# Tutoriel Rust

Rust est un nouveau langage développé au sein de la fondation Mozilla 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

## 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).

Le développement du langage, initié par Graydon Hoare, est opéré depuis 2009 par la fondation Mozilla, ainsi que par la communauté des développeurs Rust très présente sur Github. Pour suivre ce tutoriel, il est fortement recommandé d'avoir déjà développé dans au moins un autre langage (C, C++, Java, javascript, Python, etc...) car je ne passerai que très brièvement sur les bases. Ses points forts 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 :

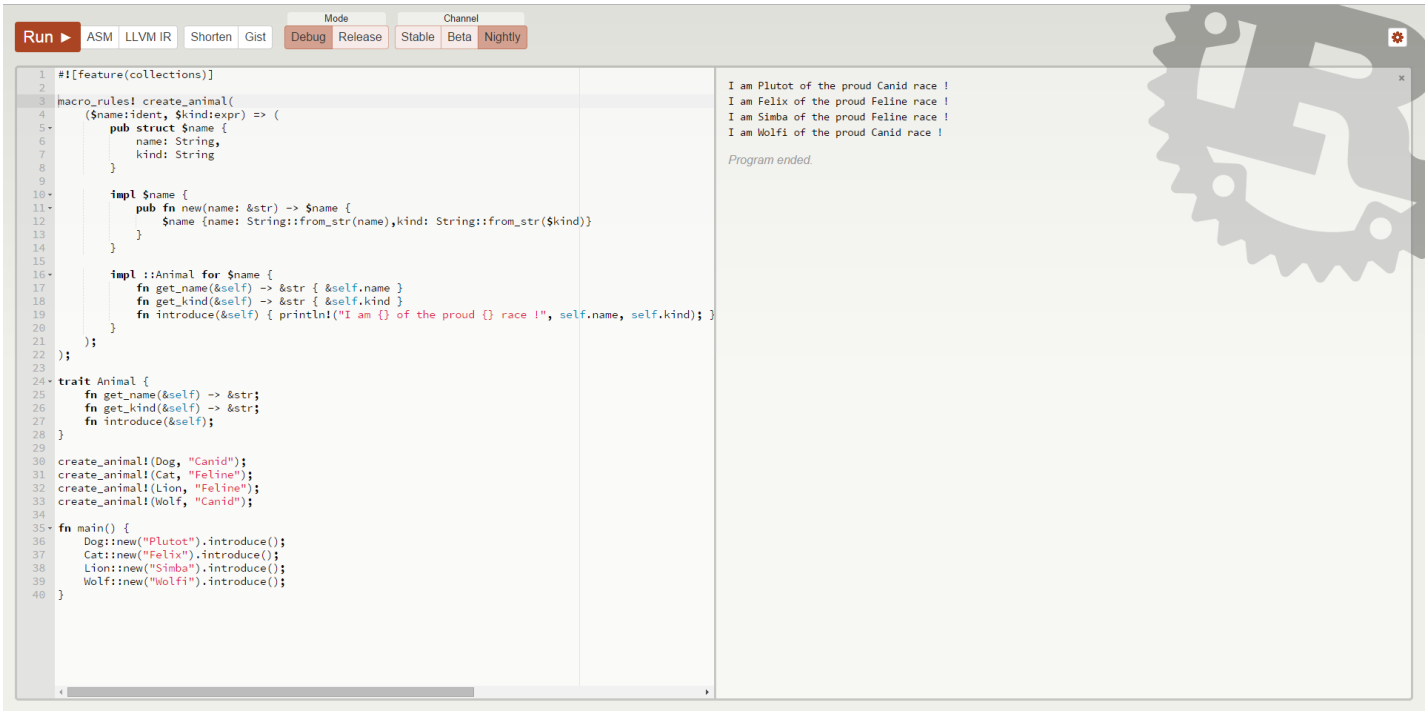
- Le site internet : rust-lang.org
- La documentation (toujours utile d'avoir ça sous la main !)
- Le dépôt Github (pour voir le code source)
- Le rustbook (le "cours" officiel, en anglais)
- Le reddit (pour poser une question)
- L'irc (pour obtenir de l'aide, en anglais). Le channel français est #rust-fr.

