
loop { // Première boucle
    println!("Toujours là !");
    let mut i = 0i32;

    loop { // Deuxième boucle.
        println!("sous-boucle !");
        i += 1;
        if i > 2 {
            // On reprend l'exécution de "Première boucle".
            break;
        }
    }
}
```

Tandis que le mot-clé **return** fait quitter la **fonction** courante :

Run the following code:
```
fn boucle_et_print() {
    loop {
        println!("Toujours là !");
        let mut i = 0i32;

        loop {
            println!("sous-boucle !");
            i += 1;
            if i > 2 {
                // On quitte la fonction "boucle_et_print".
                return;
            }
        }
    }
}
```

for

La boucle **for** est un peu plus complexe que les deux précédentes. Elle ne fonctionne qu'avec des objets implémentant le trait **IntoIterator**. À ce stade nous n'avons pas encore vu ce qu'est un trait, mais nous y reviendrons plus tard. Toutefois, la compréhension exacte du fonctionnement des traits n'est pas indispensable pour comprendre le fonctionnement de **for**. Regardons dès à présent quelques exemples :

Run the following code:
```
for i in 0..10 {
    println!("i vaut : {}", i);
}
```

Ce qui va afficher :

```
i vaut : 0
i vaut : 1
i vaut : 2
i vaut : 3
i vaut : 4
i vaut : 5
i vaut : 6
i vaut : 7
i vaut : 8
i vaut : 9
```

La variable **i**, créée pour la boucle **for**, prendra successivement toutes les valeurs allant de 0 à 9, puis la boucle prendra fin.

Maintenant revenons sur ce `0..10` : c'est un objet de type **Range** qui implémente le trait **IntoIterator**, nous permettant d'itérer dessus.

Prenons un deuxième exemple avec un **Vec** cette fois :

```
Run the following code:
let v = vec![1, 4, 5, 10, 6]; // On crée un vecteur qui contient ces valeurs.

for value in v { // Puis on itère sur les valeurs de ce vecteur.
    println!("{}", value);
}
```

Ce qui va afficher :

```
1
4
5
10
6
```

Donc comme indiqué, si votre type implémente le trait **IntoIterator**, vous pouvez utiliser la boucle **for** pour itérer dessus.

Énumération

Si vous souhaitez savoir combien de fois vous avez itéré, vous pouvez utiliser la fonction enumerate :

```
Run the following code:
for (position, valeur) in (6..10).enumerate() {
    println!("position = {} et valeur = {}", position, valeur);
}
```

Ce qui affichera :

```
position = 0 et valeur = 6
position = 1 et valeur = 7
position = 2 et valeur = 8
position = 3 et valeur = 9
```

position vaut donc le nombre d'itérations effectuées à l'intérieur de la boucle tandis que **valeur** prend successivement les valeurs du range `6..10`. Autre exemple :

```
Run the following code:
let v = vec!["a", "b", "c", "d"]; // On crée un vecteur.

for (position, value) in v.iter().enumerate() { // On itère sur ses valeurs.
    println!("position = {} et value = \"{}\"", position, value);
}
```

Ce qui affichera :

```
position = 0 et value = "a"
position = 1 et value = "b"
position = 2 et value = "c"
position = 3 et value = "d"
```

Les boucles nommées

Encore une autre chose intéressante à connaître : **les boucles nommées** ! Mieux vaut commencer par un exemple :

```
Run the following code:
// 'outer désigne le nom ou label de la boucle ci-dessous :
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        // on continue la boucle sur x
        if x % 2 == 0 { continue 'outer; }

        // on continue la boucle sur y
        if y % 2 == 0 { continue 'inner; }

        println!("x: {}, y: {}", x, y);
    }
}
```

Je pense que vous l'aurez compris, on peut directement reprendre ou arrêter une boucle en utilisant **son nom** (pour peu que vous lui en ayez donné un bien évidemment). Autre exemple :

```
Run the following code:
'global: for _ in 0..10 {
    'outer: for x in 0..10 {
        'inner: for y in 0..10 {
            // on arrête la boucle qui s'appelle global
            if x > 3 { break 'global; }

            // on continue la boucle sur x
            if x % 2 == 0 { continue 'outer; }

            // on continue la boucle sur y
            if y % 2 == 0 { continue 'inner; }

            println!("x: {}, y: {}", x, y);
        }
    }
}
```

Encore une fois, je vous invite à tester pour bien comprendre comment tout ça fonctionne.

9. Les enums

Les "enums" sont très différentes de celles que vous pourriez croiser dans des langages impératifs comme le C ou le C++. Elles ne représentent pas juste des nombres mais bien plus :

Run the following code:
```
enum UneEnum {
    Variant,
    VariantStruct { a: i32, b: i32 },
    VariantTuple(String),
}
```

Chaque "champ" d'une enum est appelé un **variant**. Comme vous avez pu le voir au-dessus, les enums permettent beaucoup plus. Il est cependant aussi possible de déclarer et utiliser des enums plus proches de celles que vous pourriez trouver en C/C++ :

Run the following code:
```
enum UneEnum {
    Variant = 12,
    Variant2,
}
```

`UneEnum::Variant` vaudra donc **12**, par contre `UneEnum::Variant2` ne vaudra pas **13** ! Il vous faudra donner une valeur à chaque variant si vous voulez que ce soit le cas.

Utilisation

Les enums peuvent se révéler très utiles dans beaucoup de cas. Par exemple, vous avez codé un jeu vidéo qui fonctionne au tour par tour (un jeu d'échecs ?). Pendant son tour, le joueur peut bouger une pièce ou bien ne rien faire. On peut exprimer ça de la façon suivante :

Run the following code:
```
enum Action {
    Bouger { piece: Piece, nouvelle_position: Position },
    Passer,
}
```

Si le joueur bouge une pièce, on aura `Action::Bouger` sinon on aura `Action::Passer`.

Un autre exemple avec du "parsing" d'adresse IP :

Run the following code:
```
enum IPKind {
    IPV4(u8, u8, u8, u8),
    IPV6(u32, u32, u32, u32),
}
```

Et cette enum s'utiliserait comme ceci :

Run the following code:
```
fn parse_ip(ip: &str) -> IPKind {
    // "le parsing"
}

match parse_ip("127.0.0.1") {
    IPKind::IPV4(_, _, _, _) => {
        println!("c'est une IPv4 !");
    }
    IPKind::IPV6(_, _, _, _) => {
        println!("c'est une IPv6 !");
    }
}
```

On peut aussi se servir d'une enum pour gérer des erreurs comme le permettent les enums **Result** et **Option** :

Run the following code:
```
match File::open("fichier.txt") {
    Ok(fichier) => {
        // le fichier existe et on a les droits d'accès, on peut s'en servir !
    }
    Err(e) => {
        // le fichier n'existe pas ou on n'a pas les droits d'accès, on ne
        // peut rien faire...
        println!("Erreur en tentant d'ouvrir le fichier : {:?}", e);
    }
}
```

Ce sont 2 utilisations très différentes des enums mais qui sont des éléments très importants permettant à Rust d'empêcher ses utilisateurs d'utiliser des types invalides (comme déréférencer un pointeur nul).

Implémenter des méthodes sur une enum

Tout comme pour les structures (que nous verrons dans le chapitre suivant), il est possible d'implémenter des méthodes (et des traits !) sur des enums. Je vais donner un exemple rapide ici mais j'en parle plus en détails dans le chapitre suivant :

Run the following code:
```
// On reprend notre enum précédente :
enum IPKind {
    IPV4(u8, u8, u8, u8),
    IPV6(u32, u32, u32, u32),
}

impl IPKind {
    fn is_v4(&self) -> bool {
        match *self {
            IPKind::IPV4(_, _, _, _) => true,
            _ => false,
        }
    }

    fn is_v6(&self) -> bool {
        !self.is_v4() // je suis un peu fainéant :)
    }
}
```

Je ne vais pas m'étendre plus sur le sujet et vous invite donc à passer au chapitre suivant pour en savoir plus !

10. Les structures

Comme certains d'entre vous vont s'en rendre compte, les structures sont à la fois très ressemblantes et très différentes de ce que vous pourriez croiser dans d'autres langages (impératifs notamment). Ce chapitre contient beaucoup de nouvelles informations donc n'hésitez surtout pas à prendre votre temps pour être sûr de bien tout comprendre. Commençons donc de ce pas !

À quoi ça ressemble ?

Sachez qu'il existe quatre types de structures en Rust :

- Les structures tuples : une structure dont les champs n'ont pas de nom. Tout comme un tuple, on peut accéder aux champs avec leur position dans la déclaration de la structure.
- Les structures unitaires (on dit aussi structure opaque). Dans le cas où il n'y a pas de données à mettre dans un type, ce sont ces structures qui sont utilisées. Le plus souvent, ce type de structure est utilisé pour assurer des règles de compilation en tant qu'argument de type générique. Nous reviendrons sur la généricité dans la deuxième partie de ce livre.
- Les structures "classiques" : ce sont les plus communes. Elles ont des champs nommés comme ce qu'on peut voir dans des langages comme le C ou le C++.
- Les structures "newtype" : c'est une structure tuple mais avec un seul champ. Cela permet d'ajouter des contrôles sur un type en particulier en le mettant dans un type qui (ré-)implémentera les opérations par-dessus.

Exemple de déclaration pour chacune d'entre elles :

```
Run the following code:
// Une structure tuple
struct Tuple(isize, usize, bool);

// Une structure unitaire
struct Unitaire;

// Une structure "classique"
struct Classique {
    name: String,
    age: usize,
    a_un_chat: bool,
}

// Une structure "newtype"
struct StructureTuple(usize);
```

Maintenant voyons comment on les instancie et on accède à leur(s) champ(s) :

```
Run the following code:
// La structure tuple
let t = Tuple(0, 2, false);
println!("0 : {}, 1 : {}, 2 : {}", t.0, t.1, t.2);

// La structure unitaire
let u = Unitaire;
// Pas de champs donc rien à montrer ici.

// La structure "classique"
let c = Classique {
    // On convertit une `&'static str` en `String`
    name: "Moi".to_owned(),
    age: 18,
    a_un_chat: false,
};
println!("name : {}, age : {}, a_un_chat : {}", c.name, c.age, c.a_un_chat);

// La structure "newtype"
let nt = NewType(1);
println!("valeur : {}", nt.0);
```

Vous devez savoir que, par convention, les noms des structures doivent être écrits en camel case en Rust. Par exemple, appeler une structure "ma_structure" serait "invalide". Il faudrait l'appeler "MaStructure". J'insiste bien sur le fait que ce n'est pas obligatoire, ce n'est qu'une convention. Cependant, il est préférable de la suivre autant que possible car cela facilite la lecture pour les autres développeurs. D'ailleurs, il est important d'ajouter :

Les noms des fonctions, par convention en **Rust**, doivent être écrits en snake case. Donc "MaFonction" est invalide, "ma_fonction" est correct.

Prenons maintenant un exemple d'utilisation de la structure tuple :

Run the following code:
```
// Une distance en mètres.
struct Distance(usize);

impl Distance {
    fn to_kilometre(&self) -> usize {
        self.0 / 1000
    }
}

let distance = Distance(2000);

// On peut récupérer la valeur contenue dans le type de cette façon.
let Distance(longueur) = distance;
println!(
    "La distance est {}m (ou {} km)",
    longueur,
    distance.to_kilometre(),
);
```

Maintenant regardons à quoi sert une structure unitaire. Comme indiqué, elles sont pratiques pour être utilisées pour la généricité (que nous aborderons dans la deuxième partie du livre). Particulièrement dans les ECS (entity component system, ou bien "système de composants d'entité" en français), très utilisés dans les jeux vidéos. Par exemple, les monstres et le joueur sont tous des "personnages" et utilisent donc le même type. Mais pour les différencier, on utilisera des structures unitaires en plus pour les différencier ("joueur" et "monstre").

Déstructuration

Il est possible de déstructurer une structure en utilisant le pattern matching ou le pattern binding :

Run the following code:
```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

// pattern matching
match origin {
    Point { x, y } => println!("({}, {})", x, y),
}
// pattern binding
let Point { x, y } = origin;
println!("({}, {})", x, y);
```

Il est d'ailleurs possible de ne matcher que certains champs en utilisant ".." :

Run the following code:
```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

// pattern matching
match origin {
    Point { y, .. } => println!("(.., {})", y),
}
// pattern binding
let Point { y, .. } = origin;
println!("(.., {})", y);
```

Ici, il ne sera pas possible d'afficher le contenu de "x", car nous l'avons volontairement ignoré lors du matching.

Maintenant que les explications sont faites, voyons comment ajouter des méthodes à une structure.

Les méthodes

Outre le fait qu'ajouter des méthodes à une structure permet de faire de l'orienté-objet, cela peut aussi permettre de forcer un développeur à appeler l'un de vos constructeurs plutôt que de le laisser initialiser tous les éléments de votre type lui-même. Exemple :

Run the following code:
```
pub struct Distance {
    // Ce champ n'est pas public donc impossible d'y accéder directement
    // en-dehors de ce fichier !
    metre: i32,
}

impl Distance {
    pub fn new() -> Distance {
        Distance {
            metre: 0,
        }
    }

    pub fn new_with_value(valeur: i32) -> Distance {
        Distance {
            metre: valeur,
        }
    }
}

// autre fichier
// Si la définition de Distance est dans fichier.rs
mod fichier;

fn main() {
    let d = fichier::Distance::new();
    // ou
    let d = fichier::Distance::new_with_value(10);
}
```

Quel intérêt vous vous demandez ? Après tout, on irait aussi vite de le faire nous-mêmes ! Dans le cas présent, il n'y en a effectivement pas beaucoup. Cependant, imaginez une structure contenant une vingtaine de champs, voire plus. C'est tout de suite plus agréable d'avoir une méthode nous permettant de le faire en une ligne. Maintenant, ajoutons une méthode pour convertir cette distance en kilomètre :

Run the following code:
```
pub struct Distance {
    metre: i32,
}

impl Distance {
    pub fn new() -> Distance {
        Distance {
            metre: 0,
        }
    }

    pub fn new_with_value(valeur: i32) -> Distance {
        Distance {
            metre: valeur,
        }
    }

    pub fn convert_in_kilometers(&self) -> i32 {
        self.metre / 1000
    }
}

// autre fichier
// Si la définition de Distance est dans fichier.rs
mod fichier;

fn main() {
    let d = fichier::Distance::new();
    // ou
    let d = fichier::Distance::new_with_value(10000);

    println!("distance en kilometres : {}", d.convert_in_kilometers());
}
```

Une chose importante à noter est qu'une méthode ne prenant pas **self** en premier paramètre est une méthode **statique**. Les méthodes **new** et **new_with_value** sont donc des méthodes statiques tandis que **convert_in_kilometers** n'en est pas une.

À présent, venons-en au "&" devant le **self** : cela indique que **self** est "prêté" à la fonction. On dit donc que "&self" est une référence vers **self**. Cela est lié au système de propriété de Rust (le fameux "borrow checker"). Nous aborderons cela plus en détails dans un autre chapitre.

Maintenant, si vous voulez créer une méthode pour modifier la distance, il vous faudra spécifier que **self** est mutable (car toutes les variables en **Rust** sont constantes par défaut). Exemple :

Run the following code:
```
impl Distance {
    // les autres méthodes
    // ...

    pub fn set_distance(&mut self, nouvelle_distance: i32) {
        self.metre = nouvelle_distance;
    }
}
```

Tout simplement !

Syntaxe de mise à jour (ou "update syntax")

Une structure peut inclure ".." pour indiquer qu'elle veut copier certains champs d'une autre structure. Exemple :

Run the following code:
```
struct Point3d {
    x: i32,
    y: i32,
    z: i32,
}

let mut point = Point3d { x: 0, y: 0, z: 0 };
// et ici on prend x et z de Point3d
let mut point2 = Point3d { y: 1, .. point };
```

Destructeur

Maintenant voyons comment faire un destructeur (une méthode appelée automatiquement lorsque notre objet est détruit) :

Run the following code:
```
struct Distance {
    metre: i32,
}

impl Distance {
    // fonctions membres
}

impl Drop for Distance {
    fn drop(&mut self) {
        println!("La structure Distance a été détruite !");
    }
}
```

"D'où ça sort ce impl Drop for Distance ?!"

On a implémenté le trait **Drop** sur notre structure **Distance**. Quand l'objet est détruit, cette méthode est appelée. Je sais que cela ne vous dit pas ce qu'est un **trait**, mais nous y reviendrons dans la deuxième partie de ce cours.

11. if let / while let

Maintenant que nous avons vu ce qu'étaient les **enums**, je peux vous parler de `if let` et de `while let`.

Qu'est-ce que le if let ?

Le `if let` permet de simplifier certains traitements de pattern matching. Prenons un exemple :

Run the following code:
```
fn fais_quelque_chose(i: i32) -> Option<String> {
    if i < 10 {
        Some("variable inférieure à 10".to_owned())
    } else {
        None
    }
}
```

Normalement, pour vérifier le retour de cette fonction, vous utiliseriez un `match` :

Run the following code:
```
match fais_quelque_chose(1) {
    Some(s) => println!("{}", &s),
    None => {} // rien à afficher donc on ne fait rien
}
```

Et bien avec le `if let` vous pouvez faire :

Run the following code:
```
if let Some(s) = fais_quelque_chose(1) {
    println!("{}", &s)
}
```

Et c'est tout. Pour faire simple, si le type renvoyé par la fonction `fais_quelque_chose` correspond à celui donné au `if let`, le code du `if` sera exécuté. On peut bien évidemment le coupler avec un `else if` ou avec un `else` :

Run the following code:
```
if let Some(s) = fais_quelque_chose(1) {
    println!("{}", &s)
} else {
    println!("il ne s'est rien passé")
}
```

Essayez en passant un nombre supérieur à 10 comme argument, vous devriez rentrer dans le `else`.

D'ailleurs, je ne l'ai pas précisé dans le chapitre "Conditions et pattern matching" mais il est possible d'être plus précis dans le pattern matching en utilisant plusieurs niveaux de types. Par exemple :

Run the following code:
```
let x = Some(10);

// on rentre dans ce if si x est un Option::Some contenant 10 !
if let Some(10) = x {
    // ...
} else if let Some(11) = x {
    // ...
}
```

Vous pouvez bien évidemment le faire sur autant de "niveaux" que vous le souhaitez :

Run the following code:
```
let x = Ok(Some(Ok(Ok(2))));

if let Ok(Some(Ok(Ok(2)))) = x {
    // ...
}
```

while let

Le `while let` fonctionne de la même façon : tant que le type renvoyé correspondra au type attendu, la boucle continuera. Donc le code suivant :

Run the following code:
```
let mut v = vec!(1, 2, 3);

loop {
    match v.pop() {
        Some(x) => println!("{}", x),
        None => break,
    }
}
```

Deviendra :

Run the following code:
```
let mut v = vec!(1, 2, 3);

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

Déstructuration

Dans le précédent chapitre, je vous ai rapidement montré ce qu'était la déstructuration. Cela fonctionne bien évidemment pour `while let` et `if let` :

Run the following code:
```
struct Point {
    x: i32,
    y: i32,
}

let origin = Point { x: 0, y: 0 };

match origin {
    Point { x, y } => println!("({},{})", x, y),
}
// est équivalent à :
if let Point { x, y } = origin {
    println!("({},{})", x, y);
}
```

12. Gestion des erreurs

Il est courant dans d'autres langages de voir ce genre de code :

```
Objet *obj = creer_objet();

if (obj == NULL) {
    // gestion de l'erreur
}
```

Vous ne verrez (normalement) pas ça en Rust.

Result

Créons un fichier par exemple :

Run the following code:
```
use std::fs::File;

let mut fichier = File::open("fichier.txt");
```

La documentation dit que File::open renvoie un Result. Il ne nous est donc pas possible d'utiliser directement la variable **fichier**. Cela nous "oblige" à vérifier le retour de File::open :

Run the following code:
```
use std::fs::File;

let mut fichier = match File::open("fichier.txt") {
    Ok(f) => {
        // L'ouverture du fichier s'est bien déroulée, on renvoie l'objet
        f
    }
    Err(e) => {
        // Il y a eu un problème, affichons l'erreur pour voir ce qu'il se passe
        println!("erreur : {:?}", e);
        // On ne peut pas renvoyer le fichier ici, donc on quitte la fonction
        return;
    }
};
```

Il est cependant possible de passer outre cette vérification, **mais c'est à vos risques et périls** !

Run the following code:
```
use std::fs::File;

let mut fichier = File::open("fichier.txt").expect("erreur lors de l'ouverture");
```

Si jamais il y a une erreur lors de l'ouverture du fichier, votre programme plantera et vous ne pourrez rien y faire. Il est toutefois possible d'utiliser cette méthode de manière "sûre" avec les fonctions is_ok et is_err :

Run the following code:
```
use std::fs::File;

let mut fichier = File::open("fichier.txt");

if fichier.is_ok() {
    // On peut faire expect !
} else {
    // Il y a eu une erreur, expect impossible !
}
```

Utiliser le pattern matching est cependant préférable.

À noter qu'il existe un équivalent de la méthode **expect** qui s'appelle **unwrap**. Elle fait exactement la même chose mais ne permet pas de fournir un message d'erreur. Pour faire simple : toujours préférer `expect` à `unwrap` !

Option

Vous savez maintenant qu'il n'est **normalement** pas possible d'avoir des objets invalides. Exemple :

Run the following code:
```
let mut v = vec![1, 2];

v.pop(); // retourne Some(2)
v.pop(); // retourne Some(1)
v.pop(); // retourne None
```

Cependant, il est tout à fait possible que vous ayez besoin d'avoir un objet qui serait initialisé plus tard pendant le programme ou qui vous permettrait de vérifier un état. Dans ce cas comment faire ? Option est là pour ça !

Imaginons que vous ayez un vaisseau personnalisable sur lequel il est possible d'avoir des bonus (disons un salon intérieur). Il ne sera pas là au départ, mais peut être ajouté par la suite :

Run the following code:
```
struct Vaisseau {
    // Pleins de champs
    salon: Option<Salon>,
}

impl Vaisseau {
    pub fn new() -> Vaisseau {
        Vaisseau {
            // On initialise les autres champs
            salon: None, // On n'a pas de salon
        }
    }
}

let mut vaisseau = Vaisseau::new();
```

Si jamais vous voulez tester le code, vous pouvez utiliser ce code pour la structure `Salon`:

Run the following code:
```
// On définit une structure "Salon" vide pour l'exemple.
struct Salon {}

impl Salon {
    fn new() -> Salon {
        Salon {}
    }
}
```

Donc pour le moment, on n'a pas de salon. Maintenant nous en ajoutons un :

Run the following code:
```
vaisseau.salon = Some(Salon::new());
```

Je présume que vous vous demandez comment accéder au salon maintenant. Tout simplement comme ceci :

Run the following code:
```
match vaisseau.salon {
    Some(s) => {
        println!("ce vaisseau a un salon");
    }
    None => {
        println!("ce vaisseau n'a pas de salon");
    }
}
```

Au début, vous risquez de trouver ça agaçant, mais la sécurité que cela apporte est un atout non négligeable ! Cependant, tout comme avec **Result**, vous pouvez utiliser la méthode **expect**.

Run the following code:
```
vaisseau.salon = Some(Salon::new());

// Pas recommandé !!!
let salon =  vaisseau.salon.expect("pas de salon");
```

Tout comme avec **Result**, il est possible de se passer du mécanisme de pattern matching avec les méthodes **is_some** et **is_none** :

Run the following code:
```
if vaisseau.salon.is_some() {
    // On peut utiliser expect !
} else {
    // Ce vaisseau ne contient pas de salon !
}
```

Encore une fois, utiliser le pattern matching est préférable.

panic!

panic! est une macro très utile puisqu'elle permet de "quitter" le programme. Elle n'est à appeler que lorsque le programme subit une erreur irrécupérable. Elle est très simple d'utilisation :

Run the following code:
```
panic!();
// panic avec une valeur de 4 pour la récupérer ailleurs (hors
// du programme par exemple)
panic!(4);
panic!("Une erreur critique vient d'arriver !");
panic!("Une erreur critique vient d'arriver : {}", "le moteur droit est mort");
```

Et c'est tout.

Question !

Pour les codes que nous avons vu au-dessus, il serait actuellement possible de les écrire de manière plus courte :

Run the following code:
```
use std::fs::File;
use std::io;

fn foo() -> io::Result<u32> {
    let mut fichier = File::open("fichier.txt")?;
    // ...
    Ok(0)
}
```

La différence étant que nous avons utilisé l'opérateur **?**. Pour pouvoir s'en servir, plusieurs conditions doivent être réunies.

Tout d'abord, on ne peut utiliser **?** que sur des types implémentant le trait Try (nous reviendrons sur ce qu'est un **trait** dans un prochain chapitre). Il faut aussi que la fonction renvoie la même chose que le type sur lequel on utilise le **?** (c'est pourquoi notre fonction **foo** renvoie `io::Result`). Dans le cas où votre fonction ne renvoie pas la même chose, il est possible de changer l'erreur pour que ça corresponde :

Run the following code:
```
use std::fs::File;

fn foo() -> Result<u32, String> {
    let mut fichier = File::open("fichier.txt")
        // On change io::Error en String avec "map_err" si File::open
        // renvoie une erreur.
        .map_err(|e| format!("open error {:?}", e))?;
    Ok(0)
}
```

Voilà pour ce chapitre, vous devriez maintenant être capables de créer des codes un minimum "sécurisés".

13. Cargo

Rust possède un gestionnaire de paquets : Cargo. Il permet aussi de grandement faciliter la gestion de la compilation (en permettant de faire des builds personnalisées notamment) ainsi que des dépendances externes. Toutes les informations que je vais vous donner dans ce chapitre peuvent être retrouvées ici (en anglais). N'hésitez pas à y faire un tour !