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 http://www.tutorialspoint.com/compile_rust_online.php
  qui permet d'éditer, compiler et exécuter des projets complets (répartis sur plusieurs fichiers) tout en proposant des outils d'import et d'export.
- Soit un éditeur de texte. Pour le moment, il n'existe pas d'IDE dédié pour Rust, il va falloir attendre encore un peu avant d'en voir un émerger.

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 !

## Le compilateur de Rust

Si vous ne souhaitez pas utiliser l'éditeur Rust en ligne, il va vous falloir télécharger le compilateur de Rust disponible ici, puis l'installer.

Nous pouvons maintenant commencer à nous intéresser au langage Rust à proprement parler !

# 3. Premier programme

Pour pouvoir tester tout ce que nous avons mis en oeuvre dans le chapitre précédent, je vous propose d'écrire votre premier programme en Rust :

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

## Si vous n'utilisez pas play.rust-lang :

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
```

## Si vous utilisez play.rust-lang :

Appuyez tout simplement sur le bouton "Run".

Vous savez maintenant comment compiler et exécuter vos programmes.

# 4. Variables

La première chose à savoir en Rust est que les variables sont toutes 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 !
```

Maintenant voyons comment fonctionnent les **types** en Rust. Ici, rien de nouveau, on a toujours des entiers, des flottants, des strings, etc... La seule différence viendra de leur syntaxe. Par exemple, pour déclarer un entier de 32 bits, vous ferez :

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

Sachez aussi que le compilateur de Rust utilise **l'inférence de type**. En gros, on n'est pas obligé de déclarer le type d'une variable, il peut généralement le déduire tout seul. Exemple :

```
Run the following code:
let i = 0; // donc c'est un entier visiblement
let max = 10i32;

if i < max { // max est un i32, donc le compilateur en déduit que i en est un aussi
    println!("i est inférieur à max !");
}
```

Donc pour résumer, voici une petite liste des différents types de base disponibles :

- i8 : un entier signé de 8 bits
- i16
- i32
- i64
- u8 : un entier non-signé de 8 bits
- u16
- u32
- u64
- f32 : un nombre flottant de 32 bits
- f64 : un nombre flottant de 64 bits
- String
- 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;
```

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

## Les slices

Pour faire simple, une slice représente un morceau de tableau. Pour ceux qui auraient fait du C/C++, c'est tout simplement un pointeur et une taille. Exemple :

Run the following code:
```
let tab = &[0, 1, 2]; // tab est une slice contenant 0, 1 et 2

println!("{:?}", tab); // ça affichera "[0, 1, 2]"
let s = &tab[1..]; // s est maintenant une slice commençant à partir du 2e élément de tab
println!("{:?}", s); // ça affichera "[1, 2]"
```

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

Run the following code:
```
let mut v: Vec<u8> = Vec::new();

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

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

Voilà qui conclut ce chapitre.

# 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 !");
}
```

Vous aurez noté que je n'ai pas mis de parenthèses (( et )) autour des conditions : elles sont superflues en Rust. Cependant, elles sont toujours 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 !

En bref : pas de parenthèses autour de la condition mais accolades obligatoires autour du bloc de la condition.

Je vais profiter de ce chapitre pour aborder le **pattern matching**.

## Pattern matching

Définition wikipédia :

"Le filtrage par motif, en anglais pattern matching, 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 ça, on peut comparer ce que l'on appelle des **expressions** de manière plus intuitive. Ceux ayant déjà utilisé des langages fonctionnels ne devraient pas se sentir dépaysés. Comme un code vaut souvent mieux que de longues explications :

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

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

Ici ça affichera donc "anglais".

Comme vous vous en doutez, on peut s'en servir sur n'importe quel type de variable. Après tout, il sert à comparer des **expressions**, vous pouvez très bien matcher sur un **i32** ou un **f64** si vous en avez besoin.

Concernant le _, il signifie "toutes les autres expressions". C'est en quelque sorte le **else** du pattern matching (il fonctionne de la même manière que le **default** d'un switch C/C++/Java). Cependant, il est obligatoire de le mettre si toutes les expressions possibles n'ont pas été testées ! Dans le cas présent, il est impossible de tester toutes les strings existantes, on met donc _ à la fin. Si on teste un booléen, on pourra faire :

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

match b {
    true => {
        // faire quelque chose
    }
    false => {
        // faire autre chose
    }
}
```