Pour commencer un projet avec **Cargo**, rien de plus facile :

```
cargo new mon_nouveau_projet
```

Un nouveau dossier s'appelant **mon_nouveau_projet** sera créé :

```
 - mon_nouveau_projet
     |
     |- Cargo.toml
     |- .gitignore
     |- src/
```

Le fichier **Cargo.toml** à la racine de votre projet devrait contenir :

```
[package]
name = "mon_nouveau_projet"
version = "0.0.1"
authors = ["Votre nom <vous@exemple.com>"]
```

Tous les fichiers sources (**.rs** normalement) doivent être placés dans un sous-dossier appelé **src**. C'est à dire qu'on va avoir un fichier **main.rs** dans le dossier **src** :

```
Run the following code:
fn main() {
    println!("Début du projet");
}
```

Maintenant pour compiler le projet, il vous suffit de faire :

```
cargo build
```

L'exécutable sera généré dans le dossier **target/debug/**. Pour le lancer :

```
$ ./target/debug/mon_nouveau_projet
Début du projet
```

Si vous voulez compiler et lancer l'exécutable tout de suite après, vous pouvez utiliser la commande **run** :

```
$ cargo run
     Fresh mon_nouveau_projet v0.0.1 (file:///path/to/project/mon_nouveau_projet)
    Running `target/debug/mon_nouveau_projet`
Début du projet
```

Par défaut, **cargo** compile en mode **debug**. Les performances sont **BEAUCOUP** plus faibles qu'en mode **release**, faites donc bien attention à vérifier que vous n'avez pas compilé en mode **debug** dans un premier temps si vous avez des problèmes de performance. Si vous souhaitez compiler en mode release, il vous faudra passer l'option "--release" :

```
cargo build --release
```

Bien évidemment, l'exécutable généré se trouvera dans le dossier **target/release**.

Cela fonctionne de la même façon pour lancer l'exécution :

```
cargo run --release
```

Gérer les dépendances

Si vous voulez utiliser une bibliothèque externe, **cargo** peut le gérer pour vous. Il y a plusieurs façons de faire :

- Soit la bibliothèque est disponible sur crates.io, et dans ce cas il vous suffira de préciser la version que vous désirez.
- Soit elle ne l'est pas : dans ce cas vous pourrez indiquer son chemin d'accès si elle est présente sur votre ordinateur, ou bien vous pourrez donner l'adresse de son dépôt git.

Avant d'aller plus loin, il est important de noter : les paquets sont appelés des crates en Rust ("cagette" en français), d'où le nom "crates.io". Il sera donc fréquent que ce mot soit utilisé à la place de "bibliothèque".

Par exemple, vous voulez utiliser la crate **sysinfo**, elle est disponible sur crates.io ici, donc pas de souci :

```
[package]
name = "mon_nouveau_projet"
version = "0.0.1"
authors = ["Votre nom <vous@exemple.com>"]

[dependencies]
sysinfo = "0.27.0"
```

Nous avons donc ajouté **sysinfo** comme dépendance à notre projet. Détail important : **à chaque fois que vous ajoutez/modifiez/supprimez une dépendance, il vous faudra relancer cargo build pour que ce soit pris en compte !** D'ailleurs, si vous souhaitez mettre à jour les crates que vous utilisez, il vous faudra utiliser la commande :

```
cargo update
```

Je ne rentrerai pas plus dans les détails concernant l'utilisation d'une bibliothèque externe ici car le chapitre suivant traite ce sujet.

Si vous voulez utiliser une version précise (antérieure) de **sysinfo** , vous pouvez la préciser comme ceci :

```
[dependencies]
sysinfo = "0.18.0"
```

Il est cependant possible de faire des choses un peu plus intéressantes avec la gestion des versions. Par exemple, vous pouvez autoriser certaines versions de la crate :

Le "^" permet notamment :

```
^1.2.3 := >=1.2.3 <2.0.0
^0.2.3 := >=0.2.3 <0.3.0
^0.0.3 := >=0.0.3 <0.0.4
^0.0 := >=0.0.0 <0.1.0
^0 := >=0.0.0 <1.0.0
```

Le "~" permet :

```
~1.2.3 := >=1.2.3 <1.3.0
~1.2 := >=1.2.0 <1.3.0
~1 := >=1.0.0 <2.0.0
```

Le "*" permet :

```
* := >=0.0.0
1.* := >=1.0.0 <2.0.0
1.2.* := >=1.2.0 <1.3.0
```

Et enfin les symboles d'(in)égalité permettent :

```
>= 1.2.0
> 1
< 2
= 1.2.3
```

Il est possible de mettre plusieurs exigences en les séparant avec une virgule : `>= 1.2, < 1.5.`.

Maintenant regardons comment ajouter une dépendance à une crate qui n'est pas sur crates.io (ou qui y est mais pour une raison ou pour une autre, vous ne voulez pas passer par elle).

```
[package]
name = "mon_nouveau_projet"
version = "0.0.1"
authors = ["Votre nom <vous@exemple.com>"]

[dependencies.sysinfo]
git = "https://github.com/GuillaumeGomez/sysinfo"
```

Ici nous avons indiqué que la crate **sysinfo** se trouvait à cette adresse de github. Il est aussi possible que vous l'ayez téléchargé, dans ce cas il va vous falloir indiquer où elle se trouve :

```
[dependencies.sysinfo]
path = "chemin/vers/sysinfo"
```

Voici en gros à quoi ressemblerait un fichier cargo :

```
[package]
name = "mon_nouveau_projet"
version = "0.0.1"
authors = ["Votre nom <vous@exemple.com>"]

[dependencies.sysinfo]
git = "https://github.com/GuillaumeGomez/sysinfo"

[dependencies.gsl]
version = "0.0.1" # optionnel
path = "path/vers/gsl"

[dependencies]
sdl = "0.3"
cactus = "0.2.3"
```

Publier une crate sur crates.io

Vous avez fait une crate et vous avez envie de la mettre à disposition des autres développeurs ? Pas de soucis ! Tout d'abord, il va vous falloir un compte sur crates.io (pour le moment il semblerait qu'il faille obligatoirement un compte sur github pour pouvoir se connecter sur crates.io). Une fois que c'est fait, allez sur la page de votre compte. Vous devriez voir ça écrit dessus :

```
cargo login abcdefghijklmnopqrstuvwxyz012345
```

Exécutez cette commande sur votre ordinateur pour que cargo puisse vous identifier. **IMPORTANT : CETTE CLEF NE DOIT PAS ETRE TRANSMISE !!!** Si jamais elle venait à être divulguée à quelqu'un d'autre que vous-même, supprimez-la et régénérez-en une nouvelle aussitôt !

Regardons maintenant les metadata que nous pouvons indiquer pour permettre "d'identifier" notre crate :

- **description** : Brève description de la crate.
- **documentation** : URL vers la page où se trouve la documentation de votre crate.
- **homepage** : URL vers la page de présentation de votre crate.
- **repository** : URL vers le dépôt où se trouve le code source de votre crate.
- **readme** : Chemin de l'emplacement du fichier README (relatif au fichier **Cargo.toml**).
- **keywords** : Mots-clés permettant pour catégoriser votre crate sur crates.io.

- **license** : Licence(s) de votre crate. On peut en mettre plusieurs en les séparant avec un '/'. La liste des licences disponibles se trouve ici.
- **license-file** : Si la licence que vous cherchez n'est pas dans la liste de celles disponibles, vous pouvez donner le chemin du fichier contenant la vôtre (relatif au fichier **Cargo.toml**).

Je vais vous donner ici le contenu (un peu raccourci) du fichier **Cargo.toml** de la crate **sysinfo** pour que vous ayez un exemple :

```
[package]
name = "sysinfo"
version = "0.27.0"
authors = ["Guillaume Gomez <guillaume1.gomez@gmail.com>"]

description = "Library to get system information such as processes, CPUs, disks, components and
repository = "https://github.com/GuillaumeGomez/sysinfo"
license = "MIT"
readme = "README.md"
rust-version = "1.59"
exclude = ["/test-unknown"]

categories = ["filesystem", "os", "api-bindings"]

build = "build.rs"
edition = "2018"

[dependencies]
cfg-if = "1.0"
rayon = { version = "^1.5.1", optional = true }

[features]
default = ["multithread"]
multithread = ["rayon"]
```

Voilà ! Comme vous pouvez le voir, il y a aussi une option `[features]`. Elle permet dans le cas de **sysinfo** de désactiver le multi-threading. Vous pouvez utiliser les features comme la gestion de version d'une bibliothèque C. Par exemple, seulement la version 1.0 est "activée" par défaut, et si l'utilisateur utilise une version plus récente il devra activer la feature correspondante (`v1_1` ou `v1_2` par exemple). Il est important de noter cependant qu'il n'y a rien de normalisé à ce niveau donc à vous de regarder quand vous utilisez une crate si elles possèdent plus de features qui pourraient vous intéresser.

Nous voici enfin à la dernière étape : **publier la crate**. **ATTENTION : une crate publiée ne peut pas être supprimée !** Il n'y a pas de limite non plus sur le nombre de versions qui peuvent être publiées.

Le nom sous lequel votre crate sera publiée est celui donné par la metadonnée **name** :

```
[package]
name = "super"
```

Si une crate portant le nom "super" est déjà publiée sur crates.io, vous ne pourrez rien y faire, il faudra trouver un autre nom. Une fois que tout est prêt, utilisez la commande :

```
cargo publish
```

Et voilà, votre crate est maintenant visible sur crates.io et peut être utilisée par tout le monde !

Si vous voulez faire un tour plus complet de ce que **Cargo** permet de faire, je vous recommande encore une fois d'aller lire le **Cargo book** (en anglais).

14. Utiliser des bibliothèques externes

Nous avons vu comment gérer les dépendances vers des bibliothèques externes dans le précédent chapitre, il est temps de voir comment s'en servir.

Commençons par le fichier **Cargo.toml**, ajoutez ces deux lignes :

```
[dependencies]
time = "0.1"
```

Nous avons donc ajouté une dépendance vers la crate **time**. Pour appeler une fonction de cette crate, il suffit de faire :

Run the following code:
```
println!("{:?}", time::now());
```

Et c'est tout ! Les imports fonctionnent de la même façon :

Run the following code:
```
use time::Tm;
```

Je vous le rappelle : vous pouvez voir toutes les crates disponibles sur le site <crates.io>.

Voilà qui conclut ce (bref) chapitre !

15. Jeu du plus ou moins

Le but de ce chapitre est de mettre en pratique ce que vous avez appris dans les chapitres précédents au travers de l'écriture d'un **jeu du plus ou moins**. Voici le déroulement :

1. L'ordinateur choisit un nombre (on va dire entre 1 et 100).
2. Vous devez deviner le nombre.
3. Vous gagnez si vous le trouvez en moins de 10 essais.

Exemple d'une partie :

```
Génération du nombre...
C'est parti !
Entrez un nombre : 50
-> C'est plus grand
Entrez un nombre : 75
-> C'est plus petit
Entrez un nombre : 70
Vous avez gagné !
```

La grande inconnue de l'écriture de ce jeu est de savoir comment générer un nombre aléatoirement. Pour cela, nous allons utiliser la crate rand. Ajoutez-la comme dépendance dans votre fichier **Cargo.toml** comme vu dans le chapitre précédent. Maintenant, pour générer un nombre il vous suffira de faire :

Run the following code:
```
use rand::Rng;

fn main() {
    let nombre_aleatoire = rand::thread_rng().gen_range(1..=100);
}
```

Il va aussi falloir récupérer ce que l'utilisateur écrit sur le clavier. Pour cela, utilisez la méthode read_line de l'objet Stdin (qu'on peut récupérer avec la fonction stdin). Il ne vous restera plus qu'à convertir cette String en entier en utilisant la méthode from_str. Je pense vous avoir donné assez d'indications pour que vous puissiez vous débrouiller seuls. Bon courage !

Je propose une solution juste en dessous pour ceux qui n'y arriveraient pas ou qui souhaiteraient tout simplement comparer leur code avec le mien.

La solution

J'ai écrit cette solution en essayant de rester aussi clair que possible sur ce que je fais.

Commençons par la fonction qui se chargera de nous retourner le nombre entré par l'utilisateur :

Run the following code:
```
use std::io;
use std::str::FromStr;

// Elle ne prend rien en entrée et retourne un Option<isize> (dans le cas où ça
// ne fonctionnerait pas).
fn recuperer_entree_utilisateur() -> Option<isize> {
    let mut entree = String::new();

    // On récupère ce qu'a entré l'utilisateur dans la variable "entree".
    if let Err(err) = io::stdin().read_line(&mut entree).is_err() {
        // Une erreur s'est produite, on doit avertir l'utilisateur !
        println!("Erreur lors de la récupération de la saisie : {:?}", err);
        return None;
    }
    // Tout s'est bien passé, on peut convertir la String en entier.
    // La méthode "trim" enlève tous les caractères "blancs" en début et fin
    // de chaîne.
    match isize::from_str(&entree.trim()) {
        // Tout s'est bien déroulé, on retourne donc le nombre.
        Ok(nombre) => Some(nombre),
        // Si jamais la conversion échoue (si l'utilisateur n'a pas rentré un
        // nombre valide), on retourne "None".
        Err(_) => {
            println!("Veuillez entrer un nombre valide !");
            None
        }
    }
}
```

Voilà une bonne chose de faite ! Il va nous falloir à présent implémenter le coeur du jeu :

Run the following code:
```
// Utilisé pour "flusher" la sortie console.
use std::io::Write;

fn jeu() -> bool {
    // On va mettre 10 tentatives avant de dire au joueur qu'il a perdu.
    let mut tentative = 10;

    println!("Génération du nombre...");
    let nombre_aleatoire = rand::thread_rng().gen_range(1..=100);
    println!("C'est parti !");
    while tentative > 0 {
        // On ne veut pas de retour à la ligne !
        print!("Entrez un nombre : ");
        // Si on n'utilise pas cette méthode, on ne verra pas l'affichage de
        // print! tout de suite
        io::stdout().flush();
        match recuperer_entree_utilisateur() {
            Some(nombre) => {
                if nombre < nombre_aleatoire {
                    println!("C'est plus grand !");
                } else if nombre > nombre_aleatoire {
                    println!("C'est plus petit !");
                } else {
                    return true;
                }
            }
            None => {}
        }
        tentative -= 1;
    }
    false
}
```

Il ne nous reste désormais plus qu'à appeler cette fonction dans notre fonction `main` et le tour est joué !

Run the following code:
```
fn main() {
    println!("=== Jeu du plus ou moins ===");
    println!("");
    if jeu() {
        println!("Vous avez gagné !");
    } else {
        println!("Vous avez perdu…");
    }
}
```

Voici maintenant le code complet (non commenté) de ma solution :

Run the following code:
```
use rand::Rng;
use std::io::Write;
use std::io;
use std::str::FromStr;

fn recuperer_entree_utilisateur() -> Option<isize> {
    let mut entree = String::new();

    if io::stdin().read_line(&mut entree).is_err() {
        println!("Erreur lors de la récupération de la saisie...");
        return None;
    }
    match isize::from_str(&entree.trim()) {
        Ok(nombre) => Some(nombre),
        Err(_) => {
            println!("Veuillez entrer un nombre valide !");
            None
        }
    }
}

fn jeu() -> bool {
    let mut tentative = 10;

    println!("Génération du nombre...");
    let nombre_aleatoire = rand::thread_rng().gen_range(1..=100);
    println!("C'est parti !");
    while tentative > 0 {
        print!("Entrez un nombre : ");
        io::stdout().flush();
        match recuperer_entree_utilisateur() {
            Some(nombre) => {
                if nombre < nombre_aleatoire {
                    println!("C'est plus grand !");
                } else if nombre > nombre_aleatoire {
                    println!("C'est plus petit !");
                } else {
                    return true;
                }
            }
            None => {}
        }
        tentative -= 1;
    }
    false
}

fn main() {
    println!("=== Jeu du plus ou moins ===");
    println!("");
    if jeu() {
        println!("Vous avez gagné !");
    } else {
        println!("Vous avez perdu…");
    }
}
```

Si vous avez un problème, des commentaires ou autres à propos de cette solution, n'hésitez pas à ouvrir une issue sur github.

Améliorations

Il est possible d'ajouter quelques améliorations à cette version comme :

- Un mode 2 joueurs (un joueur choisit un nombre, l'autre le devine).
- Proposer la possibilité de recommencer quand on a fini une partie.
- Afficher le nombre de coups qu'il a fallu pour gagner (et pourquoi pas sauvegarder les meilleurs scores ?).
- Proposer plusieurs modes de difficulté.
- …

Les choix sont vastes, à vous de faire ce qui vous tente le plus !

II. Spécificités de Rust

1. Le formatage des flux

Nous allons commencer cette deuxième partie par un chapitre relativement simple : le formatage des flux.

Exemple de print! et println!

Pour le moment, nous nous sommes contentés de faire de l'affichage sans y mettre de forme. Sachez toutefois qu'il est possible de modifier l'ordre dans lequel sont affichés les arguments sans pour autant changer l'ordre dans lesquels vous les passez à la macro. Démonstration :

```
Run the following code:
println!("{} {} {}", "Bonjour", "à", "tous !");
println!("{1} {0} {2}", "à", "Bonjour", "tous !");
```

Le code que je vous ai montré n'a pas un grand intérêt mais il sert au moins à montrer que c'est possible. Cependant on peut faire des choses nettement plus intéressantes comme limiter le nombre de chiffres après la virgule.

```
Run the following code:
let nombre_decimal: f64 = 0.56545874854551248754;

println!("{:.3}", nombre_decimal);
```

Pas mal, hein ? Hé bien sachez qu'il y a un grand nombre d'autres possibilités comme :

```
Run the following code:
let nombre = 6i32;
let nombre2 = 16i32;

println!("{:b}", nombre); // affiche le nombre en binaire
println!("{:o}", nombre); // affiche le nombre en octal (base 8)
println!("{:x}", nombre); // affiche le nombre en "petit" hexadécimal (base 16)
println!("{:X}", nombre); // affiche le nombre en "grand" hexadécimal (base 16)
println!("{:08}", nombre);  // affiche "00000006"
println!("{:08}", nombre2); // affiche "00000016"
```

Vous pouvez aussi faire en sorte que l'affichage s'aligne sur une colonne et pleins d'autres choses encore. Comme vous vous en rendrez compte par vous-même, il y a beaucoup de possibilités. Vous pourrez trouver tout ce que vous voulez à ce sujet ici (la doc officielle !).

À partir de la version 1.58 de Rust, il est aussi possible d'utiliser des arguments nommés :

```
Run the following code:
let nombre = 6i32;
println!("{nombre}");
// Qui revient au même qu'écrire :
println!("{}", nombre);
```

Tous les formatages vus juste au-dessus fonctionnent bien évidemment avec les arguments nommés :

```
Run the following code:
let nombre = 6i32;
println!("{nombre:b}"); // affiche le nombre en binaire
println!("{nombre:08}");  // affiche "00000006"
```

format!

Comme vous vous en doutez, c'est aussi une macro. Elle fonctionne de la même façon que **print!** et **println!**, mais au lieu d'écrire sur la sortie standard (votre console la majorité du temps), elle renvoie une **String**. Plus d'infos ici (oui, encore la doc !).

```
Run the following code:
let entier = 6i32;
let s_entier = format!("{}", entier);
```

Une façon simple (mais pas très efficace) de convertir un nombre en **String**.

Toujours plus loin !

Sachez que vous pouvez vous servir du formatage de la même façon pour écrire dans des fichiers ou sur tout autre type implémentant le trait **Write** (et il y en a pas mal !). Vous pouvez même faire ceci si vous le voulez :

```
Run the following code:
// On importe le trait Write...
use std::io::Write;

let mut w = Vec::new();
// ... et on l'utilise sur notre Vec !
write!(&mut w, "test");
```

Et oui, encore une autre macro ! Ne vous en faites pas, c'est la dernière... en tout cas pour l'instant ! C'était juste pour vous montrer à quel point le formatage des flux pouvait aller loin.

Je présume que vous vous dites aussi : "c'est quoi cette histoire de trait ?!". Hé bien voyons ça de suite dans le prochain chapitre !

2. Les traits

Commençons par donner une rapide définition : un trait est une interface abstraite que des types peuvent implémenter et qui est composé d'éléments associés (méthodes, types et constantes).

Dans le chapitre sur les structures, il nous fallait implémenter la méthode **drop** pour pouvoir implémenter le trait **Drop**. Et au cas où vous ne vous en doutiez pas, sachez que les traits sont utilisés partout en Rust et en sont une des briques fondamentales. On en retrouve même sur des types primitifs comme les **i32** ou les **f64** !

On va prendre un exemple tout simple : additionner deux **f64**. La doc nous dit ici que le trait **Add** a été implémenté sur le type

f64. Ce qui nous permet de faire :

```
Run the following code:
let valeur = 1f64;

println!("{}", valeur + 3f64);
```

Add était un trait importé "par défaut". Il n'est donc pas nécessaire de l'importer pour pouvoir se servir de lui. Cependant, dans la majorité des cas, il faudra importer un trait pour pouvoir utiliser les méthodes/constantes/types qui y sont associées. Exemple :

```
Run the following code:
// On importe le trait FromStr...
use std::str::FromStr;

// Ce qui nous permet d'avoir accès à la méthode from_str.
println!("{}", f64::from_str("3.6").expect("conversion failed"));
```

Facile n'est-ce pas ? Les traits fournis par la bibliothèque standard et implémentés sur les types standards apportent beaucoup de fonctionnalités. Si jamais vous avez besoin de quelque chose, il y a de fortes chances que ça existe déjà. À vous de chercher.

Je vous ai montré comment importer et utiliser un trait, maintenant il est temps de voir comment en créer un !

Créer un trait

C'est relativement similaire à la création d'une structure :

```
Run the following code:
trait Animal {
    fn get_espece(&self) -> &str;
}
```

Facile, n'est-ce pas ? Maintenant un petit exemple :

Run the following code:
```
trait Animal {
    fn get_espece(&self) -> &str;
    fn get_nom(&self) -> &str;
}

struct Chien {
    nom: String,
}

impl Animal for Chien {
    fn get_espece(&self) -> &str {
        "Chien"
    }

    fn get_nom(&self) -> &str {
        &self.nom
    }
}

struct Chat {
    nom: String,
}

impl Animal for Chat {
    fn get_espece(&self) -> &str {
        "Chat"
    }

    fn get_nom(&self) -> &str {
        &self.nom
    }
}

let chat = Chat { nom: String::from("Fifi") };
let chien = Chien { nom: String::from("Loulou") };

println!("{} est un {}", chat.get_nom(), chat.get_espece());
println!("{} est un {}", chien.get_nom(), chien.get_espece());
```

Je tiens à vous rappeler qu'il est tout à fait possible d'implémenter un trait disponible dans la bibliothèque standard comme je l'ai fait avec le trait **Drop**.

Il est aussi possible d'écrire une implémentation "par défaut" de la méthode directement dans le trait. Ça permet d'éviter d'avoir à réécrire la méthode pour chaque objet sur lequel le trait est implémenté. Exemple :

Run the following code:
```
trait Animal {
    fn get_espece(&self) -> &str;

    fn presentation(&self) -> String {
        format!("Je suis un {} !", self.get_espece())
    }
}

impl Animal for Chat {
    fn get_espece(&self) -> &str {
        "Chat"
    }
}
```

Ici, je ne définis que la méthode **get_espece** car **presentation** fait déjà ce que je veux.

Vous n'en voyez peut-être pas encore l'intérêt mais sachez cependant que c'est vraiment très utile. Quoi de mieux qu'un autre exemple pour vous le prouver ?

Run the following code:
```
fn afficher_infos<T: Animal>(animal: &T) {
    println!("{} est un {}", animal.get_nom(), animal.get_espece());
}
```

"C'est quoi ce `<T: Animal>` ?!"

Pour ceux qui ont fait du C++ ou du Java, c'est relativement proche des templates. Pour les autres, sachez juste que les templates ont e?te? invente?s pour permettre d'avoir du code **générique** (aussi appele? **polymorphisme**). Prenons un autre exemple :

Run the following code:
```
fn affiche_chat(chat: &Chat) {
    println!("{} est un {}", chat.get_nom(), chat.get_espece());
}

fn affiche_chien(chien: &Chien) {
    println!("{} est un {}", chien.get_nom(), chien.get_espece());
}
```

Dans le cas présent, ça va, cela ne représente que deux fonctions. Maintenant si on veut ajouter 40 autres espèces d'animaux, on devrait écrire une fonction pour chacune ! Pas très pratique... Utiliser la généricité est donc la meilleure solution. Et c'est ce dont il sera question dans le prochain chapitre !

Les supertraits

On appelle **supertrait** (en un seul mot) les traits qui sont requis pour l'implémentation d'un trait.

Run the following code:
```
trait Machine {}

// On ajoute "Machine" en tant que supertrait de "Car".
trait Car: Machine {}

struct FastCar;

impl Car for FastCar {}
```

Si on essaie de compiler ce code, nous aurons cette erreur:

```
error[E0277]: the trait bound `FastCar: Machine` is not satisfied
the trait `Machine` is not implemented for `FastCar`
```

Donc si l'on souhaite implémenter le trait `Car` sur un type, il faudra obligatoirement que ce type implémente aussi le trait `Machine`. Prenons l'exemple de la crate sysinfo : elle fournit des informations système, cependant chaque système supporté doit avoir sa propre implémentation (car chacun fournit des APIs très différentes pour récupérer les mêmes informations). Pour s'assurer que chaque plateforme fournit bien les mêmes fonctionnalités, elle utilise des traits. Cependant, on veut aussi que ces types implémentent aussi certains traits comme **Debug**. Hé bien c'est possible grâce aux supertraits.