Car il n'y a que deux valeurs possibles et qu'elles ont toutes les deux été testées ! Un autre exemple en utilisant un **i32** :

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

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

C'est le moment où vous vous dites quelque chose du genre "mais c'est nul ! On va pas s'amuser à écrire toutes les valeurs en dessous de 18 pour voir s'il est majeur !". Sachez que vous avez tout à fait raison. Dans le cas ci-dessus, le mieux serait d'écrire :

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

match age {
    tmp if tmp > 17 => {
        println!("majeur !");
    }
    _ => {
        println!("mineur !");
    }
}
```

Et là, vous vous demandez sans doute : "mais d'où il sort ce **tmp** ?!". C'est une variable créée (temporairement) à l'intérieur du bloc du match contenant la valeur de l'expression évaluée (la variable **age** en l'occurence, donc 17). Elle nous permet d'ajouter une condition à notre pattern matching afin d'affiner les résultats ! On peut donc ajouter des && ou des || selon les besoins.

Un petit détail avant de passer à la suite :

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

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

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

Tout comme pour les if/else 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. 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");
}
```

## 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)
};
```

À noter, dans le cas d'un `for i in 10..100 { println!("{}", i); }`, **i** prendra une valeur allant de 10 à 99 inclus.

Pratique ! 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 9 et 101", x),
    x => println!("{} n'est pas entre 9 et 101", x)
};
```

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

Run the following code:
```
match une_variable {
    "jambon" | "poisson" | "oeuf" => 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". Dans le premier cas, si `une_variable` vaut "jambon", "poisson" ou "oeuf", le match rentrera dans cette condition, et ainsi de suite.

Voilà qui clôt ce chapitre sur les conditions et le pattern matching. Encore une fois n'hésitez pas à revenir sur des points que vous n'êtes pas sûr d'avoir parfaitement compris. Ce que nous voyons actuellement est vraiment la base de ce langage. Si quelque chose n'est pas parfaitement maîtrisé, vous risquez d'avoir du mal à comprendre la suite.

# 6. if let / while let

C'est la suite directe du cours précédent. Ce chapitre est court mais très pratique.

## 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
}
```

Hé 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` :

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 précédent mais il est possible d'être plus précis dans le pattern matching. Par-exemple :

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

if let Some(10) = x {
    // faire quelque chose
} 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);
}
```

# 7. Les boucles

Nous les avons vu très rapidement à la fin du chapitre précédent donc approfondissons le sujet :

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

## while

Comme dans les autre langages, elle 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 plusieurs choses :

- Vous noterez encore une fois qu'il n'y a pas de parenthèse autour de la condition !
- Tout comme pour les conditions, les accolades sont encore une fois **obligatoires** !

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

## loop

Je pense que vous vous demandez tous : "mais à quoi ça peut bien nous servir ?". Prenons un exemple : un jeu-vidéo. L'affichage doit continuer en permanence jusqu'à ce que l'on quitte. Plutôt que d'écrire :

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

// ou

let mut end = false;

while end == false {
    //...
}
```

On écrira tout simplement :

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

Maintenant vous vous dites sans doute : "Ok, mais comment on l'arrête cette boucle ?". Tout simplement avec le mot-clé **break** . Reprenons notre exemple du début :

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

loop {
    println!("bonjour !");
    i += 1;
    if i > 10 {
        break;
    }
}
```

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 {
    println!("Toujours là !");
    let mut i = 0i32;

    loop {
        println!("sous-boucle !");
        i += 1;
        if i > 2 {
            break; // et on revient dans la boucle précédente
        }
    }
}
```

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

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

        loop {
            println!("sous-boucle !");
            i += 1;
            if i > 2 {
                return; // on quitte la fonction main et donc le programme se termine
            }
        }
    }
}
```

## for

La boucle for est un peu plus complexe que les deux précédentes. Elle ne fonctionne qu'avec des objets implémentant le trait

**IntoIterator**. Vous ne savez pas ce qu'est un **trait** ? Pour le moment ce n'est pas important, nous y reviendrons plus tard. Prenons maintenant des exemples de comment fonctionne la boucle **for** :

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
```

Comme vous l'aurez compris la variable **i**, créée pour la boucle **for**, prendra successivement toutes les valeurs allant de 0 à 9 puis la boucle s'arrêtera toute seule.

Certains d'entre vous doivent se demander si je n'ai pas menti en disant que la boucle **for** ne s'utilisait que sur des objets en voyant ce "0..10". Hé bien sachez que non, ce "0..10" est considéré comme un objet de type **Range** qui implémente le trait **IntoIterator**. J'insiste sur le fait que c'est tout à fait normal si vous ne comprenez pas toutes mes explications pour l'instant !

Prenons un deuxième exemple :

Run the following code:
```
let v = vec!(1, 4, 5, 10, 6);

for value in v {
    println!("{}", value);
}
```

Ce qui va afficher :

```
1
4
5
10
6
```

## Énumération

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

Run the following code:
```
for (i, j) in (5..10).enumerate() {
    println!("i = {} et j = {}", i, j);
}
```

Ce qui affichera :

```
i = 0 et j = 5
i = 1 et j = 6
i = 2 et j = 7
i = 3 et j = 8
i = 4 et j = 9
```

**i** vaut donc le nombre d'itérations effectuées à l'intérieur de la boucle tandis que **j** prend successivement les valeur range. Autre exemple :

Run the following code:
```
let v = vec!("a", "b", "c", "d");

for (i, value) in v.iter().enumerate() {
    println!("i = {} et value = \"{}\"", i, value);
}
```

# Les boucles nommées

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

Run the following code:
```
'outer: for x in 0..10 {
    'inner: for y in 0..10 {
        if x % 2 == 0 { continue 'outer; } // on continue la boucle sur x
        if y % 2 == 0 { continue 'inner; } // on continue la boucle sur y
        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 {
            if x > 3 { break 'global; } // on arrête la boucle qui s'appelle global
            if x % 2 == 0 { continue 'outer; } // on continue la boucle sur x
            if y % 2 == 0 { continue 'inner; } // on continue la boucle sur y
            println!("x: {}, y: {}", x, y);
        }
    }
}
```

Encore une fois, je vous invite à tester pour bien comprendre comment tout ça fonctionne. Quand ce sera bon, il sera temps de passer aux **fonctions** !

# 8. 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 beaucoup 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 variables de types i32 en paramètre et retourne un i32. Rien de très différent de ce que vous connaissez déjà donc. Maintenant un exemple d'utilisation :

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

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

Ce qui affiche :

```
1 + 2 = 3
```

Ceux ayant bien compris le chapitre précédent se demanderont sans doute : "Tu nous avais parlé du mot clé **return** mais tu ne t'en sers pas ! Comment les valeurs ont-elles été retournées ? D'ailleurs il manque pas un point-virgule là ?".