Autre information intéressante, le trait peut utiliser tout ce qui est défini dans le supertrait dans ses implémentations par défaut :

Run the following code:
```
trait Machine {
    fn serial_id(&self) -> u32;
}

trait Car: Machine {
    fn modele(&self) -> String;
    fn type_de_voiture(&self) -> String {
        // Ici nous utilisons la méthode "serial_id" qui vient du
        // supertrait "Machine".
        format!("{} (serial ID: {})", self.modele(), self.serial_id())
    }
}
```

Ce n'est donc pas de l'héritage bien que cela puisse y ressembler. Plutôt un moyen d'ajouter des conditions d'implémentation sur un trait pour s'assurer qu'il a bien tous les prérequis souhaités.

Les derive traits

Rust fournit la possibilité d'avoir des implémentations de traits "par défaut". Si tous les champs d'une structure implémentent le trait **Debug**, il est possible de ne pas avoir à implémenter le trait avec une implémentation "normale" mais d'utiliser à la place un derive trait :

Run the following code:
```
// Le trait Debug est implémenté avec le "derive".
#[derive(Debug)]
struct Foo {
    a: u32,
    b: f64,
}

let foo = Foo { a: 0, b: 1. };
// On peut donc s'en servir directement.
println!("{:?}", foo);
```

Il y a plusieurs traits qui peuvent être implémentés de la sorte tels que **Display**, **Clone**, **Ord**, **PartialOrd**, **Eq**, **PartialEq**... Et certaines crates en ajoutent encore d'autres ! Tout cela est possible grâce aux macros procédurales (aussi appelées "proc-macros") mais c'est un concept avancé de Rust donc nous y reviendrons dans la dernière partie de ce livre.

Utilisation de traits

Avant de conclure ce chapitre, j'en profite maintenant pour vous montrer quelques utilisations de traits comme **Range** (que l'on avait déjà rapidement abordé dans le chapitre des boucles) et **Index**. Ce dernier peut vous permettre de faire :

Run the following code:
```
let s = "hello";

println!("{}", s);
println!("{}", &s[0..2]);
println!("{}", &s[..3]);
println!("{}", &s[3..]);
```

Ce qui donnera :

```
hello
he
hel
lo
```

Cela fonctionne aussi sur les slices :

Run the following code:

```
// On crée un slice contenant 10 '\0'.
let v: &[u8] = &[0; 10];

println!("{:?}", &v[0..2]);
println!("{:?}", &v[..3]);
println!("{:?}", &v[3..]);
```

Ce qui donne :

```
[0, 0]
[0, 0, 0]
[0, 0, 0, 0, 0, 0, 0]
```

Voilà qui devrait vous donner un petit aperçu de tout ce qu'il est possible de faire avec les traits. Il est maintenant temps de parler de la généricité.

3. Les attributs

Il est possible d'ajouter des métadonnées sur des éléments dans un code. Ces informations peuvent servir à fournir des informations au compilateur ou bien à d'autres outils lisant ce code. Par exemple, changer le nom de la crate :

```
Run the following code:
#![crate_name = "nom_de_cagette"]
```

Les attributs ont 2 formes : externe et interne.

Dans l'exemple au dessus, c'est la forme interne qui est utilisée. Cela signifie que l'effet de l'attribut est appliqué sur l'élément dans lequel il se trouve. Ils s'écrivent sous la forme `#![]`.

À l'inverse, la forme externe signifie que l'effet de l'attribut est appliqué sur l'élément qui suit. Ils s'écrivent sous la forme `#[]` (donc pas de `!`).

Exemples:

```
Run the following code:
#![allow(non_camel_case_types)] // Appliqué sur le module courant.

#[allow(dead_code)] // Appliqué sur `module`.
mod module {
    #![allow(dead_code)] // Appliqué sur `module`.

    #[allow(unused_variables)] // Appliqué sur `fonction`.
    fn fonction() {}
}
```

Il y a 4 types d'attributs :

- Les attributs intégrés au compilateur de Rust ("built-in")
- Les attributs d'outils
- Les macros attributs
- Les attributs derive

Les attributs intégrés

Dans les 2 exemples ci-dessus, nous avons utilisé ces attributs. Comme il y en a beaucoup, nous n'allons n'en lister que quelques uns. La liste complète est ici.

allow, warn et deny

Quand vous compilez, Rust émet des messages d'avertissement ou d'erreur quand on compile. On les appelle des **lints**. Ces lints ont chacun un "niveau" que l'on peut changer grâce à ces attributs.

Par exemple, `unused_varibles` est de niveau "warning" par défaut. Si on souhaite l'ignorer, on peut utiliser `allow`, par contre si on veut qu'il arrête la compilation si jamais il est émis, on utilisera `deny`.

must_use

Cet attribut est très intéressant car quand il est utilisé sur un type, il rend l'utilisation de ce type "obligatoire" :

Run the following code:
```
#[must_use]
struct Struct;

impl Struct {
    fn init() -> Struct {
        Struct
    }
}

fn main() {
    Struct::init();
}
```

Quand on compile ce code, le compilateur affiche :

```
warning: unused `Struct` that must be used
  --> src/main.rs:11:5
   |
11 |     Struct::init();
   |     ^^^^^^^^^^^^^^
```

Sur une fonction, il force l'utilisation de la valeur de retour :

Run the following code:
```
#[must_use]
fn fonction() -> bool {
    true
}

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

Ce qui donne :

```
warning: unused return value of `fonction` that must be used
 --> src/main.rs:7:5
  |
7 |     fonction();
  |     ^^^^^^^^^^
```

Les attributs d'outils

Ces attributs ne sont pas fournis par le compilateur de Rust mais par des outils externes. Par exemple **rustfmt** que l'on peut lancer avec la commande `cargo fmt` pour formater notre code. Exemple :

Run the following code:
```
#[rustfmt::skip]
fn fonction()
{
  let variable =     "12";
}
```

Avec `#[rustfmt::skip]`, on dit à `rustfmt` de ne pas formater l'élément qui suit (donc la fonction `fonction`).

Les macros attributs

Ces attributs sont uniquement des attributs **externes**. Ce sont, comme leur nom l'indique, des **macros**. Ce qui signifie qu'ils vont modifier l'élément sur lequel ils sont utilisés. Comme cela aborde des concepts plus avancés de Rust, ils sont abordés dans la dernière partie de ce livre, dans le chapitre "Les macros procédurales".

Les attributs derive

Ils sont aussi appelés les "derive macros" et sont uniquement des attributs **externes**. Nous les avons déjà évoqué dans le chapitre sur les **traits** donc rien de nouveau ici. Pour rappel, ils ressemblent à ça :

```
Run the following code:
#[derive(Debug)]
pub struct Struct;
```

Ils permettent d'ajouter des implémentations sur des types. Dans l'exemple ci-dessus, on implémente le trait **Debug** sur notre structure `Struct`.

Tout comme les macros attributs, ce sont des macros. Ils sont aussi abordés dans la dernière partie de ce livre, dans le chapitre "Les macros procédurales".

4. Généricité

Reprenons donc notre précédent exemple :

Run the following code:
```
fn affiche_chat(chat: &Chat) -> String {
    println!("{} est un {}", chat.get_nom(), chat.get_espece());
}

fn affiche_chien(chien: &Chien) -> String {
    println!("{} est un {}", chien.get_nom(), chien.get_espece());
}
```

Comme je vous le disais, avec deux espèces d'animaux, ça ne représente que 2 fonctions, mais ça deviendra très vite long à écrire si on veut en rajouter 40. C'est donc ici qu'intervient la généricité.

La généricité en Rust

Commençons par la base en donnant une description de ce que c'est : "c'est une fonctionnalité qui autorise le polymorphisme paramétrique" (ou juste polymorphisme pour aller plus vite). Pour faire simple, ça permet de manipuler des objets différents du moment qu'ils implémentent le ou les traits requis.

Par exemple, on pourrait manipuler un chien robot, il implémenterait le trait **Machine** et le trait **Animal** :

Run the following code:
```
trait Machine {
    fn get_nombre_de_vis(&self) -> u32;
    fn get_numero_de_serie(&self) -> &str;
}

trait Animal {
    fn get_nom(&self) -> &str;
    fn get_nombre_de_pattes(&self) -> u32;
}

struct ChienRobot {
    nom: String,
    nombre_de_pattes: u32,
    numero_de_serie: String,
}

impl Animal for ChienRobot {
    fn get_nom(&self) -> &str {
        &self.nom
    }

    fn get_nombre_de_pattes(&self) -> u32 {
        self.nombre_de_pattes
    }
}

impl Machine for ChienRobot {
    fn get_nombre_de_vis(&self) -> u32 {
        40123
    }

    fn get_numero_de_serie(&self) -> &str {
        &self.numero_de_serie
    }
}
```

Ainsi, il nous est désormais possible de faire :

Run the following code:
```
fn presentation_animal<T: Animal>(animal: T) {
    println!(
        "Il s'appelle {} et il a {} patte()s !",
        animal.get_nom(),
        animal.get_nombre_de_pattes(),
    );
}

let super_chien = ChienRobot {
    nom: "Super chien".to_owned(),
    nombre_de_pattes: 4,
    numero_de_serie: String::from("super chien DZ442"),
};

presentation_animal(super_chien);
```

Mais comme c'est aussi une machine, on peut aussi faire :

Run the following code:
```
fn description_machine<T: Machine>(machine: T) {
    println!(
        "Le modèle {} a {} vis",
        machine.get_numero_de_serie(),
        machine.get_nombre_de_vis(),
    );
}
```

Revenons-en maintenant à notre problème initial : "comment faire avec 40 espèces d'animaux différentes" ? Je pense que vous commencez à voir où je veux en venir je présume ? Non ? Très bien, dans ce cas prenons un autre exemple :

Run the following code:

```
trait Animal {
    fn get_nom(&self) -> &str {
        &self.nom
    }

    fn get_nombre_de_pattes(&self) -> u32 {
        self.nombre_de_pattes
    }
}

struct Chien {
    nom: String,
    nombre_de_pattes: u32,
}

struct Chat {
    nom: String,
    nombre_de_pattes: u32,
}

struct Oiseau {
    nom: String,
    nombre_de_pattes: u32,
}

struct Araignee {
    nom: String,
    nombre_de_pattes: u32,
}

impl Animal for Chien {}
impl Animal for Chat {}
impl Animal for Oiseau {}
impl Animal for Araignee {}

fn affiche_animal<T: Animal>(animal: T) {
    println!(
        "Cet animal s'appelle {} et il a {} patte(s)",
        animal.get_nom(),
        animal.get_nombre_de_pattes(),
    );
}

let chat = Chat { nom: String::from("Félix"), nombre_de_pattes: 4 };
let spider = Araignee { nom: String::from("Yuuuurk"), nombre_de_pattes: 8 };

affiche_animal(chat);
affiche_animal(spider);
```

Et pourtant... Ce code ne compile pas !

C'est parce qu'une implémentation par défaut d'une méthode n'aura accès qu'à ce qui est fourni par le trait lui-même. Dans le cas présent, `self.nom` et `self.nombre_de_pattes` ne sont pas définis dans le trait et ne peuvent pas donc être utilisés. Cependant, si le trait fournissait des méthodes `nombre_de_pattes()` et`nom()`, on pourrait les appeler.

Voici un code fonctionnant pour ce cas :

Run the following code:

```
struct Chien {
    nom: String,
    nombre_de_pattes: u32,
}

struct Chat {
    nom: String,
    nombre_de_pattes: u32,
}

trait Animal {
    fn get_nom(&self) -> &str;
    fn get_nombre_de_pattes(&self) -> u32;
    fn affiche(&self) {
        println!(
            "Je suis un animal qui s'appelle {} et j'ai {} pattes !",
            self.get_nom(),
            self.get_nombre_de_pattes(),
        );
    }
}

// On implémente les méthodes prévues dans le trait Animal, sauf celles par
// défaut.
impl Animal for Chien {
    fn get_nom(&self) -> &str {
        &self.nom
    }

    fn get_nombre_de_pattes(&self) -> u32 {
        self.nombre_de_pattes
    }
}

// On fait de même, mais on a quand même envie de surcharger la méthode par
// défaut...
impl Animal for Chat {
    fn get_nom(&self) -> &str {
        &self.nom
    }

    fn get_nombre_de_pattes(&self) -> u32 {
        self.nombre_de_pattes
    }

    // On peut même 'surcharger' une méthode par défaut dans le trait - il
    // suffit de la réimplémenter
    fn affiche(&self) {
        println!(
            "Je suis un animal - un chat même qui s'appelle {} !",
            self.get_nom(),
        );
    }
}

fn main() {
    fn affiche_animal<T: Animal>(animal: T) {
        animal.affiche();
    }

    let chat = Chat { nom: "Félix".to_owned(), nombre_de_pattes: 4};
    let chien = Chien { nom: "Rufus".to_owned(), nombre_de_pattes: 4};

    affiche_animal(chat);
    affiche_animal(chien);
}
```

La seule contrainte étant que, même si l'implémentation des méthodes est la même, il faudra la réimplémenter pour chaque type implémentant ce trait... Cela dit, les macros pourraient grandement faciliter cette étape répétitive et laborieuse, mais nous verrons cela plus tard.

Combinaisons de traits

Il est possible de demander à ce qu'un type générique implémente plus d'un trait. On peut les combiner en utilisant le signe `+`. Cela permettra d'avoir accès aux méthodes fournis par tous les traits qui sont requis :

Run the following code:
```
// On implémente `Debug` sur `Cat` avec `#[derive()]`:
#[derive(Debug)]
struct Chat {
    nom: String,
    nombre_de_pattes: u32,
}

fn affiche_animal<T: Animal + Debug>(animal: T) {
    // On utilise `Debug` avec `{:?}`.
    println!("Affichage de {:?}", animal);
    // On utilise `Animal` avec `.affiche()`.
    animal.affiche();
}

fn main() {
    let chat = Chat { nom: "Félix".to_owned(), nombre_de_pattes: 4 };
    affiche_animal(chat);
}
```

Dans l'exemple ci-dessus, comme le type `Chat` implémente bien les traits `Animal` et `Debug`, on peut l'utiliser comme argument dans la fonction `affiche_animal`.

Where

Il est aussi possible d'écrire un type/une fonction générique en utilisant le mot-clé **where** :

Run the following code:
```
fn affiche_animal<T>(animal: T)
where
    T: Animal
{
    println!(
        "Cet animal s'appelle {} et il a {} patte(s)",
        animal.get_nom(),
        animal.get_nombre_de_pattes(),
    );
}
```

Dans l'exemple précédent, cela n'apporte strictement rien. Cependant, **where** est plus lisible sur les fonctions/types prenant beaucoup de paramètres génériques :

Run the following code:
```
fn affiche_2_animaux<T, T2>(animal1: T, animal2: T2)
where
    T: Animal + Debug,
    T2: Animal + Debug + Clone
{
    // ...
}
```

5. Propriété (ou ownership)

Jusqu'à présent, de temps à autre, on utilisait le caractère '&' devant des paramètres de fonctions sans que je vous explique à quoi ça servait. Exemple :

```
Run the following code:
fn ajouter_valeur(v: &mut Vec<i32>, valeur: i32) {
    v.push(valeur);
}

struct X {
    v: i32,
}

impl X {
    fn addition(&self, a: i32) -> i32 {
        self.v + a
    }
}
```

Il s'agit de variables passées par référence. En Rust, cela a une grande importance. Il faut savoir que chaque variable ne peut avoir qu'un seul "propriétaire" à la fois, ce qui est l'une des grandes forces de ce langage. Par exemple :

```
Run the following code:
fn une_fonction(v: Vec<i32>) {
    // le contenu n'a pas d'importance
}

let v = vec![5, 12];

une_fonction(v);
println!("{}", v[0]); // error ! "use of moved value"
```

Un autre exemple encore plus simple :

```
Run the following code:
let original = vec![1, 2, 3];
let non_original = original;

println!("original[0] is: {}", original[0]); // même erreur
```

"Mais c'est complètement idiot ! Comment on fait pour modifier la variable depuis plusieurs endroits ?!"

C'est justement pour éviter ça que ce système d'ownership (propriété donc) existe. C'est ce qui vous posera sans aucun doute le plus de problème quand vous développerez vos premiers programmes.

Dans un chapitre précédent, je vous ai parlé des traits. Hé bien sachez que l'un d'entre eux s'appelle **Copy** et permet de copier (sans rire !) un type sans perdre la propriété de l'original. Tous les types de "base" (aussi appelés **primitifs**) (**i8**, **i16**, **i32**, **isize**, **f32**, etc...) l'implémentent. Ce code est donc tout à fait valide :

```
Run the following code:
let original: i32 = 8;
let copy = original;

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

Cependant **Copy** ne peut être implémenté que sur des types primitifs ou des structures ne contenant que des types primitifs, ce qui nous limite beaucoup. Un autre trait appelé **Clone** permet lui de dupliquer des types "plus lourds". Ce n'est cependant pas toujours une bonne idée de dupliquer un type. Revenons donc à notre situation initiale.

Il est possible de "contourner" ce problème de copie de la manière suivante :

```
Run the following code:
fn fonction(v: Vec<i32>) -> Vec<i32> {
    v // on "rend" la propriété de l'objet en le renvoyant
}

fn main() {
    let v = vec![5, 12];

    let v = fonction(v); // et on la re-récupère ici
    println!("{}", v[0]);
}
```

Bof, n'est-ce pas ? Et encore c'est un code simple. Imaginez quelque chose comme ça :

```
Run the following code:
fn fonction(
    v1: Vec<i32>,
    v2: Vec<i32>,
    v3: Vec<i32>,
    v4: Vec<i32>,
) -> (Vec<i32>, Vec<i32>, Vec<i32>, Vec<i32>) {
    (v1, v2, v3, v4)
}

let v1 = vec![5, 12, 3];
let v2 = vec![5, 12, 3];
let v3 = vec![5, 12, 3];
let v4 = vec![5, 12, 3];

let (v1, v2, v3, v4) = fonction(v1, v2, v3, v4);
```

Ça devient difficile de suivre, hein ? Vous l'aurez donc compris, ce n'est pas du tout une bonne idée.

"Mais alors comment on fait ? On implémente le trait **Clone** sur tous les types ?"

Non, et heureusement ! La copie de certains types pourrait avoir un lourd impact sur les performances de votre programme, tandis que d'autres ne peuvent tout simplement pas être copiés ! C'est ici que les **références** rentrent en jeu.

Jusqu'à présent, vous vous en êtes servies sans que je vous explique à quoi elles servaient. Je pense que maintenant vous vous en doutez. Ajoutons une référence à notre premier exemple :

```
Run the following code:
fn une_fonction(v: &Vec<i32>) {
    // le contenu n'a pas d'importance
}

let v = vec![5, 12];

une_fonction(&v);
println!("{}", v[0]); // Pas de souci !
```

On peut donc dire que les références permettent **d'emprunter** une variable **sans en prendre la propriété**, et c'est très important de s'en souvenir !

Prenons un exemple : quand vous indiquez à quelqu'un où vous vivez, vous n'allez pas copier votre maison/appartement mais juste donner son adresse. Hé bien ici, c'est la même chose !

Tout comme les variables, les références aussi peuvent être mutables. "&" signifie référence constante et "&mut" signifie référence mutable. Il y a cependant plusieurs choses à savoir :

- Une référence ne peut pas "vivre" plus longtemps que la variable qu'elle référence.
- On peut avoir autant de référence constante que l'on veut sur une variable.
- On ne peut avoir **qu'une seule** référence mutable sur une variable.
- On ne peut avoir une référence mutable que sur une variable mutable.
- On ne peut avoir une référence constante et une référence mutable en même temps sur une variable.

Pour bien comprendre cela, il faut bien avoir en tête comment la durée de vie d'une variable fonctionne :

Run the following code:
```
fn func() {
    // On crée une variable.
    let mut var = 10i32;

    // On fait des opérations dessus.
    var += 12;
    var *= 2;
    // ...

    // Quand on sort de la fonction, var n'existe plus.
}

fn main() {
    // Cette variable n'a rien à voir avec celle dans la fonction func.
    let var: i32 = 12;
    let var2: f32 = 0;

    func();
    // On quitte la fonction, var et var2 n'existent plus.
}
```

Ainsi, ce code devient invalide :

Run the following code:
```
fn main() {
    let reference: &i32;
    {
        let x = 5;
        reference = &x;
    } // `x` n'existe plus ici, rendant `reference` invalide

    println!("{}", reference); // On ne peut donc pas s'en servir ici.
}
```

Ici, le compilateur vous dira que la variable **x** ne vit pas assez longtemps, elle est donc détruite en premier, rendant **reference** invalide ! Pour pallier à ce problème, rien de bien compliqué :

Run the following code:
```
fn main() {
    let x = 5;
    let reference: &i32 = &x;

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

Maintenant vous savez ce qui se cache derrière les références et vous avez des notions concernant la durée de vie des variables. Il est temps de voir ce deuxième point un peu plus en détail.

Pour les plus curieux : toutes ses règles sont appliquées par ce que l'on appelle le "borrow checker" (le "vérifieur d'emprunt" en français) dans le compilateur de Rust.

6. Durée de vie (ou lifetime)

Il existe deux types de durée de vie :

- Les durées de vie statiques.
- Les durées de vie temporaires.

Les durées de vie statiques

Les durées de vie statiques permettent d'avoir des références sur des variables statiques ou du contenu "constant" :

```
Run the following code:
// Avec une variable statique :
static VAR: i32 = 0;
let variable_statique: &'static i32 = &VAR;

// Avec une constante :
const CONST_VAR: i32 = 0;
let variable_constante: &'static i32 = &CONST_VAR;

// Avec du contenu constant (car une string écrite "en dur" dans le code est
// stockée telle quelle dans le code compilé) :
let variable_const: &'static str = "Ceci est une str constante !";
```

Une durée de vie statique veut donc dire que le contenu qu'elle référence vivra du début à la fin du programme.

Les durées de vie temporaires

Les durées de vie temporaires sont un peu plus complexes mais aussi moins visibles la plupart du temps. Imaginons que l'on écrive une structure dont l'un des champs devait être modifié à l'extérieur de la structure. On se contenterait de renvoyer `&mut self.ma_variable`. Bien que ce code fonctionne, il est important de comprendre ce qu'il se passe :

```
Run the following code:
struct MaStruct {
    variable: String,
}

impl MaStruct {
    fn get_variable(&mut self) -> &mut String {
        &mut self.variable
    }
}

fn main() {
    let mut v = MaStruct { variable: String::new() };

    v.get_variable().push_str("hoho !");
    println!("{}", v.get_variable());
}
```

La méthode `get_variable` va en fait renvoyer une référence **temporaire** sur **self.variable**. Si on voulait écrire ce code de manière "complète", on l'écrirait comme ceci :

```
Run the following code:
impl MaStruct {
    fn get_variable<'a>(&'a mut self) -> &'a mut String {
        &mut self.variable
    }
}
```

'a représente la durée de vie (cela aurait tout aussi bien pu être **'x** ou **'zaza**, peu importe). Ici, on retourne donc une référence

avec une durée de vie **'a** sur une variable.

Ici le compilateur fait ce que l'on appelle de **l'élision**. Comme il n'y a qu'une seule durée de vie possible pour cette variable, il la déduit automatiquement donc pas besoin de l'ajouter nous-mêmes. Cependant il y a beaucoup de cas où il ne peut pas, comme par exemple :

Run the following code:
```
fn foo(a: &str, b: &str) -> &str {
    a
}

fn main() {
    let c = foo("a", "b");
}
```

Ce code renvoie cette erreur :

```
1 | fn foo(a: &str, b: &str) -> &str {
  |           ----     ----     ^ expected named lifetime parameter
```

Dans le cas présent, il y a plusieurs durées de vie possibles et il ne sait pas laquelle choisir, il faut donc ajouter les durées de vie nous-même :

Run the following code:
```
fn foo<'a, 'b>(a: &'a str, b: &'b str) -> &'a str {
    a
}

fn main() {
    let c = foo("a", "b");
}
```

Types avec une référence comme champ

Les itérateurs sont un exemple assez courant où un type contient un champ qui est une référence. Pour l'illustrer, on va écrire un itérateur sur une `String` qui renvoie chaque ligne non vide :

Run the following code:

```
struct LineIterator<'a> {
    content: &'a str,
}

// Comme le type `LineIterator` contient une durée de vie, il faut aussi la
// déclarer sur tous les impl blocs.
impl<'a> LineIterator<'a> {
    fn new(content: &'a str) -> LineIterator<'a> {
        LineIterator { content }
    }

    fn retourne_substring(
        &mut self,
        début: usize,
        dernier: usize,
    ) -> Option<&'a str> {
        if dernier <= début {
            // Si jamais la string est vide, cela signifie que l'on a atteint
            // la fin de notre string donc qu'il n'y a plus rien à retourner.
            return None;
        }
        // On récupère la sous-string que l'on va retourner.
        let ret = &self.content[début..dernier];
        // On change la position du début de notre string.
        self.content = &self.content[dernier..];
        Some(ret)
    }
}

// On implémente le trait `Iterator` par commodité.
impl<'a> Iterator for LineIterator<'a> {
    type Item = &'a str;

    fn next(&mut self) -> Option<Self::Item> {
        let mut indices = self.content.char_indices();
        let mut début = 0;