Le fait de ne pas mettre de point-virgule signifie que l'on veut que "nb1 + nb2" soit interprété comme une expression. Cependant on pourrait tout aussi bien écrire :

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

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 > n2 {
        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 trouvée 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 fichier "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
```

Enfin, une dernière chose : 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 !");
}
```

"Le type **()** ? C'est une sorte de **null** ?"
Oui... Et non. En **Rust**, c'est un tuple vide. Son équivalent le plus proche en C/C++ est le type **void**.

Voilà, qui clôture ce chapitre. Il est maintenant temps de s'attaquer aux expressions !

# 9. 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, son équivalent le plus proche en C/C++ est le type **void** comme je vous l'ai 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'elle ne peut pas se terminer par un point-virgule. Démonstration :

Run the following code:
```
let var: i32 = 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 `i32`.

Je présume que vous vous dites : "Encore ce '()' ?!". Hé oui. Je pense à présent que vous avez un petit 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. Un dernier exemple 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
    }
}
```

Il est temps de passer à la suite !

# 10. 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) => {
        // Okay, 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!("{}", 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 unwrap !
} else {
    // il y a eu une erreur, unwrap 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, **NE L'UTILISEZ JAMAIS** !!

## 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 costumisable 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 trucs
    salon: Option<Salon>,
}

impl Vaisseau {
    pub fn new() -> Vaisseau {
        Vaisseau {
            // on initialise le reste
            salon: None, // on n'a pas de salon
        }
    }
}

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

Donc pour le moment, on n'a pas de salon. Maintenant nous en rajoutons 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());

let salon =  vaisseau.salon.expect("pas de salon"); // je ne le recommande pas !
```

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 unwrap !
} 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 a une erreur irrécupérable. Elle est très simple d'utilisation :

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

Et c'est tout.

## try!

Et maintenant voici la macro **try!** ! Elle permet de se "passer" du pattern matching en retournant directement le résultat en cas de réussite ou bien en quittant la fonction en cas d'erreur. Exemple :

```
Run the following code:
let mut fichier = try!(File::create("fichier.txt"));
try!(fichier.write_all("Test"));
```

La fonction qui utilise cette macro doit obligatoirent retourner **Result**.

À noter qu'il y a maintenant la possibilité d'utiliser **?**. Cet opérateur fonctionne exactement de la même façon que **try!** si ce n'est que c'est plus court à écrire :

```
Run the following code:
File::create("fichier.txt")?.write_all("Test")?;
```

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

# 11. Cargo

Rust possède un gestionnaire de paquets : Cargo. Il permet 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 project avec **Cargo**, rien de plus facile :

```
> cargo new mon_nouveau_project
```

Un nouveau dossier s'appelant **mon_nouveau_project** sera crée :

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

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

```
[package]
name = "mon_nouveau_project"
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
```

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

```
> ./target/debug/mon_nouveau_project
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_project v0.0.1 (file:///path/to/project/mon_nouveau_project)
    Running `target/debug/mon_nouveau_project`
Début du projet
```

Et voilà !

Par défaut, **cargo** compile en mode **debug**. Les performances sont **BEAUCOUP** plus faibles qu'en mode **release**, faites 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, soit vous pourrez donner son adresse github.

Par exemple, vous voulez utiliser la bibliothèque GTK+, elle est disponible sur crates.io (ici) donc pas de souci :

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

[dependencies]
gtk = "0.3.0"
```

Nous avons donc ajouté **gtk** 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 bibliothèques 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 de ce sujet.

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

```
[dependencies]
gtk = "0.0.2"
```

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 bibliothèque :

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 bibliothèque 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_project"
version = "0.0.1"
authors = ["Votre nom <vous@exemple.com>"]

[dependencies.gtk]
git = "http://github.com/gtk-rs/gtk"
```

Ici nous avons indiqué que la bibliothèque **gtk** 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.gtk]
path = "chemin/vers/gtk"
```

Voici en gros à quoi ressemblerait un *gros* fichier cargo :

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

[dependencies.gtk]
git = "http://github.com/gtk-rs/gtk"

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

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

## Publier une bibliothèque sur crates.io

Vous avez fait une bibliothèque 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 bibliothèque :

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

- **license** : Licence(s) de votre bibliothèque. 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 votre (relatif au fichier **Cargo.toml**).