        // D'abord on passe tous les retours à la ligne pour arriver au contenu.
        while let Some((pos, c)) = indices.next() {
            if c != '\n' {
                début = pos;
                break;
            }
        }
        while let Some((pos, c)) = indices.next() {
            if c == '\n' {
                // On a trouvé un retour à la ligne donc on renvoie ce qu'on a
                // trouvé.
                return self.retourne_substring(début, pos);
            }
        }
        // Nous avons atteint la fin de notre string, on renvoie tout le
        // contenu.
        self.retourne_substring(début, self.content.len() - 1)
    }
}

fn main() {
    // On crée notre itérateur.
    let iterator = LineIterator::new("a\n\nbc\n");
    // On récupère toutes les `String`s dans un vecteur.
    let strings = iterator.into_iter().collect::<Vec<_>>();
    // Si tout s'est bien passé, cet `assert_eq` ne devrait pas paniquer.
    assert_eq!(strings, vec!["a", "bc"]);
}
```

Il est bon de noter que nous aurions pu remplacer la durée de vie (`'a`) du champ `content` par `'static`. Cependant, faire cela nous aurait empêcher d'utiliser autre chose que des `str` statiques, ce qui aurait été une grosse limitation.

Un autre cas d'usage assez répandu pour l'utilisation des références directement dans un type est pour les **parseurs**. Le plus

souvent, vous n'avez pas besoin de prendre la propriété de la donnée que vous souhaitez parser. Cela offre le plus souvent la possibilité d'éviter des allocations qui ne sont pas nécessaires. Dans l'exemple que l'on vient de voir, il n'y a aucune allocation pour les `str` puisqu'on ne renvoie que des "vues" sur un espace mémoire. Si vous avez besoin de modifier ce contenu, vous pouvez toujours le faire de votre côté en allouant la mémoire nécessaire.

Contraintes sur les durées de vie

Tout comme on peut ajouter des contraintes sur les traits avec les **supertraits**, on peut aussi ajouter des contraintes sur les durées de vie :

Run the following code:
```
fn foo<'a, 'b: 'a>(a: &'a str, b: &'b str) -> &'a str {
    a
}
```

Ici, on indique au compilateur que la durée de vie `'b` doit vivre **au moins aussi longtemps** que `'a`. Cela reste cependant une utilisation avancée des durées de vie et il y a peu de chances que vous en croisiez, mais il semblait important que vous soyiez au courant au cas où vous veniez à en rencontrer.

D'ailleurs, tout comme pour les arguments génériques, il est possible d'utiliser le mot-clé **where** pour améliorer la lisibilité des durées de vie :

Run the following code:
```
fn foo<'a, 'b, 'c>(a: &'a str, b: &'b str, c: &'c str) -> &'c str
where
    'b: 'a,
    'c: 'b + 'c,
{
    c
}
```

7. Déréférencement

Après les gros chapitres précédents, celui-là ne devrait pas vous prendre beaucoup de temps. Il vous arrivera de croiser ce genre de code :

```
Run the following code:
fn une_fonction(x: &mut i32) {
    *x = 2; // on déréférence
}

fn main() {
    let mut x = 0;

    println!("avant : {}", x);
    une_fonction(&mut x);
    println!("après : {}", x);
}
```

La valeur a donc été modifiée dans la fonction `une_fonction`. Pour ceux ayant fait du C/C++, c'est exactement la même chose que le déréférencement d'un pointeur. La seule différence est que cela passe par les traits **Deref** et **DerefMut** en **Rust** .

Là où ça devient intéressant c'est que ces traits sont implémentés par "&" et "&mut". Donc "&" implémente **Deref** tandis que "&mut" implémente à la fois **Deref** et **DerefMut**. Ce qui permet de faire `*x = 2` dans l'exemple précédent. Cependant, il est aussi possible de faire :

```
Run the following code:
let x = String::new();
// On déréférence &String en &str.
let deref_x: &str = &*x;
```

Il est donc possible de déréférencer un objet en implémentant ce trait.

Implémentation

On va prendre un exemple pour que vous compreniez le tout plus facilement :

```
Run the following code:
// On importe le trait.
use std::ops::Deref;

struct UneStruct {
    value: u32
}

impl Deref for UneStruct {
    // Pour préciser quel type on retourne en déréférençant.
    type Target = u32;

    fn deref(&self) -> &u32 {
        &self.value
    }
}

fn main() {
    let x = UneStruct { value: 0 };
    assert_eq!(0u32, *x); // on peut maintenant déréférencer x
}
```

Je pense que le code est suffisamment explicite pour se passer d'explications supplémentaires.

Auto-déréférencement

Vous utilisez cette fonctionnalité sans le savoir lorsque vous faites :

```
Run the following code:
// On a donc une String("toto").
let x = "toto".to_owned();
// On passe une &String comme argument à la fonction.
affiche_la_str(&x);

// On obtient une &str.
fn affiche_la_str(s: &str) {
    println!("affichage : {}", s);
}
```

La question étant : "Pourquoi si on passe `&String` on obtient `&str` ?". Sachez que Rust implémente un système **d'auto-déréférencement** (basé sur le trait **Deref** bien évidemment, rien de magique). Cela permet d'écrire des codes de ce genre :

```
Run the following code:
struct UneStruct;

impl UneStruct {
    fn foo(&self) {
        println!("UneStruct");
    }
}

let f = UneStruct;

f.foo();
(&f).foo();
(&&f).foo();
(&&&&&&&&f).foo();
```

Le compilateur va déréférencer jusqu'à obtenir le type voulu (en l'occurrence, celui qui implémente la méthode `foo` dans le cas présent, donc `UneStruct`) ou jusqu'à renvoyer une erreur. Je pense que certains d'entre vous ont compris où je voulais en venir concernant **String**.

Le compilateur voit qu'on envoie `&String` dans une méthode qui reçoit `&str` comme paramètre. Il va donc déréférencer **String** pour obtenir `&str`. Nous obtenons donc `&str`. On peut imager ce que fait le compilateur de cette façon : `&(*(String.deref()))`.

Pour ceux que ça intéresse, voici comment fait le compilateur, étape par étape :

- `&String` -> pas `&str`, on déréférence `String`
- `&(*String)` -> Le type `String` implémente le trait **Deref**, on appelle donc ce trait sur notre type.
- `&(*(String.deref()))`
- `&(*(&str))`
- `&(str)`
- `&str`

Et voilà, le compilateur a bien le type attendu !

8. Sized et String vs str

Ce chapitre approfondit ce dont nous avons déjà vu dans le chapitre sur les variables et plus particulièrement les **slices**, à savoir : la différence entre **String** et **str**. Ou encore : "Pourquoi deux types pour représenter la même chose ?". Tâchons d'y répondre !

str

Le type **str** représente tout simplement une adresse mémoire et une taille. C'est pourquoi on ne peut modifier son contenu. Mais ce n'est pas la seule chose à savoir à son sujet. Commençons par regarder le code suivant :

```
Run the following code:
let x = "str";
```

`x` est donc une variable de type `&str`. Mais que se passe-t-il si nous tentons de déréférencer `x` pour obtenir un type **str** ?

```
Run the following code:
let x = *"str";
```

Ce qui donnera :

```
error: the trait `core::marker::Sized` is not implemented for the type `str` [E0277]
```

Mais quel est donc ce trait **Sized**, et pourquoi ça pose un problème que **str** ne l'implémente pas ?

Le trait Sized

str n'est pas le seul type qui n'implémente pas le trait **Sized**. Les **slice** non plus ne l'implémentent pas :

```
Run the following code:
fn fonction(x: [u32]) {
    // ...
}
```

Ce qui donne :

```
error[E0277]: the size for values of type `[u32]` cannot be known at compilation time
 --> src/main.rs:1:8
  |
1 | fn foo(x: [u32]) {
  |        ^ doesn't have a size known at compile-time
  |
  = help: the trait `Sized` is not implemented for `[u32]`
```

Le problème est donc que si le trait **Sized** n'est pas implémenté sur le type, cela signifie que l'on ne peut pas connaître sa taille au moment de la compilation car on ne sait pas combien d'éléments le type contiendra et donc quelle taille en mémoire il occupera. Par conséquent, nous sommes obligés de passer par d'autres types pour les manipuler. Dans le cas des **str** et des **slice**, on peut se contenter d'utiliser des références qui ont une taille connue au moment de la compilation :

```
Run the following code:
fn fonction(x: &[u32], s: &str) {
    // ...
}
```

Maintenant revenons-en aux **String** et aux **str**.

String

Les **String** permettent donc de manipuler des chaînes de caractères. En plus de ce que contient **str** (à savoir : une adresse mémoire et une taille), elles contiennent aussi une capacité qui représente la quantité de mémoire réservée (mais pas nécessairement utilisée).

Pour résumer un peu le tout, **str** est une vue mémoire de taille constante tandis que **String** est une structure permettant de manipuler des chaînes de caractères (et donc d'en changer la taille au besoin) et qui peut être déréférencée en **str**. C'est d'ailleurs pour ça qu'il est très simple de passer de l'un à l'autre :

```
Run the following code:
let x: &str = "a";
// On pourrait aussi utiliser `String::from` ou `str::into`.
let y: String = x.to_owned();
let z: &str = &y;
```

Vec vs slice

C'est plus ou moins le même fonctionnement : une **slice** est une vue mémoire de taille constant tandis que le type **Vec** permet de manipuler une "vue mémoire" (et notamment d'en modifier la taille). En rentrant dans les détails plus techniques, voyez cela comme un pointeur qui pointerait vers une zone mémoire dont la taille serait réallouée au besoin. Exemple :

```
Run the following code:
let x: &[i32] = &[0, 1, 2];
let y: Vec<i32> = x.to_vec();
let z: &[i32] = &y;
```

Le type **String** n'est d'ailleurs qu'un wrapper sur un **Vec<u8>** qu'elle utilise pour manipuler les chaînes de caractères. C'est d'ailleurs pour ça qu'il est possible de créer une **String** à partir d'un **Vec<u8>** (avec la méthode **String::from_utf8** notamment).

Ce chapitre (et notamment le trait **Sized**) est particulièrement important pour bien comprendre les mécanismes sous-jacents de **Rust**. Soyez bien sûr d'avoir tout compris avant de passer à la suite !

9. Unsafe

Le code Rust que l'on a vu jusque là est **sûr** (sound/safe) : il ne peut pas causer de comportement non-défini (undefined behaviour). Cependant, il est possible que vous ayez besoin d'écrire du code dont la sûreté ne peut pas être assurée par le compilateur de Rust. Par exemple si vous utilisez une bibliothèque écrite dans un autre langage.

Il est cependant important de noter que même dans un bloc **unsafe**, les règles d'emprunts et de propriétés sur les variables sont exactement les même ! **unsafe** n'est donc pas un mot-clé magique qui permet d'ignorer les règles de Rust.

Voici la liste des cas où le mot-clé **unsafe** doit être utilisé :

- Déréférencer un pointeur.
- Implémenter un trait défini comme **unsafe**.
- Appeler une fonction définie comme **unsafe**.
- Modifier la valeur d'une variable statique.
- Accéder aux champs d'une **union** (on revient sur ce type dans le livre juste après).

Si vous tentez de faire une de ces opérations en dehors d'un bloc **unsafe**, la compilation échouera en indiquant qu'il faut que ce code est **unsafe**. Par exemple ce code :

```
Run the following code:
fn main() {
    let x = 0u32;
    let y = &x as *const u32;

    println!("{}", *y);
}
```

donnera cette erreur :

```
error[E0133]: dereference of raw pointer is unsafe and requires unsafe function or block
 --> src/main.rs:5:20
  |
6 |     println!("{}", *y);
  |                     ^^ dereference of raw pointer
```

Le mot-clé **unsafe** a donc deux utilités :

1. Il indique que le compilateur ne peut pas s'assurer que ce code ne contient pas de comportement non-défini, et donc que cette responsabilité revient au développeur.
2. Il permet au développeur de rapidement voir que ce code a sans doute besoin de plus d'attention que le reste car il risque d'avoir des comportements non-définis qu'il faudra vérifier soi-même.

Donc le code précédent doit être écrit ainsi :

```
Run the following code:
fn main() {
    let x = 0u32;
    let y = &x as *const u32;

    unsafe {
        println!("{}", *y);
    }
}
```

On déréférence maintenant `y` dans un bloc **unsafe**.

Dernier point : toutes les utilisations de **unsafe** n'ont pas le même sens. On va donc voir ce que chacune signifie.

Blocs unsafe

Les blocs **unsafe** permettent de déréférencer des pointeurs, mais aussi d'appeler des fonctions/méthodes **unsafe**, comme vu dans l'exemple précédent.

Ils servent aussi de marqueurs visuels pour nous permettre de voir quel code a besoin de plus d'attention car c'est au développeur de s'assurer que le code n'aura pas de comportement non-défini.

Fonctions/méthodes unsafe

Les fonctions et méthodes **unsafe** peuvent avoir un comportement non-défini dans certains contextes et/ou selon les arguments qu'elles reçoivent. On définit une fonction **unsafe** de cette façon :

```
Run the following code:
unsafe fn fonction() {
    // code
}
```

Un bon exemple est la méthode `slice::get unchecked` : elle retourne la valeur à l'index donné sans vérifier si cet index est bien inclus dans la slice. Donc si on lui donne un index en dehors de ces limites, le comportement sera non-défini. Cela peut causer une erreur de segmentation (segmentation fault) entraînant le plantage du programme ou bien juste renvoyer une valeur dans la mémoire se trouvant à cet emplacement. C'est donc pour cela qu'elle est définie

Bien qu'il ne soit pas obligatoire d'ajouter des blocs **unsafe** dans une fonction définie comme **unsafe** pour pouvoir faire des opérations **unsafe**, il est cependant recommandé de quand même en ajouter un pour améliorer la lisibilité du code :

```
Run the following code:
unsafe fn fonction() -> u32 {
    let x = 12u32;
    let y = &x as *const u32;

    // Cela permet de voir quelle partie de la fonction a besoin d'être unsafe.
    unsafe {
        *y
    }
}
```

Traits unsafe

Un trait **unsafe** est un trait avec des prérequis qui ne peuvent être vérifiés par le compilateur lorsqu'il est implémenté sur un type. Ce sera donc au développeur de s'assurer que l'implémentation respecte bien ces conditions.

On peut déclarer un trait **unsafe** comme ceci :

```
Run the following code:
unsafe trait UnsafeTrait {
    // Les éléments du trait.
}
```

L'implémentation d'un trait défini comme **unsafe** utilise aussi ce mot-clé :

```
Run the following code:
struct Structure;

unsafe impl UnsafeTrait for Structure {
    // Implémentation des éléments du trait.
}
```

Un bon exemple sont les traits `Send` et `Sync` qui permettent respectivement de d'indiquer qu'un type peut être transféré dans un autre thread et que la référence d'un type peut être partagée dans un autre thread. Nous reviendrons plus en détail sur ces 2 traits et sur le multi-threading plus tard dans ce livre.

Les blocs externes

Si vous voulez utiliser une bibliothèque codée en langage C, il vous faudra définir les fonctions de cette bibliothèque que vous voulez utiliser. Par exemple si on veut utiliser la fonction `puts` de la bibliothèque standard du langage C qui est définie comme ceci :

```
int puts(const char *s);
```

On va donc écrire ce code en Rust :

```
Run the following code:
unsafe extern "C" {
    fn puts(s: *const i8) -> i32;
}

// Qu'on appellera comme ceci :
fn main() {
    unsafe {
        puts(b"bonjour\n\0".as_ptr() as *const _);
    }
}
```

Veuillez noter que ce code est **incorrect** car les types `char` et `int` ne correspondent pas nécessairement à un entier signé de 8 bits et à un entier de 32 bits selon la plateforme. Ne l'utilisez donc surtout pas ! Nous reviendrons sur comment utiliser **correctement** une bibliothèque C dans la troisième partie de ce livre.

On doit s'assurer que les éléments que l'on importe ont la bonne signature car si ce n'est pas le cas, cela conduira à des comportements non-définis.

Dernier point : il n'est pas obligatoire de définir un bloc externe comme **unsafe**, cependant je considère que cela rend plus évident que ce code a des risques très élevés de conduire à des comportements non-définis. De plus, que l'on définisse un bloc externe comme **unsafe** ou non, les éléments qui sont définis dedans sont considérés comme **unsafe** par le compilateur quoi qu'il arrive.

10. Les unions

Les **unions** ressemblent beaucoup aux **structures** tout en étant très différentes : tous les champs d'une **union** partagent le même espace mémoire. Si la valeur d'un champ d'une union est changé, cela peut écrire par-dessus un autre champ. Autre information importante : la taille d'une union est la taille de son champ avec la plus grande taille.

Bien évidemment, vous vous doutez bien qu'avec toutes ces restrictions, les types des champs d'une **union** doivent suivre certaines règles : ils doivent implémenter le trait `Copy` ou bien être wrappés dans le type `ManuallyDrop`.

Chaque accès à un champ d'une **union** est considéré comme **unsafe** et vous ne pourrez pas faire des emprunts mutable sur plus d'un champ à la fois car ils sont considérés comme faisant tous parties du même espace mémoire.

La plupart des **derive traits** ne peuvent pas être utilisés non plus (par exemple `#[derive(Debug)]`. Cela ne veut pas dire qu'une **union** ne peut pas implémenter ces traits, juste qu'il vous faudra les implémenter vous-même.

Enfin, dernier point : quand on instancie une **union**, on ne doit spécifier **qu'un seul** champ.

Mise en pratique

Prenons un exemple :

Run the following code:
```
union Foo {
    a: u16,
    b: u8,
}

let f = Foo { a: 1 };
unsafe { // Nécessaire pour pouvoir accéder aux champs.
    println!("a: {} b: {}", f.a, f.b);
}
```

Ce qui affichera :

```
a: 1 b: 1
```

Et oui, souvenez-vous : les champs partagent le même espace mémoire. Par contre que se passe-t-il pour le champ `b` si on assigne au champ `a` une valeur plus grande que ce que peut contenir un `u8` ?

Run the following code:
```
let f = Foo { a: u16::MAX };
unsafe {
    println!("a: {} b: {}", f.a, f.b);
}
```

Ce qui affichera :

```
a: 65535 b: 255
```

Donc `b` représente la partie "basse" de `a`. Ce qui illustre parfaitement l'espace mémoire partagé.

Que se passe-t-il si on change l'ordre des types et que l'on commence par le `u8` à la place du `u16` ?

Run the following code:
```
union Foo {
    a: u8,
    b: u16,
}

// Ce sera maintenant le champ `b` qu'on va initialiser.
let f = Foo { b: u16::MAX };
unsafe {
    println!("a: {} b: {}", f.a, f.b);
}
```

Ce qui affichera :

```
a: 255 b: 65535
```

Donc rien n'a changé, le `u8` représente toujours la partie "basse" du `u16`. Et que se passe-t-il si on ajoute un autre champ de type `u8` ?

Run the following code:
```
union Foo {
    a: u16,
    b: u8,
    c: u8,
}

let f = Foo { a: 10 };
unsafe {
    println!("a: {} b: {} c: {}", f.a, f.b, f.c);
}
```

Ce qui affichera :

```
a: 10 b: 10 c: 10
```

Donc un type plus petit représentera toujours la partie basse d'un type plus grand, même s'il y en a plusieurs.

Regardons maintenant un exemple un peu concret : manipuler une couleur. Une couleur est composée de 4 valeurs :

- rouge
- vert
- bleu
- transparence

Chacune de ces valeurs peut aller de 0 à 255 inclus (un `u8` donc). Cependant, il est assez fréquent de vouloir passer un `u32` pour représenter une couleur plutôt que chaque composant. Les `unions` sont donc un excellent moyen de faire ça :

Run the following code:
```
#[derive(Default, Clone, Copy)]
struct Color {
    red: u8,
    green: u8,
    blue: u8,
    alpha: u8,
}

union ColorUnion {
    color: Color,
    value: u32,
}

let mut color = ColorUnion { value: 0 };
unsafe {
    assert_eq!(color.color.green, 0);
    // Une couleur verte à moitié transparente.
    color.color.green = 255;
    color.color.alpha = 128;
    // On peut comparer la valeur avec des décalages binaires pour se faciliter la vie :
    assert_eq!(color.value, (255 << 8) + (128 << 24));
    // Ou bien directement avec la valeur du `u32`, mais plus difficile à lire :
    assert_eq!(color.value, 2_147_548_928);
}
```

Pattern matching

Maintenant regardons rapidement comment le **pattern matching** fonctionne avec une **union**. Tout comme lorsque l'on initialise une **union**, il ne faut spécifier qu'un seul champ. Et bien évidemment, un bloc **unsafe** est nécessaire pour pouvoir accéder au champ. Exemple :

Run the following code:
```
let f = Foo { a: 10 };
unsafe {
    match f {
        Foo { a: 10 } => println!("ok"),
        _ => println!("not ok"),
    }
}
```

Voilà qui conclut ce chapitre sur les **unions**.

11. Closure

Nous allons maintenant aborder un chapitre très important pour le langage **Rust**. Ceux ayant déjà utilisé des langages fonctionnels n'y verront qu'une révision (mais ça ne fait jamais de mal après tout !).

Pour ceux qui n'ont jamais utilisé de closures, on peut les définir comme des fonctions anonymes qui capturent leur environnement.

"Une fonction "anonyme" ? Elle "capture" son environnement ?"

Ne vous inquiétez pas, vous allez très vite comprendre, prenons un exemple simple :

Run the following code:
```
let multiplication = |nombre: i32, multiplicateur: i32| nombre * multiplicateur;

println!("{}", multiplication(2, 2));
```

Pour le moment, vous vous dites sans doute qu'en fait, ce n'est qu'une fonction. Maintenant ajoutons un élément :

Run the following code:
```
let nombre = 2i32;
let multiplication = |multiplicateur: i32| nombre * multiplicateur;

println!("{}", multiplication(2));
```

Là je pense que vous vous demandez comment il fait pour trouver la variable **nombre** puisqu'elle n'est pas dans le scope de la "fonction". Comme je vous l'ai dit, une closure **capture** son environnement, elle a donc accès à toutes les variables présentes **dans le scope de la fonction qui la crée**.

Mais à quoi ça peut bien servir ? Imaginons que vous ayez une interface graphique et que vous souhaitez effectuer une action lorsque l'utilisateur clique sur un bouton. Cela donnerait quelque chose dans ce genre :

Run the following code:
```
let mut bouton = Bouton::new();
let mut clicked = false;

bouton.clicked(|titre| {
    clicked = true;
    println!("On a cliqué sur le bouton {} !", titre);
});
```

Très pratique pour partager des informations avec des éléments en dehors du scope de la closure sans avoir besoin d'ajouter des mécanismes qui s'en chargeraient. Les closures sont utilisées pour trier des **slice**s par exemple.

Si jamais vous souhaitez écrire une fonction recevant une closure en paramètre, voici à quoi cela va ressembler :

Run the following code:
```
fn fonction_avec_closure<F>(closure: F) -> i32
    where F: Fn(i32) -> i32
{
    closure(1)
}
```

Ici, la closure prend un **i32** comme paramètre et renvoie un **i32**. Vous remarquerez que la syntaxe est proche de celle d'une fonction générique, la seule différence venant du mot-clé **where** qui permet de définir à quoi doit ressembler la closure. À noter qu'on aurait aussi pu écrire la fonction de cette façon :

Run the following code:
```
fn fonction_avec_closure<F: Fn(i32) -> i32>(closure: F) -> i32 {
    closure(1)
}
```

Chose intéressante à noter : le trait **Fn** est implémenté sur les closures… mais aussi sur les fonctions ! Un générique qui accepte une closure acceptera aussi une fonction. Nous pourrions donc faire :

Run the following code:
```
fn fonction_avec_closure<F: Fn(i32) -> i32>(closure: F) -> i32 {
    closure(1)
}

// On définit qui correspond à la définition du générique "F" de
// "fonction_avec_closure".
fn fonction(nb: i32) -> i32 {
    nb * 2
}

// Les 2 appels font exactement la même chose.
fonction_avec_closure(|nb: i32| nb * 2);
fonction_avec_closure(fonction);
```

Nous avons maintenant vu les closures de type **Fn**. Il en existe cependant deux autres types avec chacune ses propres caractéristiques.

FnMut

Si jamais vous souhaitez avoir un accès mutable sur une variable capturée dans une closure, il vous faudra utiliser le trait **FnMut** :

Run the following code:
```
fn appelle_2_fois<F>(mut func: F)
    where F: FnMut()
{
    func();
    func();
}

let mut x: usize = 1;
// Cette closure a besoin d'un accès mutable à la variable x.
let ajoute_deux_a_x = || x += 2;
appelle_2_fois(ajoute_deux_a_x);

assert_eq!(x, 5);
```

Si jamais `appelle_2_fois` attendait une **Fn** à la place, on aurait eu l'erreur suivante :

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implement
closure is `FnMut` because it mutates the variable `x`
```

FnOnce

Voici le dernier type de closure : les closures **FnOnce**. Elles ne peuvent être appelées qu'une seule fois :

Run the following code:
```
fn utilisation<F>(func: F)
    where F: FnOnce() -> String
{
    println!("Utilisation de func : {}", func());
    // On ne peut plus utiliser "func" ici.
}

let x = String::from("x");
let return_x: FnOnce() -> String = move || x;
utilisation(return_x));
// On ne peut plus utiliser "func" ici non plus puisqu'on l'a déplacée
// dans "utilisation".
```

Une fonction qui prend FnOnce en argument apporte une information très intéressante : vous pouvez être sûr que cette closure ne sera appelé qu'une seule et unique fois. Si vous voulez faire une opération qui ne doit pas être exécutée plus d'une fois, c'est une garantie qui se révéler très utile.

Nous avons donc vu les bases des closures. C'est une partie importante, je vous conseille donc de bien vous entraîner dessus jusqu'à être sûr de bien les maîtriser !

Après ça, il est temps d'attaquer un chapitre un peu plus "tranquille".

12. Multi-fichier

Il est maintenant grand temps de voir comment faire en sorte que votre projet contienne plusieurs fichiers. Vous allez voir, c'est très facile. Imaginons que votre programme soit composé des fichiers **vue.rs** et **internet.rs**. Nous allons considérer le fichier **vue.rs** comme le fichier "principal" : c'est à partir de lui que nous allons inclure les autres fichiers. Pour ce faire :

Run the following code:
```
mod internet;

// le code de vue.rs
```

... Et c'est tout. Il n'y a rien besoin de changer dans la ligne de compilation non plus, **rustc**/**Cargo** se débrouillera pour trouver les bons fichiers tout seul. Veuillez noter que **mod** ne peut (et ne doit) être utilisé qu'une seule fois pour chaque fichier/dossier.

Si vous voulez utiliser un élément de ce fichier (on dit aussi **module**), faites tout simplement :

Run the following code:
```
internet::LaStruct {}
internet::la_fonction();
```

Si vous voulez éviter de devoir réécrire `internet::` devant chaque struct/fonction/objet venant de **internet.rs**, il vous suffit de faire comme ceci :

Run the following code:
```
// Cela veut dire que l'on inclut TOUT ce que contient ce module.
use internet::*;
// Ou comme ceci :
use internet::{LaStruct, la_fonction};

mod internet;
```

Et voilà, c'est à peu près tout ce qu'il y a besoin de savoir... Ou presque ! Si on veut utiliser un élément de **vue.rs**, on fera comme ceci :

Run the following code:
```
// internet.rs

pub use super::LaStruct; // "super" voulant dire dans "le scope supérieur".

// ou bien:

pub use crate::LaStruct; // "crate" voulant dire "le module à la racine de la crate".

// Ou bien directement dans le code:

super::LaStruct;
crate::LaStruct;
```

Fini ? Presque ! Imaginons maintenant que vous vouliez mettre des fichiers dans des sous-dossiers : dans ce cas là, il vous faudra créer un fichier **mod.rs** dans le sous-dossier dans lequel vous devrez utiliser "pub use" sur les éléments que vous voudrez réexporter dans le scope supérieur (et n'oubliez pas d'importer les fichiers avec mod !).

Maintenant disons que vous créez un sous-dossier appelé "tests", voilà comment utiliser les éléments qui y sont :

Run the following code:
```
// tests/mod.rs

pub use self::test1::Test1; // on réexporte Test1 directement
pub use self::test2::Test2; // idem

mod test1; // pour savoir dans quel fichier on cherche
mod test2; // idem
pub mod test3; // là on aura directement accès à test3

// dossier supérieur
// fichier lib.rs ou mod.rs
use tests::{Test1, Test2, test3}; // et voilà !
```

On peut résumer tout ça de la façon suivante :

- Si vous êtes à la racine du projet, vous ne pouvez importer les fichiers/modules que dans le fichier "principal" (**lib.rs** si c'est une bibliothèque ou bien **main.rs** si c'est un binaire).
- Si vous êtes dans un sous-dossier, vous ne pouvez les importer que dans le fichier **mod.rs**.
- Si vous voulez qu'un module parent ait accès aux éléments du module courant ou d'un module enfant, il faudra que ces éléments soient réexportés.

Un dernier exemple plus concret :

```
 - le_projet
    |
    |- lib.rs  <- le fichier principal
    |- un_fichier.rs
    |- module1
        |
        |- mod.rs
        |- file1.rs
        |- module2
            |
            |- mod.rs
            |- file1.rs
```

lib.rs

Run the following code:
```
// On réexporte "UnElement" de un_fichier.rs
pub use un_fichier::UnElement;

// On réexporte "UnAutreElement" de module1/file1.rs
pub use module1::file1::UnAutreElement;

// On réexporte "Element" de module1/file1.rs
pub use module1::Element;
// On aurait pu le réexporter de cette façon aussi : "pub use module1::file1::Element;"

// on réexporte "UnDernierElement" de module1/module2/file1.rs
pub use module1::module2::file1::UnDernierElement;

mod un_fichier;
mod module1;
```

un_fichier.rs

Run the following code:
```
// Vous avez besoin de le déclarer public sinon les autres modules n'y auront
// pas accès.
pub struct UnElement;
```

module1/mod.rs

```
Run the following code:
pub use file1::Element;

pub mod file1;
pub mod module2;
```

module1/file1.rs

```
Run the following code:
pub struct Element;
pub struct UnAutreElement;
```

module1/module2/mod.rs

```
Run the following code:
pub mod file1;
```

module1/module2/file1.rs

```
Run the following code:
pub struct UnDernierElement;
```

Voilà qui clôture ce chapitre. Celui qui arrive est assez dur (si ce n'est le plus dur), j'espère que vous avez bien profité de la facilité de celui-ci ! Je vous conseille de bien souffler avant car il s'agit des... macros !

13. Les macros

Nous voici enfin aux fameuses macros dont je vous ai déjà parlé plusieurs fois ! Pour rappel, une macro s'appelle des façons suivantes :

```
Run the following code:
la_macro!();
// ou bien :
la_macro![];
// ou encore :
la_macro! {};
```

Le point important ici est la présence du **!** après le nom de la macro. Nous ne parlerons ici pas des macros procédurales (proc-macros), un chapitre leur est dédié dans la troisième partie de ce cours.

Fonctionnement

Nous rentrons maintenant dans le vif du sujet : une macro est définie au travers d'une série de règles qui ressemblent à du **pattern-matching**. C'est toujours bon ? Parfait !

Une déclaration de macro se fait avec le mot-clé `macro_rules` (suivie de l'habituel "**!**"). Exemple :

```
Run the following code:
macro_rules! dire_bonjour {
    () => {
        println!("Bonjour !");
    }
}

dire_bonjour!();
```

Et on obtient :

```
Bonjour !
```

Merveilleux ! Bon jusque-là, rien de bien difficile. Mais ne vous inquiétez pas, ça arrive !

Les arguments (ou flux de tokens)

Bien évidemment, les macros peuvent recevoir des "arguments" même s'il serait plus exact de dire qu'elles reçoivent un flux de tokens :

```
Run the following code:
macro_rules! dire_quelque_chose {
    ($x:expr) => {
        println!("Il dit : '{}'", $x);
    };
}

dire_quelque_chose!("hoy !");
```

Ce qui affichera :

```
Il dit : 'hoy !'
```

Regardons un peu plus en détails le code. Le `($x:expr)` en particulier. Ici, nous avons indiqué que notre macro prenait une **expression** appelée **x** en paramètre. Après il nous a juste suffi de l'afficher.

Pour le lexique : `$x` est une **metavariable** (en un mot) tandis que `expr` est un **spécificateur de fragment**.

Maintenant on va ajouter la possibilité de passer une deuxième expression (tout en gardant la possibilité de n'en passer qu'une seule) :

Run the following code:
```
macro_rules! dire_quelque_chose {
    ($x:expr) => {
        println!("Il dit : '{}'", $x);
    };
    ($x:expr, $y:expr) => {
        println!("Il dit '{}' à {}", $x, $y);
    };
}

dire_quelque_chose!("hoy !");
dire_quelque_chose!("hoy !", "quelqu'un");
```

Et nous obtenons :

```
Il dit : 'hoy !'
Il dit 'hoy !' à quelqu'un
```

Les macros fonctionnent donc exactement de la même manière qu'un match, sauf qu'ici on "matche" sur les arguments.

Les différents spécificateurs de fragment

Comme vous vous en doutez, il y a d'autres spécificateurs de fragment en plus des **expr**. En voici la liste complète :

- **ident** : un identifiant (utilisé pour un nom de variable, de type, de fonction, etc). Exemples : `x`, `foo`.
- **path** : un nom qualifié. Exemple : `T::SpecialA`.
- **expr** : une expression. Exemples : `2 + 2`, `if true then { 1 } else { 2 }`, `f(42)`.
- **ty** : un type. Exemples : `i32`, `Vec<(char, String)>`, `&T`.
- **pat_param** : un motif (ou "pattern"). Exemples : `Some(x)` dans `if let Some(x) = Some(12)`, `(17, 'a')`, `_`.
- **pat** : plus ou moins pareil que **pat_param**. Supporte potentiellement plus de cas en fonction de l'édition de Rust.
- **stmt** : une instruction unique (ou "single statement"). Exemple : `let x = 3`.
- **block** : une séquence d'instructions délimitée par des accolades. Exemple : `{ log(error, "hi"); return 12; }`.
- **item** : un item. Exemples : `fn foo() { }`, `struct Bar;`.
- **meta** : un attribut. Exemple : `#[allow(unused_variables)]`.
- **tt** : un " token tree " contenu dans les délimiteurs `[]`, `()` ou `{}`.
- **lifetime** : Un token de durée de vie. Exemples : `'a`, `'static`.
- **vis** : un qualifieur de visibilité (qui peut être vide). Exemples : `pub`, `pub(crate)`.
- **literal** : une expression littérale. Exemples : `a"`, `'a'`, `5`.

Répétition

Les macros comme **vec!**, **print!**, **write!**, etc... permettent le passage d'un nombre "d'arguments" variable (un peu comme les `va_args` en C ou les templates variadiques en C++). Cela fonctionne de la façon suivante :

Run the following code:
```
macro_rules! vector {
    (
        $($x:expr),*
    ) => {
        [ $($x),* ].to_vec()
    }
}

let mut v: Vec<u32> = vector!(1, 2, 3);

v.push(6);
println!("{:?}", &v);
```

Ici, on dit qu'on veut une expression répétée un nombre inconnu de fois (le `$(votre_variable),*`). La virgule devant l'étoile indique le séparateur entre les arguments. On aurait aussi très bien pu mettre un `;`. D'ailleurs pourquoi ne pas essayer ?

```
Run the following code:
macro_rules! vector {
    (
        $($x:expr);*
    ) => {
        [ $($x),* ].to_vec()
    }
}

let mut v: Vec<u32> = vector!(1; 2; 3);

v.push(6);
println!("{:?}", &v);
```

Dans le cas présent, on récupère le tout dans une slice qui est ensuite transformée en Vec. On pourrait aussi afficher tous les arguments un par un :

```
Run the following code:
macro_rules! vector {
    (
        $x:expr,$($y:expr),*
    ) => (
        println!("Nouvel argument : {}", $x);
        vector!($($y),*);
    );
    ( $x:expr ) => (
        println!("Nouvel argument : {}", $x);
    )
}

vector!(1, 2, 3, 12);
```

Vous aurez noté que j'ai remplacé les parenthèses par des accolades. Il aurait aussi été possible d'utiliser "{{ }}" ou même "[ ]". Il est davantage question de préférence personnelle.

Pourquoi "{{ }}" ?

Tout simplement parce qu'ici nous avons besoin d'un bloc d'instructions. Si votre macro ne renvoie qu'une simple expression, vous n'en aurez pas besoin.

Pattern matching encore plus poussé

En plus de simples "arguments", une macro peut en fait englober tout un code :

Run the following code:
```
macro_rules! modifier_struct {
    ($(struct $n:ident { $($name:ident: $content:ty,)+ } )+) => {
        $(struct $n { $($name: f32),+ })+
    };
}

modifier_struct! {
    struct Temperature {
        degree: u64,
    }

    struct Point {
        x: u32,
        y: u32,
        z: u32,
    }
}

fn main() {
    // error: expected f32, found u32
    let temp = Temperature { degree: 0u32 };
    // error: expected f32, found u32 (pour les 3 champs)
    let point = Point { x: 0u32, y: 0u32, z: 0u32 };
}
```

Ce code transforme tous les champs des structures en **f32**, et ce quel que soit le type initial.

Pas très utile mais ça vous permet de voir que les macros peuvent vraiment étendre les possibilités offertes par **Rust**.

Scope et exportation d'une macro

Créer des macros c'est bien, pouvoir s'en servir, c'est encore mieux ! Si vos macros sont déclarées dans un fichier à part (ce qui est une bonne chose !), il vous faudra ajouter cette ligne en haut du fichier où se trouvent vos macros :

Run the following code:
```
#![macro_use]
```

Vous pourrez alors les utiliser dans votre projet.

Si vous souhaitez exporter des macros (parce qu'elles font partie d'une bibliothèque par exemple), il vous faudra ajouter au-dessus de la macro :

Run the following code:
```
#[macro_export]
```

Enfin, si vous souhaitez utiliser des macros d'une des dépendances de votre projet, vous pourrez les importer comme cela :

Run the following code:
```
use nom_de_la_dependance::nom_de_la_macro;
```

À noter qu'avant, les imports de macros avaient besoin de `#[macro_use]` et ressemblaient à ceci :

Run the following code:
```
#[macro_use]
extern crate nom_de_la_dependance;
```

Comme ça si jamais vous croisez ce genre de code, vous ne serez pas surpris.

Quelques macros utiles

En bonus, je vous donne une petite liste de macros qui pourraient vous être utiles :

- panic!
- assert!
- assert_eq!
- compile_error!
- unreachable!
- unimplemented!
- column!
- line!
- file!

Petite macro mais grande économie de lignes !

Pour clôturer ce chapitre, je vous propose le code suivant qui permet d'améliorer celui présenté dans le chapitre sur la généricité grâce à une macro :

Run the following code:
```
macro_rules! creer_animal {
    ($nom_struct:ident) => {
        struct $nom_struct {
            nom: String,
            nombre_de_pattes: usize
        }

        impl Animal for $nom_struct {
            fn get_nom(&self) -> &str {
                &self.nom
            }

            fn get_nombre_de_pattes(&self) -> usize {
                self.nombre_de_pattes
            }
        }
    }
}

trait Animal {
    fn get_nom(&self) -> &str;
    fn get_nombre_de_pattes(&self) -> usize;
    fn affiche(&self) {
        println!("Je suis un animal qui s'appelle {} et j'ai {} pattes !", self.get_nom(), self.
    }
}

creer_animal!(Chien);
creer_animal!(Chat);

fn main() {
    fn affiche_animal<T: Animal>(animal: T) {
        animal.affiche();
    }

    let chat = Chat { nom: "Félix".to_owned(), nombre_de_pattes: 4};
    let chien = Chien { nom: "Rufus".to_owned(), nombre_de_pattes: 4};

    affiche_animal(chat);
    affiche_animal(chien);
}
```

Je tiens cependant encore à préciser que nous n'avons vu ici que la base des macros : elles permettent de faire des choses nettement plus impressionnantes (certaines crates le démontrent d'ailleurs fort bien). Les possibilités étant quasiment infinies, il ne vous reste plus qu'à expérimenter de votre côté avec ce que nous avons vu ici.

14. Box

Le type **Box** est "tout simplement" un pointeur sur des données stockées "sur le tas" (la "heap" donc).

On s'en sert notamment quand on veut éviter de trop surcharger la pile (la "stack") en instanciant directement "sur le tas".

Ou encore pour avoir une adresse constante quand on utilise une FFI (Foreign Function Interface), comme des pointeurs sur objet/fonction. Nous reviendrons sur ce sujet dans la troisième partie du cours.

Pour rappel, un programme a accès a deux types de mémoires : le tas et la pile. La pile est utilisée quand on appelle une fonction ou que l'on crée une variable. Le tas est utilisé quand vous allouez de la mémoire vous-même. Si vous souhaitez donc que de la mémoire survive au scope de sa fonction, il vous faudra donc utilisée le tas.

Pour mieux illustrer ce qu'est le type **Box**, je vous propose deux exemples :

Structure récursive

On s'en sert aussi dans le cas où on ignore quelle taille fera le type, comme les types récursifs par exemple :

```
Run the following code:
#[derive(Debug)]
enum List<T> {
    Element(T, List<T>),
    Vide,
}

fn main() {
    let list: List<i32> = List::Element(1, List::Element(2, List::Vide));
    println!("{:?}", list);
}
```

Si vous essayez de compiler ce code, vous obtiendrez une magnifique erreur : "invalid recursive enum type". (Notez que le problème sera le même si on utilise une structure). Ce type n'a pas de taille définie, nous obligeant à utiliser un autre type qui lui en a une (donc & ou bien **Box**) :

```
Run the following code:
#[derive(Debug)]
enum List<T> {
    Element(T, Box<List<T>>),
    Vide,
}

fn main() {
    let list: List<i32> = List::Element(
        1,
        Box::new(List::Element(2, Box::new(List::Vide))),
    );
    println!("{:?}", list);
}
```

Liste chaînée

Box est également utile pour la création de listes chaînées (même s'il vaut mieux utiliser le type **Vec** à la place quasiment tout le temps) :

Run the following code:

```
use std::fmt::Display;

struct List<T> {
    a: T,
    // "None" signifiera qu'on est à la fin de la liste chaînée.
    next: Option<Box<List<T>>>,
}

impl<T> List<T> {
    pub fn new(a: T) -> List<T> {
        List {
            a: a,
            next: None,
        }
    }

    pub fn add_next(&mut self, a: T) {
        match self.next {
            Some(ref mut n) => n.add_next(a),
            None => {
                self.next = Some(Box::new(List::new(a)));
            }
        }
    }
}

impl<T: Display> List<T> {
    pub fn display_all_list(&self) {
        println!("-> {}", self.a);
        match self.next {
            Some(ref n) => n.display_all_list(),
            None => {}
        }
    }
}

fn main() {
    let mut a = List::new(0u32);

    a.add_next(1u32);
    a.add_next(2u32);
    a.display_all_list();
}
```

Voilà pour ce petit chapitre rapide. **Box** est un type important auquel les gens ne pensent pas forcément alors qu'il pourrait résoudre leur(s) problème(s). Il me semblait donc important de vous le présenter.

15. Les itérateurs

Un problème couramment rencontré par les débutants en Rust est l'implémentation du trait `Iterator`. Nous allons donc tenter de remédier à cela en expliquant comme il fonctionne.

Jusqu'ici, nous savons qu'il existe deux types d'`Iterator`s :

- Les itérateurs sur/liés à un type.
- Les générateurs.

Les itérateurs sur/liés à un type

Ce type va itérer sur un ensemble de données. Bien que cette approche reste la plus complexe des deux (à cause des durées de vie notamment), sa mise en place n'a rien d'insurmontable.

Imaginons que vous ayez besoin de wrapper un `Vec` tout en ayant la capacité d'itérer sur le type fraîchement créé pour l'occasion.

Définissons la structure proprement dite :

`Run` the following code:
```
struct NewType<T>(Vec<T>);
```

Nous allons, maintenant, avoir besoin d'implémenter le trait `Iterator`. Le principal problème est que vous ne pouvez pas stocker un paramètre dans la structure `NewType` qui pourrait vous permettre de suivre la progression de la lecture à l'intérieur de votre vecteur et... c'est ici que la plupart des gens sont perdus. La solution est en réalité plutôt simple :

`Run` the following code:
```
// On crée une nouvelle structure qui contiendra une référence de votre ensemble
// de données.
struct IterNewType<'a, T: 'a> {
    inner: &'a NewType<T>,
    // Ici, nous utiliserons `pos` pour suivre la progression de notre
    // itération.
    pos: usize,
}

// Il ne nous reste plus alors qu'à implémenter le trait `Iterator` pour
// `IterNewType`.
impl<'a, T> Iterator for IterNewType<'a, T> {
    type Item = &'a T;

    fn next(&mut self) -> Option<Self::Item> {
        if self.pos >= self.inner.0.len() {
            // Il n'y a plus de données à lire, on stoppe l'itération.
            None
        } else {
            // On incrémente la position de notre itérateur.
            self.pos += 1;
            // On renvoie la valeur courante pointée par notre itérateur.
            self.inner.0.get(self.pos - 1)
        }
    }
}
```

Simple, non ? Il nous reste plus qu'à ajouter la méthode `iter` à notre structure `NewType` :

Run the following code:
```
impl<T> NewType<T> {
    fn iter<'a>(&'a self) -> IterNewType<'a, T> {
        IterNewType {
            inner: self,
            pos: 0,
        }
    }
}
```

Fini !

Voici un petit exemple d'utilisation de notre structure :

Run the following code:
```
for x in NewType(vec![1, 3, 5, 8]).iter() {
    println!("=> {}", x);
}
```

Résultat :

```
=> 1
=> 3
=> 5
=> 8
```

Les générateurs

Un générateur est une manière plutôt intéressante (et simple) d'utiliser les `Iterator`s en Rust. Un exemple sera certainement plus parlant dans ce cas précis :

Run the following code:
```
// Notre structure itère (on peut aussi dire "génère") uniquement sur les
// nombres impairs.
struct Impair {
    current: usize,
}

impl Impair {
    fn new() -> Impair {
        Impair {
            // La première valeur impaire positive est 1, donc commençons à 1.
            current: 1,
        }
    }
}

impl Iterator for Impair {
    type Item = usize;

    fn next(&mut self) -> Option<Self::Item> {
        // Déplaçons-nous à la valeur impaire suivante.
        self.current += 2;
        // On renvoie la valeur impaire courante.
        Some(self.current - 2)
    }
}

fn main() {
    // Pour éviter de boucler indéfiniment avec notre itérateur `Impair`, nous
    // avons limité la boucle à 3 valeurs.
    for x in Impair::new().take(3) {
        println!("=> {}", x);
    }
}
```

Résultat :

```
=> 1
=> 3
=> 5
```

Comme vous pouvez le constater, `Impair` génère ses propres valeurs, contrairement à l'exemple précédent qui était basé sur celles d'un vecteur. Sa conception rend la génération infinie, mais il est tout à fait possible d'établir une limite (aussi bien interne à la structure que dans son utilisation). À vous de voir selon vos besoins !

Par exemple, si on créait un itérateur sur des nombres premiers, il ne pourrait continuer que jusqu'au dernier nombre premier connu (ou alors vous possédez un data-center personnel).

Conclusion

Les itérateurs peuvent se montrer puissants et restent relativement simples à implémenter en Rust, mais les débutants ont tendance à directement gérer la ressource et itérer dessus, ce qui complique généralement la recherche de solutions potentiellement plus adaptées.

Il est toujours question de penser "Rust" ou non !

Article original

Ce chapitre a été écrit à partir de cet article de blog. N'hésitez pas à y faire un tour !

III. Aller plus loin

1. Les macros procédurales (ou proc-macros)

Je vous avais présenté les **macros** dans un chapitre précédent. Cependant, elles sont vite limitées et compliquées dès que la complexité de ce qu'on souhaite faire augmente. Pour pallier à ce problème, les **proc-macros** ont été créées. D'ailleurs, vous vous en êtes déjà servies :

```
Run the following code:
#[derive(Debug)]
pub struct S;
```

Dans ce code, `#[derive(Debug)]` est en fait une proc-macro. Il en existe plusieurs types différents :

- Les proc-macros similaires aux macros (dans leur appel) appelées **function-like macros**.
- Les **derive macros** comme dans l'exemple au-dessus.
- Les **macros attributs** :
  ```
  Run the following code:
  #[une_proc_macro]
  fn une_fonction() {}
  ```

Elles fonctionnent toutes les 3 sur le même principe : elles reçoivent un flux de tokens en argument qui représentent le code source et renvoient un autre flux de tokens (le plus souvent modifié par la proc-macro).

Avant d'aller plus loin, il faut déclarer certaines choses dans son `Cargo.toml`. En effet : une proc-macro ne peut être créée que dans une crate de type bibliothèque, pas dans un binaire. Donc si vous avez besoin de créer une proc-macro pour les besoins d'un projet, il faudra créer une bibliothèque qui contiendra spécifiquement cette proc-macro. La raison en est toute simple : le compilateur ne compile pas le code pour une proc-macro de la même façon.

Déclarons maintenant notre projet "proc_test" dans notre `Cargo.toml` :

```
[package]
name = "proc_test"
version = "0.1.0"
edition = "2021"

[lib]
proc-macro = true
```

Au final, la seule chose qui change est l'ajout de `proc-macro = true` au final. Cependant, en ajoutant cette option, votre code aura maintenant accès à la crate proc_macro qui fournit des types qui seront nécessaires pour leur écriture.

function-like macro

Écrivons maintenant un petit exemple avec une **function-like macro** :

```
Run the following code:
use proc_macro::TokenStream;

#[proc_macro]
pub fn genere_dit_bonjour(_item: TokenStream) -> TokenStream {
    "fn dit_bonjour() { println!(\"bonjour\"); }".parse().unwrap()
}
```

Expliquons ce code maintenant.

TokenStream représente le flux des tokens fournit par le compilateur. C'est dans ce flux que les arguments qui seront

passés dans notre macro seront stockés.

`#[proc_macro]` est un attribut qui indique le type de notre proc-macro. Il y a un attribut différent pour chaque type de proc-macro, nous y reviendrons plus tard.

La fonction `genere_dit_bonjour` reçoit donc en argument le **TokenStream** qui contient ce qui est écrit dans l'appel de macro et renvoie un autre **TokenStream** qui contient ce qui doit être mis à la place de l'appel de cette macro.

Enfin, nous générons donc la fonction `dit_bonjour` qui appelle `println` et se termine. La partie intéressante étant `.parse().unwrap()`. Il est possible de convertir une `String` en **TokenStream** de cette façon. Le compilateur va parser la

`String` comme il le ferait avec du code Rust puis générer le flux de tokens.

Donc maintenant on appelle cette proc-macro dans un autre code :

Run the following code:
```
use proc_test::genere_dit_bonjour;

genere_dit_bonjour!();

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

Si on compile ce code et qu'on l'exécute, on va obtenir :

```
bonjour
```

C'est bien évidemment un test très basique mais je pense que vous commencez à en voir les possibilités. On va maintenant regarder un autre exemple avec une **derive macro**.

derive macro

Pour nous faciliter la vie, on va utiliser les crates syn pour parser le **TokenStream**, et quote pour générer le **TokenStream**. Ces deux crates sont parmi les plus téléchargées de tout l'écosystème de Rust, et pour cause : elles facilitent énormément l'écriture des proc-macros.

Le but de notre derive macro va être de générer des getters et des setters pour chaque champs du type sur lequel elles seront utilisées. Pour nous faciliter la vie, si le type en question est une enum, on va juste renvoyer une erreur de compilation.

Donc avant d'aller plus loin, il faut que l'on tienne compte de plusieurs choses :

- Est-ce que le champs est visible ou non ? Les méthodes que l'on va générer doivent avoir la même visibilité.
- Est-ce que le type a des génériques ? Si oui il ne faut pas oublier de les ajouter dans le bloc d'`impl` sinon ça ne va pas compiler.

Et c'est plus ou moins tout. Commençons par la création de notre fonction de derive :

Run the following code:
```
#[proc_macro_derive(GetSet)]
pub fn derive_get_set(input: TokenStream) -> TokenStream {
    // le code
}
```

Notre derive-macro sera donc appelée de cette façon :

Run the following code:
```
#[derive(GetSet)]
pub struct S {
    a: u8,
}

// Les getters et setters pour `S::a` seront donc générés.
```

Maintenant commençons son implémentation :

Run the following code:
```
use proc_macro::TokenStream;
use syn::{DeriveInput, parse_macro_input};

#[proc_macro_derive(GetSet)]
pub fn derive_get_set(input: TokenStream) -> TokenStream {
    // On parse le contenu de `TokenStream` avec `syn`.
    let input = parse_macro_input!(input as DeriveInput);

    // On peut maintenant gérer chaque type facilement.
    match input.data {
        Data::Enum(_) => {
            return "compile_error!(\"Enum types are not supported\")"
                .parse()
                .unwrap()
        }
        Data::Struct(s) => {
            // Générer getters et setters pour les structs.
        }
        Data::Union(u) => {
            // Générer getters et setters pour les union.
        }
    }
}
```

Comme vous pouvez le voir, on génère une erreur si jamais le type sur lequel notre proc-macro est utilisée est une enum.

Il reste maintenant à gérer le type **union** et les différents genres du type **struct**. Pour chacun de ces types, nous devons récupérer pour chaque champ : son nom, sa visibilité et son type. Nous aurons aussi besoin du nom du type sur lequel notre proc-macro est utilisée, ses génériques ainsi qu'une information importante : est-ce que le type est une **union** (pour savoir si on doit déclarer les méthodes comme **unsafe** ou non). Nous enverrons ensuite ces informations dans une fonction qui se chargera de générer les getters et les setters :

Run the following code:

```
use syn::{Data, DeriveInput, Fields, parse_macro_input};
use proc_macro::TokenStream;

#[proc_macro_derive(GetSet)]
pub fn derive_get_set(input: TokenStream) -> TokenStream {
    let input = parse_macro_input!(input as DeriveInput);

    // On récupère le nom (ident), la visibilité (vis) et le type (ty) de
    // chaque champ.
    match input.data {
        Data::Enum(_) => {
            return "compile_error!(\"Enum types are not supported\")"
                .parse()
                .unwrap()
        }
        Data::Struct(s) => match s.fields {
            Fields::Named(fields) => {
                let iterateur = fields.named
                    .iter()
                    .map(|champ| {
                        (champ.ident.as_ref().unwrap(), &champ.vis, &champ.ty)
                    });
                implementer_getters_setters(
                    input.ident, input.generics, iterateur, false
                )
            }
            Fields::Unnamed(fields) => {
                // Si jamais on a `struct Foo(u32, pub u8, char)`, il vaut gérer
                // le nom de chaque champ différemment. `u32` sera donc 0 et
                // ainsi de suite.
                let iterateur = fields.unnamed
                    .iter()
                    .enumerate()
                    .map(|(position, champ)| (position, &champ.vis, &champ.ty));
                implementer_getters_setters(
                    input.ident, input.generics, iterateur, false,
                )
            }
            // S'il n'y a pas de champ, on retourne un flux de tokens vide car
            // il n'y a rien à faire.
            Fields::Unit => return TokenStream::new(),
        },
        Data::Union(u) => {
            let iterateur = u.fields
                .named
                .iter()
                .map(|champ| {
                    (champ.ident.as_ref().unwrap(), &champ.vis, &champ.ty)
                });
            implementer_getters_setters(
                input.ident, input.generics, iterateur, true,
            )
        }
    }
}
```

Notre première fonction est terminée. Implémentons donc maintenant `implementer_getters_setters` dans laquelle nous allons notamment nous servir de la crate `quote` :

Run the following code:

```
use syn::{Generics, Ident, Type, Visibility};
use proc_macro::TokenStream;
use quote::{format_ident, quote};

fn implementer_getters_setters<'a, S: ToString, I: Iterator<Item = (S, &'a Visibility, &'a Type)
    nom_du_type: Ident,
    generiques: Generics,
    champs: I,
    est_une_union: bool,
) -> TokenStream {
    // Dans un premier tempsm on convertit l'itérateur de champs en une liste de
    // `TokenStream`.
    let getters_setters = champs
        .map(|(nom, visibilite, type_)| {
            // On convertit le nom (qui est un `ToString`) en `Ident` pour pouvoir
            // l'utiliser dans `format_ident`.
            let nom = format_ident!("{}", nom.to_string());
            // On génère le nom du getter.
            let getter = format_ident!("get_{}", nom);
            // On génère le nom du setter.
            let setter = format_ident!("set_{}", nom);
            // Si le type est une union, il faut un bloc `unsafe` pour pouvoir
            // avoir accès à ses champs.
            let unsafe_ident = if est_une_union {
                Some(format_ident!("unsafe"))
            } else {
                None
            };

            // On génère le getter et le setter pour ce champ. Chaque `#` est
            // par `quote` pour qu'il génère le code de la variable qui suit
            // et pas simplement écrire le nom tel quel.
            quote! {
#visibilite #unsafe_ident fn #getter(&self) -> &#type_ {
    &self.#nom
}

#visibilite #unsafe_ident fn #setter(&mut self, value: #type_) {
    self.#nom = value;
}
            }
        })
        .collect::<Vec<_>>();

    // Si jamais il n'y avait pas de champs, inutile de faire quoi que ce soit
    // de plus.
    if getters_setters.is_empty() {
        return TokenStream::new();
    }
    // On sépare les génériques pour pouvoir les déclarer correctement dans le
    // bloc de l'impl.
    let (generiques_pour_impl, generiques_pour_type, where_clause) =
        generiques.split_for_impl();
    // Dernière partie, on génère le bloc d'impl avec le nom du type ainsi que
    // ses génériques.
    TokenStream::from(quote! {
impl #generiques_pour_impl #nom_du_type #generiques_pour_type #where_clause {
    #(#getters_setters)*
}
    })
}
```

Et voilà ! Pour tester le résultat :

Run the following code:
```
use proc_test::GetSet;

#[derive(Default, GetSet)]
pub struct A<T> {
    foo: u32,
    pub bar: f64,
    pub(crate) gen: T,
}

#[derive(GetSet)]
pub union B {
    x: u16,
    pub y: u8,
}

fn main() {
    let mut a = A {
        foo: 0,
        bar: 1.,
        gen: String::from("a"),
    };

    a.set_gen(String::from("une autre string"));
    println!("=> {}", a.get_gen());

    let mut b = B {
        x: 0,
    };

    unsafe {
        b.set_y(5);
        println!("=> {}", b.get_y());
    }
}
```

Une autre façon serait de générer la documentation avec `cargo doc` et de vérifier que les méthodes sont bien générées.

Si jamais vous souhaitez utiliser des attributs qui n'existent pas dans votre proc-macro (par exemple en disant qu'on ne souhaite pas qu'un champ ait un getter, un setter ou aucun des deux), vous devez les déclarer dans `proc_macro_derive`. Par exemple :

Run the following code:
```
#[proc_macro_derive(GetSet, attributes(no_getter, no_setter))]
```

Après il suffira de regarder si l'attribut est présent dans les champs `attrs` des différents types de `syn` et d'ajouter l'information dans l'itérateur. Regardons à présent les **macros attributs**.

macro attribut

Contrairement aux deux précédentes, celle-ci permet de modifier l'item sur lequel elle est utilisée. Sa signature est aussi un peu différente :

Run the following code:
```
#[proc_macro_attribute]
pub fn modifier_item(attribut: TokenStream, item: TokenStream) -> TokenStream {
    item
}
```

En premier paramètre, elle prend les arguments de l'attribut et en second elle prend tout l'item sur lequel elle est utilisée (toujours sous forme de `TokenStream`, bien évidemment). Modifions un peu la fonction pour qu'elle affiche ce qu'elle reçoit :

Run the following code:
```
#[proc_macro_attribute]
pub fn modifier_item(attribut: TokenStream, item: TokenStream) -> TokenStream {
    println!("attribut: \"{}\"", attribut.to_string());
    println!("item: \"{}\"", item.to_string());
    item
}
```

Et maintenant regardons ce que ça affiche quand on utilise cet attribut :

Run the following code:
```
use proc_test::modifier_item;

#[modifier_item]
pub fn foo() {}

#[modifier_item(bonjour)]
pub struct Bonjour;

#[modifier_item { bonjour }]
pub type BonjourType = Bonjour;

#[modifier_item(bonjour >>> 2)]
pub fn foo2() {}
```

Ce qui affichera (à la compilation) :

```
attribut: ""
item: "pub fn foo() {}"
attribut: "bonjour"
item: "pub struct Bonjour ;"
attribut: "bonjour"
item: "pub type BonjourType = Bonjour ;"
attribut: "bonjour >> > 2"
item: "pub fn foo2() {}"
```

Elle est donc beaucoup plus puissante et permissive que les deux précédentes. Comme je vous ai déjà montré un exemple avec une **derive macro**, je pense que vous avez les bases pour vous en sortir.

2. La compilation conditionnelle

Si vous souhaitez qu'une partie de votre code soit compilée mais seulement dans certaines conditions, par exemple sur un système d'exploitation en particulier, il est possible de le faire avec la **compilation conditionnelle**.

Par exemple pour savoir sur quelle système le programme est compilé, on va écrire en C :

```
#ifdef linux
  #define SYSTEM "linux"
#elif _WIN32
  #define SYSTEM "windows"
#endif

void show_system() {
    printf("%s", SYSTEM);
}
```

Et en Rust on va écrire :

Run the following code:
```
#[cfg(target_os = "linux")]
const SYSTEM: &str = "linux";
#[cfg(target_os = "windows")]
const SYSTEM: &str = "windows";

fn show_system() {
    println!("{}", SYSTEM);
}
```

C'est donc avec l'attribut `cfg` que la compilation conditionnelle est gérée.

Ajouter des conditions dans l'attribut cfg

Avec le code précédent, si on compile sur un autre système que Windows ou Linux, la compilation va échouer car `SYSTEM` ne sera pas défini. Pour contourner le problème, il nous suffit de rajouter une autre déclaration de `SYSTEM` dans le cas où le système n'est ni Linux ni Windows :

Run the following code:
```
#[cfg(not(any(target_os = "linux", target_os = "windows")))]
const SYSTEM: &str = "inconnu";
```

`cfg` peut donc prendre des conditions qui peuvent être imbriquées. Dans l'exemple ci-dessus, nous avons utilisé `not` et `any`. Il existe une troisième condition : `all`. Expliquons ce que chacun de ces attributs fait :

- `all` renverra `true` tant qu'aucun de ses arguments ne renvoie `false`.
- `any` renverra `true` tant qu'au moins un de ses arguments renvoie `true`.
- `not` inverse la condition. C'est un équivalent de `!`. Il ne prend qu'un seul argument.

Donc pour résumer :

Run the following code:
```
#[cfg(all())] // true
#[cfg(any())] // false
```

Arguments de cfg

Jusqu'à présent, nous n'avons vu que `target_os`, cependant il en existe bien d'autres :

- `target_arch` : Correspond à l'architecture du CPU. Par exemple `x86_64`, `arm`, `aarch64`...
- `target_family` : Une "famille" de système d'exploitations comme `windows`, `unix` ou `wasm`.
- `target_endian` : Correspond à l'endianness du CPU. Peut prendre comme valeur `big` ou `small`.
- `target_pointer_width` : Correspond à la taille d'un pointeur. Ce doit être une puissance de 2. Par exemple `16`, `32`, `64`...
- `feature` : Les features dans Rust sont déclarées dans le fichier `Cargo.toml` comme déjà évoqué dans le chapitre sur "Cargo" justement. Elles permettent de rendre certaines fonctionnalités optionnelles pour pouvoir par exemple compiler plus rapidement, générer un binaire plus petit, etc.

Il existe aussi des cas sans valeur associée :

- `test` : Quand on compile notre programme avec `--test` pour lancer les tests unitaires. On revient sur les tests unitaires un peu plus loin dans ce livre.
- `doc` : Quand on est en train de générer la documentation pour notre crate. Cela peut être utile dans certains cas pour unifier l'API visible dans la documentation.
- `doctest` : On on lance les tests de la documentation.

Il y a encore beaucoup d'autres valeurs possible. Une liste plus exhaustive est disponible dans la référence.

L'attribut cfg_attr

Imaginons que vous ne vouliez générer les implémentations du trait `Debug` via `derive` uniquement lorsque la feature `debug` est activée. On pourrait écrire :

Run the following code:
```
#[cfg(feature = "debug")]
#[derive(Debug)]
pub struct Struct;

#[cfg(not(feature = "debug"))]
pub struct Struct;
```

Cependant ce n'est pas très pratique, surtout si on doit dupliquer beaucoup de code. C'est là que `cfg_attr` devient utile. Plutôt que de dupliquer ce code, on peut écrire :

Run the following code:
```
#[cfg_attr(feature = "debug", derive(Debug))]
pub struct Struct;
```

Le premier argument de `cfg_attr` est la condition de compilation. Le deuxième est l'attribut que l'on souhaite générer si la condition du premier argument est satisfaite.

Donc si vous voulez utiliser un attribut mais seulement dans certaines conditions, utilisez `cfg_attr`.

La macro cfg!

Voici le dernier cas pour la compilation conditionnelle : la macro `cfg!`. Reprenons notre premier exemple :

Run the following code:
```
fn show_system() {
    if cfg!(target_os = "linux") {
        println!("linux");
    } else if cfg!(target_os = "windows") {
        println!("windows");
    } else {
        println!("inconnu");
    }
}
```

Comme la condition dans `cfg!` sera remplacée par `true` ou `false` au moment de la compilation quand la macro sera

étendue ("expanded" en anglais), si on compile sur Linux, le code ressemblera à ça :

Run the following code:
```
fn show_system() {
    if true {
        println!("linux");
    } else if false {
        println!("windows");
    } else {
        println!("inconnu");
    }
}
```

Et comme le compilateur voit au moment de la compilation que les conditions des branches sont déjà résolues, il va simplement les supprimer. Ce qui va donner :

Run the following code:
```
fn show_system() {
    println!("linux");
}
```

Vous savez maintenant comment gérer la compilation conditionnelle en Rust.

3. Utiliser du code compilé en C

Rust permet d'exécuter du code compilé en C au travers des Foreign Function Interface (aussi appelées FFI). Ce chapitre va vous montrer comment faire.

Les bases

La première chose à faire est d'ajouter une dépendance à la crate **libc** :

Cargo.toml :

```
[dependencies]
libc = "0.2"
```

Bien que cette étape ne soit pas obligatoire, **libc** fournit un grand nombre de type C sur un grand nombre de plateformes/architectures. Il serait bête de s'en passer et de devoir le refaire soi-même !

Toute fonction que vous voudrez utiliser doit être déclarée ! Par exemple, utilisons la fonction rename :

Run the following code:
```
use std::ffi::CString;

extern "C" {
    fn rename(
        old: *const libc::c_char,
        new_p: *const libc::c_char,
    ) -> libc::c_int;
}

fn main() {
    if unsafe {
        rename(
            CString::new("old").unwrap().as_ptr(),
            CString::new("new").unwrap().as_ptr(),
        )
    } != 0 {
        println!("Rename failed");
    } else {
        println!("successfully renamed !");
    }
}
```

À noter qu'il est tout à fait possible de ne pas passer par les types fournis par la **libc** :

Run the following code:
```
extern "C" {
    fn rename(old: *const i8, new_p: *const i8) -> i32;
}
```

Cependant je vous le déconseille fortement. Les types fournis par la **libc** ont l'avantage d'être plus clairs et surtout de correspondre au type C. Dans ce code, **char** n'est pas nécessairement un entier signé, ni même de 8 bits.

Regardons maintenant comment utiliser des fonctions d'une bibliothèque C.

Interfaçage avec une bibliothèque C

Tout d'abord, il va falloir linker notre code avec la bibliothèque C que l'on souhaite utiliser :

```
Run the following code:
// Dans le fichier principal.

#[cfg(target_os = "linux")]
mod platform {
    #[link(name = "nom_de_la_bibliotheque")] extern {}
}
```

Dans le cas présent j'ai mis **linux**, mais sachez que vous pouvez aussi mettre **win32**, **macos**, etc.... Il est aussi possible de préciser l'architecture de cette façon :

```
Run the following code:
#[cfg(target_os = "linux")]
mod platform {
    #[cfg(target_arch = "x86")]
    #[link(name = "nom_de_la_bibliotheque_en_32_bits")] extern{}
    #[cfg(target_arch = "x86_64")]
    #[link(name = "nom_de_la_bibliotheque_en_64_bits")] extern{}
}
```

Nous avons donc maintenant les bases.

Interfacer les fonctions

Tout comme je vous l'ai montré précédemment, il va falloir redéclarer les fonctions que vous souhaitez utiliser. Il est recommandé de les déclarer dans un fichier **ffi.rs** (c'est ce qui généralement fait). Vous allez aussi enfin voir les **structures unitaires** en action !

On va dire que la bibliothèque en C ressemble à ça :

```
#define NOT_OK 0
#define OK 1

// On ne sait pas ce que la structure contient.
struct Handler;

Handler *new();
int do_something(Handler *h);
int add_callback(Handler *h, int (*pointeur_sur_fonction)(int, int)););
void destroy(Handler *h);
```

Nous devons écrire son équivalent en Rust, ce que nous allons faire dans le fichier **ffi.rs** :

Run the following code:

```
use libc::{c_int, c_void, c_char};

enum Status {
    NotOk = 0,
    Ok = 1,
}

// Cette metadata n'est pas obligatoire mais il est recommandé de la mettre
// quand on manipule des objets venant du C.
#[repr(C)]
pub struct FFIHandler; // La structure unitaire.

extern "C" {
    pub fn new() -> *mut FFIHandler;
    pub fn do_something(handler: *mut FFIHandler) -> c_int;
    pub fn add_callback(
        handler: *mut FFIHandler,
        fonction: *mut c_void,
    ) -> c_int;
    pub fn set_name(handler: *mut FFIHandler, name: *const c_char);
    pub fn get_name(handler: *mut FFIHandler) -> *const c_char;
    pub fn destroy(handler: *mut FFIHandler);
}
```

Voilà pour les déclarations du code C. Nous pouvons attaquer le portage à proprement parler. Comme l'objet que l'on va binder s'appelle **Handler**, on va garder le nom en Rust :

Run the following code:

```
// Dans le fichier handler.rs :
use libc::{c_int, c_void, c_char};
use ffi::{self, FFIHandler};

pub struct Handler {
    pointer: *mut FFIHandler,
}

impl Handler {
    pub fn new() -> Result<Handler, ()> {
        let tmp = unsafe { ffi::new() };

        if tmp.is_null() {
            Ok(Handler { pointer: tmp })
        } else {
            Err(())
        }
    }

    pub fn do_something(&self) -> Status {
        unsafe { ffi::do_something(self.pointer) }
    }

    pub fn add_callback(&self, fonction: fn(isize, isize) -> isize) -> Status {
        unsafe { ffi::add_callback(self.pointer, fonction as *mut c_void) }
    }

    pub fn set_name(&self, name: &str) {
        unsafe { ffi::set_name(self.pointer, name.as_ptr() as *const c_char) }
    }

    pub fn get_name(&self) -> String {
        let tmp unsafe { ffi::get_name(self.pointer) };

        if tmp.is_null() {
            String::new()
        } else {
            unsafe {
                String::from_utf8_lossy(
                    std::ffi::CStr::from_ptr(tmp).to_bytes(),
                ).to_string()
            }
        }
    }
}

impl Drop for Handler {
    fn drop(&mut self) {
        if !self.pointer.is_null() {
            unsafe { ffi::destroy(self.pointer); }
            self.pointer = std::ptr::null_mut();
        }
    }
}
```

Voilà, vous devriez maintenant pouvoir vous en sortir avec ces bases. Nous avons vu comment ajouter un callback, convertir une **String** entre C et Rust et nous avons surtout pu voir les **structures unitaires** en action !

4. Documentation et rustdoc

En plus du compilateur, Rust possède un générateur de documentation. Toute la documentation en ligne de la bibliothèque standard (disponible ici) a été générée avec cet outil. Vous allez voir qu'il est très facile de s'en servir.

Génération de la documentation

Avant de voir comment écrire la documentation, je pense qu'il serait plus intéressant de voir comment la générer et surtout à quoi ça ressemble.

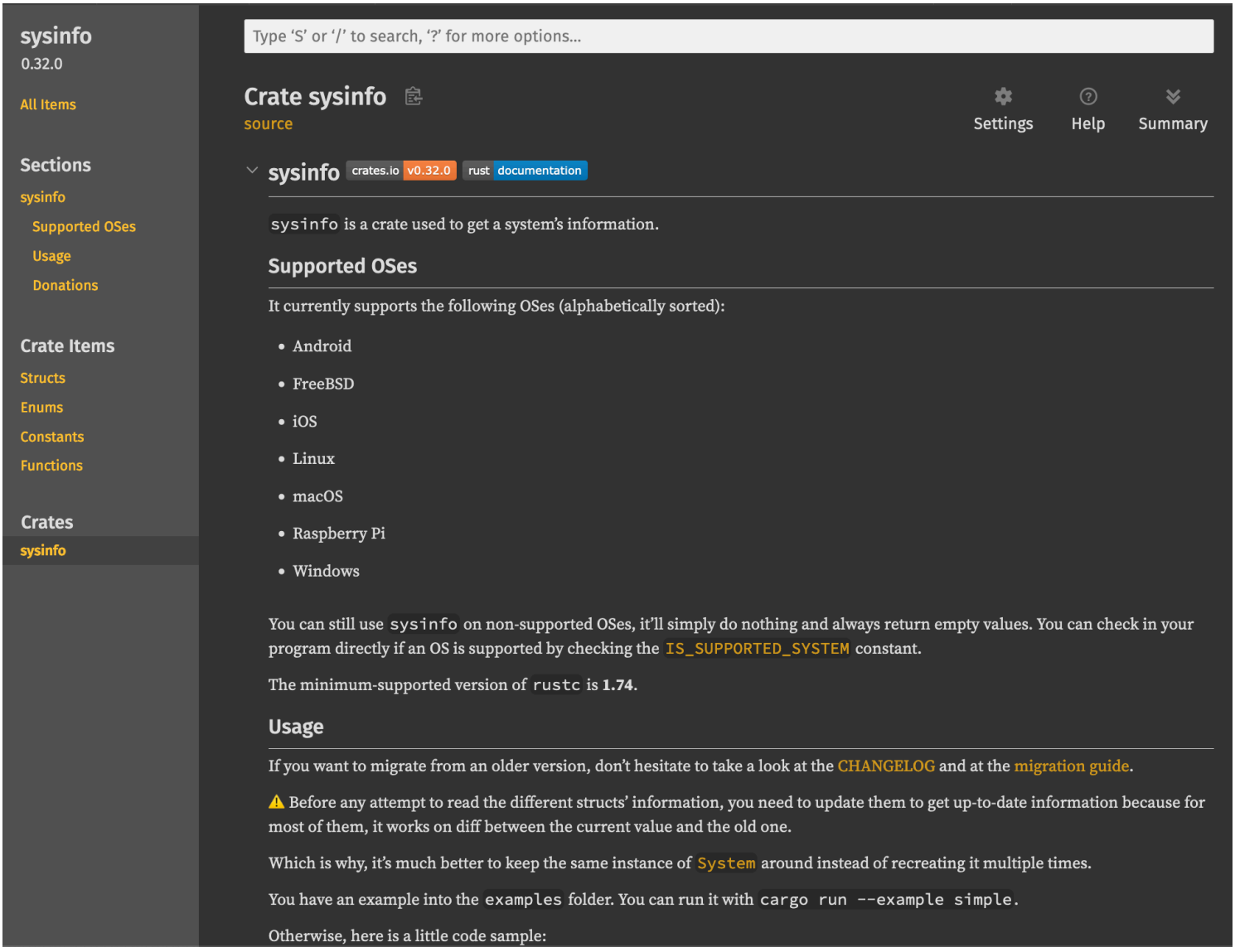
Si vous utilisez **Cargo**, rien de plus simple :

```
cargo doc
```

Votre documentation se trouvera dans le dossier **target/doc/le_nom_de_votre_cagette/**. Pour l'afficher, ouvrez le fichier **index.html** qui s'y trouve avec votre navigateur internet préféré, ou utilisez la commande :

```
cargo doc --open
```

Ce qui devrait vous ouvrir une page dans votre navigateur. Si je lance `cargo doc --open` sur la crate `sysinfo`, la page ressemblera à ça :



Maintenant, si vous souhaitez le faire sans passer par **Cargo** :

```
rustdoc le_nom_de_votre_fichier_source
```

Le contenu sera généré dans le dossier **doc/**. Pour consulter la documentation générée, c'est pareil que pour **Cargo**.

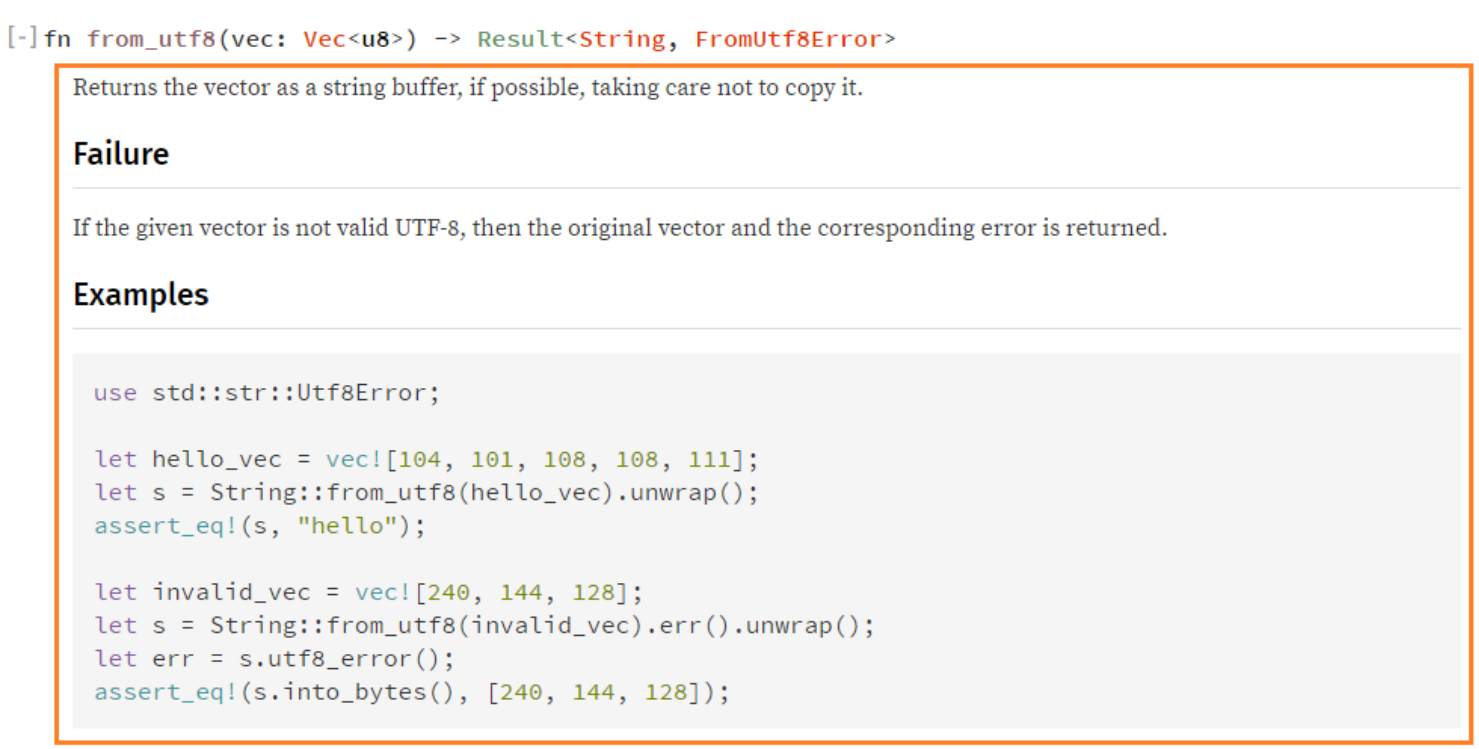
Il est important de noter que **rustdoc** accepte aussi les fichiers markdown (**.md**) comme argument :

```
rustdoc fichier.md
```

Cela créera un fichier **doc/fichier.html**.

Ajouter de la documentation

Pour le moment, la documentation qu'on a généré ne contient que du code sans rien d'autre. Pas génial pour de la documentation donc... Au final, ce serait bien qu'on ait des explications sur les items, comme ici :



Pour cela, rien de plus simple, il suffit d'utiliser les "///" (aussi appelé "doc comments") :

Run the following code:

```
/// Et ici je mets la description
/// que je veux !
fn une_fonction() {}

/// Et le markdown aussi fonctionne :
///
/// ```
/// println!("quelque chose");
/// // ou même un exemple d'utilisation de la structure !
/// ```
struct UneStruct {
    /// ce champ sert à faire ceci
    un_champ: 32,
    /// et ce champ sert à faire cela
    un_autre_champ: i32
}
```

Bon à savoir : `///` est du sucre syntaxique qui est remplacé par `#[doc = "..."]`. Si vous vous souvenez du chapitre sur les attributs, cela signifie que l'on peut avoir l'équivalent interne de cet attribut :

```
Run the following code:
//! Je documente l'item dans lequel je suis.
#![doc = "Je suis la deuxième ligne de cette documentation."]
//! Et moi la troisième.
```

La documentation en Rust utilise le format **commonmark**, qui est une spécification de **markdown**. Donc vous pouvez ajouter du style sans problème. Par exemple :

```
Run the following code:
/// _italique_ *italique aussi*
/// _gras_ **gras aussi**
/// `code inline`
/// # Gros titre
/// ## Titre plus petit
/// [lien vers mon site](https://blog.guillaume-gomez.fr)
```

Je vous invite maintenant à essayer cela sur vos codes pour voir le résultat obtenu. Il est cependant important de noter que les "///" doivent être mis **avant** l'objet qu'ils doivent documenter. Ce code ne fonctionnera donc pas :

```
Run the following code:
enum Option<T> {
    None,
    Some(T), /// Some value `T`
}
```

Lien intra-doc

Une autre fonctionnalité très utile appelée "lien intra-doc" permet de générer des liens vers des items. Par exemple :

```
Run the following code:
/// Cette fonction retourne un [Type].
pub fn fonction() -> Type {
    Type
}

/// Cette structure est créé dans la fonction [fonction].
pub struct Type;
```

Dans l'exemple ci-dessus, `[Type]` et `[fonction]` seront des liens qui pointeront vers les pages de `Type` et de `fonction`. Ils fonctionnent aussi avec les paths :

```
Run the following code:
/// Ce type peut être initialisé avec la méthode [Type::new].
pub struct Type;

impl Type {
    pub fn new() -> Type {
        Type
    }
}
```

`[Type::new]` pointera vers la méthode `new` de `Type`.

Cacher un item

Si vous souhaitez pouvoir utiliser un item défini dans une crate mais que cet item n'apparaisse pas dans la documentation, vous pouvez le cacher avec `#[doc(hidden)]` :

```
Run the following code:
#[doc(hidden)]
pub struct Struct;
```

`Struct` ne sera donc pas généré dans la documentation et ne pourra pas être trouvé avec la fonctionnalité de recherche.

Ajouter un alias de recherche

La fonctionnalité de recherche est très pratique dans rustdoc, et il est possible de s'assurer que certaines recherches renvoient des items dont le nom n'a rien à voir en utilisant `#[doc(alias)]` :

```
Run the following code:
#[doc(alias = "error")]
#[doc(alias = "Error")]
pub struct JeSuisUneErreur;
```

Avec ces 2 alias, que l'on cherche "error" ou "Error", `JeSuisUneErreur` sera affiché parmi les résultats.

Personnalisation du rendu

Il est possible de personnaliser quelques éléments dans la documentation générée, comme la favicon (l'image miniature dans l'onglet de votre navigateur) ou le logo dans la barre latérale :

```
Run the following code:
// Pour changer la favicon.
#![doc(html_favicon_url = "https://example.com/favicon.ico")]
// Pour changer le logo.
#![doc(html_logo_url = "https://example.com/logo.jpg")]
```

Ces 2 attributs doivent être utilisés dans le module racine du projet, donc très sans doute dans le fichier `main.rs` ou `lib.rs` en fonction de si votre crate est un binaire ou une bibliothèque.

Voilà, vous savez maintenant gérer des documentations en Rust ! Il reste toutefois un point que nous n'avons pas abordé : il est possible d'ajouter des exemples de codes qui seront testés directement dans votre documentation. Nous allons en parler dans le prochain chapitre.

5. Ajouter des tests

Dans ce chapitre, nous allons parler des tests et en particulier de l'attribut `#[test]`.

En Rust, il est possible d'écrire des tests unitaires directement dans un fichier qui peuvent être lancés par **Cargo** ou le compilateur de **Rust**.

Avec **Cargo** :

```
cargo test
```

Avec **rustc** :

```
rustc --test votre_fichier_principal.rs
./votre_fichier_principal
```

Regardons maintenant comment créer ces tests unitaires :

L'attribut #[test]

Pour indiquer au compilateur qu'une fonction est un test unitaire, il faut ajouter l'annoter avec l'attribut `#[test]`. Exemple :

Run the following code:
```
fn some_func(valeur1: i32, valeur2: i32) -> i32 {
    valeur1 + valeur2
}

#[test]
fn test_some_func() {
    assert_eq!(3, some_func(1, 2));
}
```

Et c'est tout... Il est courant de grouper les tests unitaires dans un module :

Run the following code:
```
fn some_func(valeur1: i32, valeur2: i32) -> i32 {
    valeur1 + valeur2
}

#[cfg(test)] // On ne compile ce module que si on est en mode "test".
mod tests {
    use super::some_func;

    #[test] // Cette fonction est donc un test unitaire.
    fn test_some_func() {
        assert_eq!(3, some_func(1, 2));
    }
}
```

Ça permet de découper un peu le code.

La métadonnée #[should_panic]

Maintenant, si vous voulez vérifier qu'un test **échoue**, il vous faudra utiliser cet attribut :

```
Run the following code:
fn some_func(valeur1: i32, valeur2: i32) -> i32 {
    valeur1 + valeur2
}

#[test] // C'est un test.
#[should_panic] // Il est censé paniquer.
fn test_some_func() {
    assert_eq!(4, some_func(1, 2)); // 1 + 2 != 4, donc ça doit paniquer.
}
```

Quand vous lancerez l'exécutable, il vous confirmera que le test s'est bien déroulé (parce qu'il a paniqué comme attendu). Petit bonus : vous pouvez ajouter du texte qui sera affiché lors de l'exécution du test :

```
Run the following code:
#[test]
#[should_panic(expected = "1 + 2 != 4")]
fn test_some_func() {
    assert_eq!(4, some_func(1, 2));
}
```

Mettre les tests dans un dossier à part

Si vous utilisez **Cargo**, il est aussi possible d'écrire des tests dans un dossier à part. Commencez par créer un dossier **tests** puis créez un fichier **.rs**:

```
Run the following code:
#[test]
fn test_some_func() {
    assert_eq!(3, ma_lib::some_func(1, 2));
}
```

Ensuite cela fonctionne de la même façon : lancez la commande `cargo test` et les tests dans ce dossier seront exécutés.

Écrire des suites de tests

Si vous souhaitez regrouper plusieurs tests dans un même dossier (mais toujours dans le dossier **tests**), rien de bien difficile une fois encore. Ça devra ressembler à ça :

```
 - tests
    |
    |- la_suite_de_tests.rs
    |- sous_dossier
        |
        |- fichier1.rs
        |- fichier2.rs
        |- mod.rs
```

Je pense que vous voyez déjà où je veux en venir : il va juste falloir importer le module **sous_dossier** pour que les tests contenus dans **fichier1.rs** et **fichier2.rs** soient exécutés.

la_suite_de_tests.rs

```
Run the following code:
mod sous_dossier; // Et c'est tout !
```

sous_dossier/mod.rs

```
Run the following code:
mod fichier1;
mod fichier2;
```

Et voilà ! Vous pouvez maintenant écrire tous les tests que vous voulez dans **fichier1.rs** et **fichier2.rs** (en n'oubliant pas d'ajouter `#[test]` !).

Tests dans la documentation ?

Comme évoqué dans le chapitre précédent, on peut ajouter des exemples de code dans la documentation. Ce que je ne vous avais pas dit, c'est que lorsque vous lancez `cargo test`, ces exemples sont eux aussi testés. C'est très pratique car cela permet de les maintenir à jour assez facilement.

Options de test

Il est possible d'ajouter des options de test pour les codes d'exemple dans la documentation. Nous allons voir certains cas.

```
/// ```
/// let x = 12;
/// ```
```

C'est l'exemple de code par défaut. Si aucune option n'est passée, **rustdoc** partira donc du principe que c'est un code **Rust** et qu'il est censé compiler et s'exécuter sans paniquer.

Il est strictement équivalent au code suivant :

```
/// ```rust
/// let x = 12;
/// ```
```

Si vous voulez écrire du code dans un autre langage, écrivez juste son nom à la place de l'attribut **rust** :

```
/// ```C
/// int c = 12;
/// ```
```

Dans ce cas-là, ce code sera ignoré lors des tests.

Il se peut aussi que vous ayez envie d'ignorer un test :

```
/// ```ignore
/// let x = 12;
/// ```
```

Il sera marqué comme **ignored** mais vous le verrez lors des tests.

Un autre cas assez courant est de vouloir tester que la compilation se passe bien mais sans exécuter le code (généralement pour des exemples d'I/O) :

```
/// ```no_run
/// let x = File::open("Un-fichier.txt").expect("Fichier introuvable");
/// ```
```

Il est aussi possible de combiner plusieurs options en les séparant par une virgule :

```
/// ```compile_fail,no_run
/// let x = 12;
/// ```
```

Un dernier exemple :

```
```test_harness
#[test]
fn foo() {
    fail!("oops! (will run & register as failure)")
}
```
```

Cela compile le code comme si le flag "--test" était donné au compilateur.

En bref, il y a pas mal d'options qui vous sont proposées dont voici la liste complète :

- **rust** : par défaut
- **ignore** : pour dire à **rustdoc** d'ignorer ce code
- **should_panic** : le test échouera si le code s'exécute sans erreur
- **no_run** : ne teste que la compilation
- **test_harness** : compile comme si le flag "--test" était donné au compilateur
- **compile_fail** : teste que la compilation échoue
- **allow_fail** : en gros, si l'exécution échoue, ça ne fera pas échouer le test. Par contre le test doit compiler.

Tout autre option sera considérée comme un langage autre que Rust et passera le code en **ignore** invisible (vous ne le verrez pas apparaître dans la liste des codes testés).

Cacher des lignes

Dans certains cas, vous pourriez vouloir cacher des lignes lors du rendu du code dans la documentation mais les garder lors du test. Exemple :

```
/// ```
/// # fn foo() -> io::Result<()> {
/// let f = File::open("un-fichier.txt")?;
/// # }
/// ```
```

Quand la doc sera générée, le lecteur ne verra plus que :

```
Run the following code:
let f = File::open("un-fichier.txt")?;
```

Par contre, lors du lancement des tests, tout le code sera bien présent. Plutôt pratique si jamais vous avez besoin de concentrer l'attention du lecteur sur un point précis !

6. Rc et RefCell

Ce chapitre va vous permettre de comprendre encore un peu plus le fonctionnement du borrow-checker de Rust au travers des types **RefCell** et **Rc**.

RefCell

Le type **RefCell** est utile pour garder un accès mutable sur un objet. Le "borrowing" est alors vérifié au runtime plutôt qu'à la compilation.

Imaginons que vous vouliez dessiner une interface graphique contenant plusieurs vues. Ces vues seront mises dans un layout pour faciliter leur agencement dans la fenêtre. Seulement, on ne peut pas s'amuser à créer un vecteur contenant une liste de références mutables sur un objet, ça ne serait pas pratique du tout !

Run the following code:

```
struct Position {
    x: i32,
    y: i32,
}

impl Position {
    pub fn new() -> Position {
        Position {
            x: 0,
            y: 0,
        }
    }
}

struct Vue {
    pos: Position,
    // plein d'autres champs
}

struct Layout {
    vues: Vec<&mut Vue>,
    layouts: Vec<&mut Layout>,
    pos: Position,
}

impl Layout {
    pub fn update(&mut self) {
        for vue in self.vues {
            vue.pos.x += 1;
        }
        for layout in self.layouts {
            layout.update();
        }
    }
}

fn main() {
    let mut vue1 = Vue { pos: Position::new() };
    let mut vue2 = Vue { pos: Position::new() };
    let mut lay1 = Layout {
        vues: vec!(), layouts: vec!(), pos: Position::new(),
    };
    let mut lay2 = Layout {
        vues: vec!(), layouts: vec!(), pos: Position::new(),
    };

    lay1.vues.push(&mut vue1);
    lay2.layouts.push(&mut lay1);
    lay2.vues.push(&mut vue2);
    lay2.update();
}
```

Si on compile le code précédent, on obtient :

```
<anon>:23:15: 23:23 error: missing lifetime specifier [E0106]
<anon>:23     vues: Vec<&mut Vue>,
                        ^~~~~~~~
<anon>:23:15: 23:23 help: see the detailed explanation for E0106
<anon>:24:18: 24:29 error: missing lifetime specifier [E0106]
<anon>:24     layouts: Vec<&mut Layout>,
                           ^~~~~~~~~~~
<anon>:24:18: 24:29 help: see the detailed explanation for E0106
error: aborting due to 2 previous errors
```

"Arg ! Des lifetimes !"

En effet. Et réussir à faire tourner ce code sans soucis va vite devenir très problématique ! C'est donc là qu'intervient **RefCell**. Il permet de "balader" une référence mutable et de ne la récupérer que lorsque l'on en a besoin avec les méthodes **borrow** et

borrow_mut. Exemple :

Run the following code:
```
use std::cell::RefCell;

struct Position {
    x: i32,
    y: i32,
}

impl Position {
    pub fn new() -> Position {
        Position {
            x: 0,
            y: 0,
        }
    }
}

struct Vue {
    pos: Position,
    // plein d'autres champs
}

struct Layout {
    vues: Vec<RefCell<Vue>>,
    layouts: Vec<RefCell<Layout>>,
    pos: Position,
}

impl Layout {
    pub fn update(&mut self) {
        // Nous voulons "&mut Vue" et pas juste "Vue".
        for vue in &mut self.vues {
            vue.borrow_mut().pos.x += 1;
        }
        // Pareil que pour la boucle précédente.
        for layout in &mut self.layouts {
            layout.borrow_mut().update();
        }
    }
}

fn main() {
    let mut vue1 = Vue { pos: Position::new() };
    let mut vue2 = Vue { pos: Position::new() };
    let mut lay1 = Layout {
        vues: vec!(), layouts: vec!(), pos: Position::new(),
    };
    let mut lay2 = Layout {
        vues: vec!(), layouts: vec!(), pos: Position::new(),
    };

    lay1.vues.push(RefCell::new(vue1));
    lay2.layouts.push(RefCell::new(lay1));
    lay2.vues.push(RefCell::new(vue2));
    lay2.update();
}
```

Rc

Le type **Rc** est un compteur de référence (d'où son nom d'ailleurs, "reference counter"). Exemple :

Run the following code:
```
use std::rc::Rc;

let r = Rc::new(5);
println!("{}", *r);
```

Jusque là, rien de problématique. Maintenant, que se passe-t-il si on clone ce **Rc** ?

Run the following code:
```
use std::rc::Rc;

let r = Rc::new(5);
let r2 = r.clone();
println!("{}", *r2);
```

Rien de particulier, `r` et `r2` pointent vers la même valeur. Et si on modifie la valeur de l'un des deux ?

Run the following code:
```
let mut r = Rc::new("a".to_owned());
println!("1. {:?} = {}", (&*r) as *const String, *r);
let r2 = r.clone();
*Rc::make_mut(&mut r) = "b".to_owned();
println!("2. {:?} = {}", (&*r2) as *const String, *r2);
println!("3. {:?} = {}", (&*r) as *const String, *r);
```

Ce code affichera :

```
1. 0x55769a45c920 = a
2. 0x55769a45c920 = a
3. 0x55769a45ca20 = b
```

Les valeurs de **r** et de **r2** ne sont plus les mêmes et leur pointeur non plus. La raison est la suivante : **make_mut** va vérifier si il y a une autre copie de ce pointeur. Si c'est le cas, pour éviter de faire une opération **unsafe** qui serait de modifier de la mémoire partagée, il va cloner le contenu et créer un nouveau pointeur vers ce contenu dupliqué pour pouvoir le modifier.

Pour éviter qu'une copie ne soit faite lorsque vous manipulez **Rc**, il vous faudra passer par les types **Cell** ou **RefCell** car ils n'ont pas besoin d'être mutable pour pouvoir modifier leur contenu comme expliqué dans ce chapitre. Cela pourra vous être très utile si vous avez des soucis avec des closures notamment.

7. Le multi-threading

Commençons par un exemple tout simple :

Run the following code:
```
use std::thread;

fn main() {
    // On lance le thread.
    let handle = thread::spawn(|| {
        "Salutations depuis un thread !"
    });

    // On attend que le thread termine son travail avant de quitter.
    handle.join().unwrap();
}
```

La fonction **thread::spawn** exécute le code de la closure dans un nouveau thread. On appelle ensuite la méthode **JoinHandle::join** pour attendre la fin de l'exécution du thread.

Jusque-là, on reste dans le classique. Que peut bien apporter Rust ici ? Hé bien essayons maintenant de partager des variables entre les threads :

Run the following code:
```
use std::thread;

let mut data = vec![1u32, 2, 3];
// On va stocker les handlers des threads dans ce `Vec` pour pouvoir
// attendre la fin de leur exécution.
let mut handles = Vec::new();

for i in 0..3 {
    // On lance le thread.
    handles.push(thread::spawn(move || {
        data[i] += 1;
    }));
}

// On attend que les threads aient fini.
for handle in handles {
    handle.join().expect("`join` a échoué");
}
```

Vous devriez obtenir une magnifique erreur :

```
error: capture of moved value: `data`
        data[i] += 1;
```

Le système de propriété rentre ici aussi en jeu. Nous avons trois références mutables sur un même objet et Rust ne le permet pas, c'est aussi simple que cela. Pour contourner ce problème, plusieurs solutions s'offrent à nous :

Mutex

Le type **Mutex** permet d'utiliser une même donnée depuis plusieurs endroits. Une solution naïve serait de les utiliser de cette façon :

Run the following code:

```
use std::thread;
use std::sync::Mutex;

fn main() {
    // On crée notre mutex.
    let mut data = Mutex::new(vec![1u32, 2, 3]);
    let mut handles = Vec::new();

    for i in 0..3 {
        // On locke.
        let data = data.lock().unwrap();
        // On lance le thread.
        handles.push(thread::spawn(move || {
            data[i] += 1;
        }));
    }

    for handle in handles {
        handle.join().expect("`join` a échoué");
    }
}
```

Cependant nous tombons sur un autre problème :

```
<anon>:9:9: 9:22 error: the trait `core::marker::Send` is not implemented for the type `std::syn
<anon>:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
<anon>:9:9: 9:22 note: `std::sync::mutex::MutexGuard<'_, collections::vec::Vec<u32>>` cannot be
<anon>:11         thread::spawn(move || {
                  ^~~~~~~~~~~~~
```

Le trait **Sync** n'est pas implémenté sur le type **MutexGuard** retourné par la méthode **Mutex::lock**. Impossible de partager l'accès aux données de manière sûre ! C'est ici que rentre en jeu le type **Arc** !

Arc

Le type **Arc** est le même type que **Rc**, mais thread-safe car il implémente le trait **Sync**. Corrigeons le code précédent :

Run the following code:

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    // On crée notre mutex,
    let data = Arc::new(Mutex::new(vec![1u32, 2, 3]));
    let mut handles = Vec::new();

    for i in 0..3 {
        // On incrémente le compteur interne de Arc.
        let data = data.clone();
        handles.push(thread::spawn(move || {
            // On locke.
            let mut ret = data.lock();

            // on vérifie qu'il n'y a pas de problème
            match ret {
                Ok(ref mut d) => {
                    // Tout est bon, on peut modifier la donnée en toute sécurité !
                    d[i] += 1;
                }
                Err(e) => {
                    // Une erreur s'est produite.
                    println!("Impossible d'accéder aux données {:?}", e);
                }
            }
        }));
    }

    for handle in handles {
       handle.join().expect("`join` a échoué");
    }
}
```

Nous avons vu comment partager des données entre threads. Il existe cependant une autre façon de faire ça : les channels.

Les channels

On peut se servir des channels pour envoyer des données entre threads. Dans le cas présent, on va s'en servir pour notifier le thread principal qu'un des threads a fini son exécution. On crée un channel via la fonction **mpsc::channel** :

Run the following code:
```
use std::sync::{Arc, Mutex};
use std::thread;
use std::sync::mpsc;

fn main() {
    let data = Arc::new(Mutex::new(0u32));

    // On crée le channel.
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let (data, tx) = (data.clone(), tx.clone());

        thread::spawn(move || {
            let mut data = data.lock().unwrap();
            *data += 1;

            // On envoie le signal de fin du thread.
            tx.send(()).expect("échec de l'envoi des données");
        });
    }

    for _ in 0..10 {
        // On attend le signal de fin du thread.
        rx.recv().expect("échec de réception des données");
    }
    // On va maintenant récupérer la donnée contenue dans `Arc<Mutex<>>`.
    let mutex = Arc::into_inner(data).expect("échec de récupération du contenu de Arc");
    let data = mutex.into_inner().expect("échec de récupération du contenu du mutex");
    println!("data: {}", data);
}
```

Dans ce code, on crée 10 threads qui vont chacun envoyer une donnée dans le channel avant de se terminer. Il nous suffit donc d'attendre d'avoir reçu 10 fois quelque chose pour savoir que tous les threads se sont terminés.

Dans cet exemple, on ne s'en sert que comme d'un signal en envoyant un tuple vide. Il est cependant possible d'envoyer plus que ça, du moment que le type envoyé implémente le trait **Send** :

Run the following code:
```
use std::thread;
use std::sync::mpsc;

fn main() {
    // On crée le channel.
    let (tx, rx) = mpsc::channel();

    for index in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = format!("index : {}", index);

            // On envoie la donnée dans le channel.
            tx.send(answer).expect("échec de l'envoi des données");
        });
    }

    for _ in 0..10 {
        match rx.recv() {
            Ok(data) => println!("Le channel vient de recevoir : {:?}", data),
            Err(e) => println!("Une erreur s'est produite : {:?}", e),
        }
    }
}
```

Ici, nous avons généré des **String** dans nos threads que nous avons ensuite réceptionné et utilisé dans le thread principal.

Les channels sont particulièrement pratiques quand on veut lancer un (ou plusieurs) thread en arrière-plan pour faire des

calculs lourds pour éviter de bloquer le thread principal. Par exemple, si vous faites une interface graphique pour lire des emails, pendant que les requêtes pour récupérer les données sont en cours, vous ne voulez pas bloquer l'interface graphique, donc vous allez faire ça dans un thread et utiliser un channel pour récupérer la donnée quand elle sera arrivée.

Dernier point : il est important de noter que seule la **méthode send** est non-bloquante. Si vous souhaitez ne pas attendre que des données soient disponibles, il vous faudra utiliser la méthode **try_recv**.

Utilisation détournée

Il est possible d'utiliser un thread pour isoler du code de cette façon :

```
Run the following code:
use std::thread;

match thread::spawn(move || {
    panic!("oops!");
}).join() {
    Ok(_) => println!("Tout s'est bien déroulé"),
    Err(e) => println!("Le thread a planté ! Erreur : {:?}", e),
};
```

Cela permet d'exécuter du code qui pourrait paniquer tout en empêchant le programme de s'arrêter si c'est le cas. Cela peut se révéler pratique dans de rares cas.

Les atomiques

Les atomiques sont des types primitifs qu'on peut utiliser pour communiquer entre des threads. À l'exception de **AtomicBool**, ce sont tous des entiers. Chaque opération sur un atomique doit préciser de quel façon on veut que la mémoire soit synchronisée. Prenons un exemple où l'on va augmenter la valeur d'un entier entre plusieurs threads :

```
Run the following code:
use std::sync::atomic::{AtomicU32, Ordering};
use std::sync::Arc;
use std::thread;

fn main() {
    let atomic = Arc::new(AtomicU32::new(0));
    let mut handles = Vec::new();

    for index in 1..10 {
        let atomic = Arc::clone(&atomic);
        handles.push(thread::spawn(move || {
            for _ in 0..index {
                atomic.fetch_add(1, Ordering::Relaxed);
            }
        }));
    }

    for handle in handles {
        handle.join().expect("`join` a échoué");
    }
    println!("On a fait {} itérations", atomic.load(Ordering::Relaxed));
}
```

Dans cet exemple, peu importe l'ordre dans lequel les opérations sur l'atomique sont exécutés donc on a utilisé `Ordering::Relaxed`, mais dans certains cas, il peut être utile de choisir des restrictions différentes. Si tel est votre cas, je vous recommande de jeter un oeil à la documentation de **Ordering** et surtout lire le chapitre sur les atomiques dans le rustnomicon que vous trouverez ici.

Empoisonnement de Mutex

Vous savez maintenant comment partager les données de manière sûre entre des threads. Il reste cependant un petit détail à connaître concernant les mutex : si jamais un thread `panic` alors qu'il a le lock, le **Mutex** sera "empoisonné".

Run the following code:
```
use std::sync::{Arc, Mutex};
use std::thread;

let lock = Arc::new(Mutex::new(0_u32));
let lock2 = lock.clone();

let _ = thread::spawn(move || -> () {
    // On locke.
    let _lock = lock2.lock().unwrap();

    // On lance un panic! alors que le mutex est toujours locké.
    panic!();
}).join();
```

Et maintenant vous vous retrouvez dans l'incapacité de lock de nouveau le **Mutex** dans les autres threads. Il est toutefois possible de "désempoisonner" le mutex :

Run the following code:
```
let mut guard = match lock.lock() {
    Ok(guard) => guard,
    // On récupère les données malgré le fait que le mutex soit lock.
    Err(poisoned) => poisoned.into_inner(),
};

*guard += 1;
```

Autres façons d'utiliser les threads

Il existe un plusieurs crates dans l'écosystème de **Rust** qui permettent d'utiliser les threads de manière bien plus simple. Je vous recommande au moins d'y jeter un coup d'oeil :

- rayon
- crossbeam

8. Le réseau

Je présenterai ici surtout tout ce qui a attrait à des échanges réseaux en mode "connecté", plus simplement appelé TCP. Vous serez ensuite tout à fait en mesure d'utiliser d'autres protocoles réseaux comme l'UDP (qui est un mode "non-connecté") sans trop de problèmes. Le code présenté sera **synchrone**, donc nous ne verrons pas l'**asynchrone** en Rust ici.

Commençons par écrire le code d'un client :

Le client

Commençons par écrire le code d'un client. Pour le moment, nous allons tenter de comprendre le code suivant :

Run the following code:
```
use std::net::TcpStream;

fn main() {
    println!("Tentative de connexion au serveur...");
    match TcpStream::connect("127.0.0.1:1234") {
        Ok(_) => {
            println!("Connexion au serveur réussie !");
        }
        Err(e) => {
            println!("La connexion au serveur a échoué : {}", e);
        }
    }
}
```

Si vous exécutez ce code, vous devriez obtenir l'erreur "Connection refused". Cela signifie tout simplement qu'aucun serveur n'a accepté notre demande de connexion (ce qui est normal puisqu'aucun serveur n'écoute **normalement** sur ce port).

Je pense que ce code peut se passer de commentaire. L'objet intéressant ici est **TcpStream** qui permet de lire et écrire sur un flux réseau. Il implémente les traits **Read** et **Write**, donc n'hésitez pas à regarder ce qu'ils offrent !

Concernant la méthode **connect**, elle prend en paramètre un objet implémentant le trait **ToSocketAddrs**. Les exemples de la documentation vous montrent les différentes façons d'utiliser la méthode **connect**, mais je vous les remets :

Run the following code:
```
let ip = Ipv4Addr::new(127, 0, 0, 1);
let port = 1234;

let tcp_s = TcpStream::connect(SocketAddrV4::new(ip, port));
let tcp_s = TcpStream::connect((ip, port));
let tcp_s = TcpStream::connect(("127.0.0.1", port));
let tcp_s = TcpStream::connect(("localhost", port));
let tcp_s = TcpStream::connect("127.0.0.1:1234");
let tcp_s = TcpStream::connect("localhost:1234");
```

Il est important de noter que "localhost" est la même chose que "127.0.0.1". Nous savons donc maintenant comment nous connecter à un serveur.

Le serveur

Voici maintenant le code du serveur :

```
Run the following code:
use std::net::TcpListener;

fn main() {
    let listener = TcpListener::bind("127.0.0.1:1234").expect("failed to bind");

    println!("En attente d'un client...");
    match listener.accept() {
        Ok((client, addr)) => {
            println!("Nouveau client [adresse : {}]", addr);
        }
        _ => {
            println!("Un client a tenté de se connecter...")
        }
    }
}
```

L'objet **TcpListener** permet de "se mettre en écoute" sur un port donné. La méthode (statique encore une fois !) **bind** spécifie l'adresse (et surtout le port) sur lequel on "écoute". Elle prend le même type de paramètre que la méthode **connect**. Il ne reste ensuite plus qu'à attendre la connexion d'un client avec la méthode **accept**. En cas de réussite, elle renvoie un tuple contenant un **TcpStream** et un **SocketAddr** (l'adresse du client).

Pour tester, lancez d'abord le serveur puis le client. Vous devriez obtenir cet affichage :

```
$ ./server
En attente d'un client...
Nouveau client [adresse : 127.0.0.1:38028]
```

Et côté client :

```
$ ./client
Tentative de connexion au serveur...
Connexion au server réussie !
```

Multi-client

Gérer un seul client, c'est bien, mais qu'en est-il si on veut en gérer plusieurs ? Il vous suffit de boucler sur l'appel de la méthode **accept** et de gérer chaque client dans un thread (c'est une gestion volontairement très simplifiée d'un serveur !). **Rust** fournit aussi la méthode **incoming** qui permet de gérer cela un peu plus élégamment :

```
Run the following code:
let listener = TcpListener::bind("127.0.0.1:1234").unwrap();

println!("En attente d'un client...");
for stream in listener.incoming() {
    match stream {
        Ok(stream) => {
            let adresse = match stream.peer_addr() {
                Ok(addr) => format!("[adresse : {}]", addr),
                Err(_) => "inconnue".to_owned()
            };

            println!("Nouveau client {}", adresse);
        }
        Err(e) => {
            println!("La connexion du client a échoué : {}", e);
        }
    }
    println!("En attente d'un autre client...");
}
```

Pas beaucoup de changements donc. Maintenant comment pourrait-on faire pour gérer plusieurs clients en même temps ? Comme dit un peu au-dessus, les threads semblent être une solution acceptable :

Run the following code:

```
use std::net::{TcpListener, TcpStream};
use std::thread;

fn handle_client(mut stream: TcpStream) {
    // mettre le code de gestion du client ici
}

fn main() {
    let listener = TcpListener::bind("127.0.0.1:1234").unwrap();

    println!("En attente d'un client...");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                let adresse = match stream.peer_addr() {
                    Ok(addr) => format!("[adresse : {}]", addr),
                    Err(_) => "inconnue".to_owned()
                };

                println!("Nouveau client {}", adresse);
                thread::spawn(move|| {
                    handle_client(stream)
                });
            }
            Err(e) => {
                println!("La connexion du client a échoué : {}", e);
            }
        }
        println!("En attente d'un autre client...");
    }
}
```

Rien de bien nouveau.

Gérer la perte de connexion

Épineux problème que voilà ! Comment savoir si le client/serveur auquel vous envoyez des messages est toujours connecté ? Le moyen le plus simple est de lire sur le flux. Il y a alors 2 cas :

- Une erreur est retournée.
- Pas d'erreur, mais le nombre d'octets lus est égal à 0.

À vous de bien gérer ça en vérifiant bien à chaque lecture si tout est ok.

Exemple d'échange de message entre un serveur et un client

Le code qui va suivre permet juste de recevoir un message et d'en renvoyer un. Cela pourra peut-être vous donner des idées pour la suite :

Code complet du serveur :

Run the following code:

```
use std::net::{TcpListener, TcpStream};
use std::io::{Read, Write};
use std::thread;

fn handle_client(mut stream: TcpStream, adresse: &str) {
    let mut msg: Vec<u8> = Vec::new();
    loop {
        let mut buf = &mut [0; 10];

        match stream.read(buf) {
            Ok(received) => {
                // si on a reçu 0 octet, ça veut dire que le client s'est déconnecté
                if received < 1 {
                    println!("Client disconnected {}", adresse);
                    return;
                }
                let mut x = 0;

                for c in buf {
                    // si on a dépassé le nombre d'octets reçus, inutile de continuer
                    if x >= received {
                        break;
                    }
                    x += 1;
                    if *c == '\n' as u8 {
                        println!("message reçu {} : {}",
                            adresse,
                            // on convertit maintenant notre buffer en String
                            String::from_utf8(msg).unwrap()
                        );
                        stream.write(b"ok\n");
                        msg = Vec::new();
                    } else {
                        msg.push(*c);
                    }
                }
            }
            Err(_) => {
                println!("Client disconnected {}", adresse);
                return;
            }
        }
    }
}

fn main() {
    let listener = TcpListener::bind("127.0.0.1:1234").unwrap();

    println!("En attente d'un client...");
    for stream in listener.incoming() {
        match stream {
            Ok(stream) => {
                let adresse = match stream.peer_addr() {
                    Ok(addr) => format!("[adresse : {}]", addr),
                    Err(_) => "inconnue".to_owned()
                };

                println!("Nouveau client {}", adresse);
                thread::spawn(move|| {
                    handle_client(stream, &*adresse)
                });
            }
            Err(e) => {
                println!("La connexion du client a échoué : {}", e);
            }
        }
        println!("En attente d'un autre client...");
    }
}
```

Code complet du client :

Run the following code:

```
use std::net::TcpStream;
use std::io::{Write, Read, stdin, stdout};

fn get_entry() -> String {
    let mut buf = String::new();

    stdin().read_line(&mut buf);
    buf.replace("\n", "").replace("\r", "")
}

fn exchange_with_server(mut stream: TcpStream) {
    let stdout = std::io::stdout();
    let mut io = stdout.lock();
    let mut buf = &mut [0; 3];

    println!("Enter 'quit' when you want to leave");
    loop {
        write!(io, "> ");
        // pour afficher de suite
        io.flush();
        match &*get_entry() {
            "quit" => {
                println!("bye !");
                return;
            }
            line => {
                write!(stream, "{}\n", line);
                match stream.read(buf) {
                    Ok(received) => {
                        if received < 1 {
                            println!("Perte de la connexion avec le serveur");
                            return;
                        }
                    }
                    Err(_) => {
                        println!("Perte de la connexion avec le serveur");
                        return;
                    }
                }
                println!("Réponse du serveur : {:?}", buf);
            }
        }
    }
}

fn main() {
    println!("Tentative de connexion au serveur...");
    match TcpStream::connect("127.0.0.1:1234") {
        Ok(stream) => {
            println!("Connexion au serveur réussie !");
            exchange_with_server(stream);
        }
        Err(e) => {
            println!("La connexion au serveur a échoué : {}", e);
        }
    }
}
```

Voilà ce que ça donne :

```
$ ./server
En attente d'un client...
Nouveau client [adresse : 127.0.0.1:41111]
En attente d'un autre client...
message reçu [adresse : 127.0.0.1:41111] : salutations !
message reçu [adresse : 127.0.0.1:41111] : tout fonctionne ?
```

```
$ ./client
Tentative de connexion au serveur...
Connexion au serveur réussie !
Entrez 'quit' quand vous voulez fermer ce programme
> salutations !
Réponse du serveur : [111, 107, 10]
> tout fonctionne ?
Réponse du serveur : [111, 107, 10]
```

Si vous avez bien compris ce chapitre (ainsi que les précédents), vous ne devriez avoir aucun mal à comprendre ces deux codes. En espérant que cette introduction au réseau en **Rust** vous aura plu !

IV. Annexes

1. Codes annexes

Cette section n'a pas réellement d'intérêt si ce n'est montrer quelques fonctionnalités ou comportements que j'ai trouvé intéressants.

Écrire des nombres différemment

Run the following code:
```
let a = 0_0;
let b = 0--0_0--0;
let c = 0-!0_0-!0;
let d = 0xdeadbeef;
let e = 0x_a_bad_1dea_u64;
```

On peut aussi se servir du _ pour faciliter la lecture des nombres :

Run the following code:
```
let a = 12_u32;
let b = 1_000_000;
```

Toujours plus de parenthèses !

Run the following code:
```
fn tmp() -> Box<FnMut() -> Box<FnMut() -> Box<FnMut() -> Box<FnMut(i32) -> i32>>>> {
    Box::new(|| { Box::new(|| { Box::new(|| { Box::new(|a| { 2 * a }) }) }) })
}

fn main() {
    println!("{}", tmp()()()()(1));
}
```

Utiliser la méthode d'un trait

Vous savez qu'il est possible de définir une méthode dans un trait, mais qu'on est forcé d'implémenter ce trait pour pouvoir l'appeler. Hé bien voici une méthode pour contourner cette limitation :

Run the following code:
```
trait T {
}

impl dyn T {
    fn yop() {
        println!("yop");
    }
}

fn main() {
    <dyn T>::yop()
}
```

Toujours plus vers le fonctionnel avec le slice pattern !

Cette fonctionnalité assez intéressante permet de directement matcher une valeur d'un tableau (et aussi de s'assurer qu'il n'est pas vide) :

Run the following code:
```
fn sum(values: &[i32]) -> i32 {
    match values {
        [head, tail @ ..] => head + sum(tail),
        [] => 0,
    }
}

fn main() {
    println!("Sum: {}", sum(&[1, 2, 3, 4]));
}
```

Une autre façon de faire des boucles infinies

Tout le monde connait `loop` et `while true` je présume. Hé bien il existe d'autres façons de faire des boucles infinies dont notamment :

Run the following code:
```
for idx in 0.. {
    // le code
}
```

Pratique dans le cas où on veut une boucle infinie avec un index pour savoir à quelle itération on en est !

Calculer des factorielles avec un itérateur

Il est possible de calculer des factorielles en utilisant la méthode `Iterator::product` :

Run the following code:
```
fn calculer_factorielle(f: usize) -> usize {
    (1..=f).into_iter().product()
}
```

Façon alambiquée d'itérer sur un Range

Il est possible de renvoyer une valeur avec `break`, ce qui permet dans le code qui suit :

Run the following code:
```
for x in loop { break 0..2 } {
    println!("{x}");
}
```

fonctionne de la même façon que :

Run the following code:
```
for x in 0..2 {
    println!("{x}");
}
```

2. Comparaison avec C++

Ce chapitre n'a pas pour but de savoir quel langage est le meilleur des deux mais plutôt de comparer comment une chose est faite différemment. L'ordre ne revêt pas d'importance particulière non plus. Tous les concepts Rust évoqués ici sont abordés dans ce livre.

Les variables

En Rust comme en C++, on peut déclarer des variables sans déclarer leur type grâce à l'inférence :

```
auto valeur = 10;
```

En Rust :

```
Run the following code:
let valeur = 10;
```

La grosse différence entre les 2 langages, c'est qu'en Rust, quand on déclare une variable, on est **obligés** de l'initialiser (le fameux RAII "Resource Acquisition Is Initialization").

Gestion des erreurs

En C++, il existe 2 façons de gérer des erreurs : les exceptions et les valeurs de retours de fonctions/méthodes. En Rust c'est uniquement les valeurs de retours avec les types **Option** et **Result**.

Pour l'ouverture d'un fichier par exemple :

```
ifstream input_stream;
input_stream.open("file", ios::in);
if (input_stream) {
    // Le fichier a bien été ouvert.
} else {
    // Gestion de l'erreur
}
```

En Rust :

```
Run the following code:
match File::open("file") {
    Ok(file) => {
        // Le fichier a bien été ouvert.
    }
    Err(err) => {
        // Gestion de l'erreur.
    }
}
```

La seule différence ici est donc le typage fort en Rust qui force à matcher sur la valeur de retour pour pouvoir s'en servir.

Gestion de la mémoire

En C++ comme en Rust, on peut allouer de la mémoire dans la heap (le "tas") ou sur la stack (la "pile). Pour la stack, les deux langages sont similaires : le destructeur est appelé quand on le scope courant est détruit. Exemple :

```
void func() {
    string valeur = "baguette";

    // Le scope de `func` est détruit, donc le destructeur de `valeur`
    // est appelé et la mémoire utilisée est libérée.
}
```

En Rust :

```
Run the following code:
fn func() {
    let valeur = String::from("baguette");

    // Le scope de `func` est détruit, donc le destructeur de `valeur`
    // est appelé et la mémoire utilisée est libérée.
}
```

Par contre, pour la heap, les choses se passent différemment. Il est plus rare de s'en servir en Rust (par rapport à la stack) et on le fera avec le type **Box**. La mémoire sera libérée automatiquement quand la **Box** sortira du scope.

En C++ par contre, l'utilisation de la heap est beaucoup plus courante. Pour ce faire, on utilisera le mot-clé `new` et on devra libérer cette mémoire nous-mêmes avec le mot-clé `delete`.

```
int *pointeur = new int;
if (pointeur) {
    *pointeur = 10;
    // On libère la mémoire.
    delete pointeur;
} else {
    // L'allocation de la mémoire a échoué.
}
```

En Rust :

```
Run the following code:
let valeur = Box::new(10);
// La mémoire est supprimée quand on sort du scope courant.
```

Métaprogrammation

La métaprogrammation est plus permissive en C++ car en Rust elle se fera uniquement au travers des traits. Un petit exemple avec une fonction retournant la plus grande valeur entre deux arguments :

```
template <typename T>
T get_max(T x, T y) {
    if (x > y) {
        return x;
    }
    return y;
}
```

En Rust :

```
Run the following code:
fn get_max<T: PartialOrd>(x: T, y: T) -> T {
    if x > y {
        x
    } else {
        y
    }
}
```

Comme vous pouvez le voir, Rust a besoin de plus d'information que C++ car on a besoin de préciser que le type `T` doit

implémenter le trait `PartialOrd` pour pouvoir utiliser l'opérateur > pour que ce code compile.

Les macros

En Rust, les macros sont beaucoup plus puissante : elles reçoivent un flux de tokens et en renvoient un autre, modifiant le code source qui sera ensuite compilé. On peut les considérer comme une extension du compilateur.

Cependant, pour les codes simples, on peut faire des comparaisons avec C++ :

Run the following code:
```
macro_rules! bonjour {
    ($name:literal) => {
        println!(concat!("Bonjour ", $name, " !"));
    }
}

fn main() {
    bonjour!("monde");
}
```

En C++ :

```
#define say_hello(name) std::cout << "Bonjour " << (name) << " !" << std::endl;

int main() {
    say_hello("monde");
    return 0;
}
```

Cependant là où les macros en Rust deviennent vraiment utiles, c'est quand on se sert des proc-macros. Cela peut permettre de faire en sorte que le compilateur de Rust compile un autre langage au moment de la compilation !

Par exemple, la crate rinja transforme des templates Jinja en code Rust pendant la compilation. Et tout ce qu'il y a besoin de faire pour ça, c'est ajouter `derive(Template)` sur notre type :

Run the following code:
```
use rinja::Template;

#[derive(Template)]
// On indique quel template on veut compiler.
#[template(path = "template.html")]
struct Template {
    name: String,
}

fn main() {
    let template = Template { name: String::from("monde") };
    // Le trait `Template` implémente la méthode `render`.
    println!("{}", template.render().unwrap());
}
```

Multi-threading

C++ fournit une API pour des threads et des mutexes dans sa bibliothèque standard, cependant ça reste aux développeurs de s'assurer que leur code ne va pas créer des accès concurrents.

En Rust, c'est aussi fourni par la bibliothèque standard. Cependant, le système de type va tout simplement interdire d'utiliser un type dans un thread si elle n'implémente pas les traits **Sync** et **Send**. Il faudra donc utiliser des types implémentant ces 2 traits.

Prenons un exemple d'un thread qui met à jour une valeur dans un vecteur pendant que le thread principal affiche ce vecteur

:

```
#include <chrono>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

struct Data {
    std::vector<int> data;
    std::mutex mutex;
};

void update_vector(struct Data *v) {
    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        // On locke la donnée.
        std::lock_guard<std::mutex> guard(v->mutex);
        // On la met à jour.
        v->data[2] = v->data[2] + 1;
    }
}

int main() {
    struct Data d = {
        {8, 4, 5, 9},
        std::mutex()
    };

    // On lance le thread.
    std::thread t(update_vector, &d);

    for (int i = 0; i < 10; ++i) {
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
        // On locke la donnée.
        std::lock_guard<std::mutex> guard(d.mutex);
        // On l'affiche.
        for (auto value : d.data) {
            std::cout << value << ",";
        }
        std::cout << std::endl;
    }
    return 0;
}
```

Comme vous pouvez le voir, le mutex n'est pas lié à la donnée, c'est à l'utilisateur de lier les 2.

En Rust cela donne :

Run the following code:

```
use std::sync::{Arc, Mutex};
use std::thread;
use std::time::Duration;

fn main() {
    let mut data = Arc::new(Mutex::new(vec![8, 4, 5, 9]));

    let data_copy = Arc::downgrade(&mut data);
    // On lance le thread.
    thread::spawn(move || {
        let data = data_copy.upgrade().expect("upgrade a échoué");
        for _ in 0..10 {
            thread::sleep(Duration::from_millis(10));
            // On locke la donnée.
            if let Ok(mut content) = data.lock() {
                // On la met à jour.
                content[2] += 1;
            }
        }
    });

    for _ in 0..10 {
        thread::sleep(Duration::from_millis(10));
        // On locke la donnée.
        if let Ok(content) = data.lock() {
            // On l'affiche.
            println!("{:?}", content);
        }
    }
}
```

Pour pouvoir afficher le vecteur tout en le mettant à jour, on est obligés de le garder dans un `Mutex` (qui implémente `Sync`) et dans un `Arc` (qui implémente `Send`).

Déclarations

C++ hérite directement du C de ce point de vue : si on veut utiliser quelque chose, il faut que ce quelque chose soit déclaré avant d'être utilisé. Si cela se trouve dans un autre fichier, il faudra inclure un fichier header décrivant cet objet.

En Rust, on peut se servir d'un item tant que cet item est accessible dans le scope courant parce qu'il y est déclaré ou parce qu'il y est importé.

```
#include <vector>

struct Data {
    // On peut se servir de `vector` parce qu'il a été importé dans le
    // `#include <vector>`.
    std::vector<int> data;
};

int func() {
    // On peut utiliser `Data` parce qu'il est déclaré avant.
    struct Data = {{0, 1}};
}
```

En Rust :

```
Run the following code:
fn func() {
    // `Duration` est importé dans le scope courant donc c'est bon.
    let s = Duration::from_millis(10);
    // `Bonjour` est déclaré dans le scope courant donc c'est bon.
    let s = Bonjour;
}

use std::time::Duration;

struct Bonjour;
```

Gestion des dépendances

C++ ne possède pas d'outil officiel (bien qu'il en existe un certain nombre) pour gérer un projet et ses dépendances. Parmi les outils les plus connus, il y a `Makefile` et `CMake`. Le premier ne gère pas les dépendances tandis que le deuxième fournit les outils pour, mais cela reste au développeur de gérer ça.

En Rust, il y a `cargo`. Il permet de gérer le build ainsi que les dépendances d'un projet, que ce soit à partir d'un dépôt `git` ou bien de crates.io (qui centralise toutes les crates publiées).

Outils autour du langage

C++ a beaucoup d'outils mais rien qui soit officiel. Cependant, `clang` fournit un linter et même un formateur de code. Mais de manière générale, si on veut faire quelque chose en C++, il faudra chercher et installer soi-même l'outil.

En Rust, tout tourne autour de `cargo`. Il y a un linter officiel (`clippy`), un formateur de code (`rustfmt`), un outil pour générer la documentation (`rustdoc`), un outil pour garder Rust et ses outils à jour (`rustup`), etc. Les outils étant des crates, ils sont disponibles sur crates.io et donc installables avec `cargo`.

Cross-compilation

La cross-compilation est un sujet complexe en C++ et il n'y a rien de standard.

En Rust, `cargo` et `rustup` simplifient grandement les choses. Par exemple pour compiler depuis Linux vers Android :

```
# On installe la target Android.
rustup target add arm-linux-androideabi
# On compile vers cette target.
cargo build --target=arm-linux-androideabi
```