Je vais vous donner ici le contenu du fichier **Cargo.toml** de la bibliothèque **GTK** pour que vous ayez un exemple :

```
[package]
name = "gtk"
version = "0.0.2"
authors = ["The Gtk-rs Project Developers"]

description = "Rust bindings for the GTK+ library"
repository = "http://github.com/rust-gnome/gtk"
license = "MIT"
homepage = "http://github.com/rust-gnome/gtk"
documentation = "http://github.com/rust-gnome/gtk"

readme = "README.md"

keywords = ["gtk", "gnome", "GUI"]

[lib]
name = "gtk"

[features]
default = ["gtk_3_6"]
gtk_3_4 = ["gtk-sys/gtk_3_4", "gdk/gdk_3_4"]
gtk_3_6 = ["gtk-sys/gtk_3_6", "gdk/gdk_3_6", "gtk_3_4"]
gtk_3_8 = ["gtk-sys/gtk_3_8", "gdk/gdk_3_8", "gtk_3_6"]
gtk_3_10 = ["gtk-sys/gtk_3_10", "gdk/gdk_3_10", "cairo-rs/cairo_1_12", "gtk_3_8"]
gtk_3_12 = ["gtk-sys/gtk_3_12", "gdk/gdk_3_12", "gtk_3_10"]
gtk_3_14 = ["gtk-sys/gtk_3_14", "gdk/gdk_3_14", "gtk_3_12"]

[dependencies]
libc = "0.1"
gtk-sys = "^0"
glib = "^0"
glib-sys = "^0"
gdk-sys = "^0"
gdk = "^0"
pango-sys = "^0"
pango = "^0"
cairo-sys-rs = "^0"
cairo-rs = "^0"
```

Voilà ! Comme vous pouvez le voir, il y a aussi une option `[features]`. Elle permet dans le cas de **GTK** de faire une compilation conditionnelle dépendant de la version que vous possédez sur votre ordinateur. Vous ne pouvez par exemple pas utiliser du code de la version 3.12 si vous avez une version 3.4.

Nous voilà enfin à la dernière étape : **publier la bibliothèque**. **ATTENTION : une bibliothèque 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 bibliothèque sera publiée est celui donné par la metadonnée **name** :

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

Si une bibliothèque 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 bibliothèque 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).

# 12. 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 bibliothèque **time**. Maintenant dans votre fichier principal (celui que vous avez indiqué à Cargo), ajoutez :

Run the following code:
```
extern crate time;
```

Pour appeler une fonction depuis la bibliothèque, 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;
```

Voilà qui conclut ce (bref) chapitre !

# 13. 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.

Relativement simple. Je pense que vous commencez déjà à voir comment tout ça va s'articuler. 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 !
```

(Je sais, je suis vraiment trop fort à ce jeu.)

"Mais comment fait-on pour générer un nombre aléatoire ?"

Bonne question ! On va utiliser la bibliothèque externe rand. Ajoutez-la comme dépendance dans votre fichier **Cargo.toml** et ensuite importez-la dans votre fichier principal. Maintenant, pour générer un nombre il vous suffira de faire :

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

let nombre_aleatoire = rand::thread_rng().gen_range(1, 101);
```

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 !

Maintenant vous savez ce que vous avez à faire. 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;

fn recuperer_entree_utilisateur() -> Option<isize> { // elle ne prend rien en entrée et retourne
    let mut entree = String::new();

    match io::stdin().read_line(&mut entree) { // on récupère ce qu'a entré l'utilisateur dans l
        Ok(_) => { // tout s'est bien passé, on peut convertir la String en entier
            match isize::from_str(&entree.trim()) { // la méthode trim enlève tous les caractère
                Ok(nombre) => Some(nombre), // tout s'est bien déroulé, on retourne donc le nomb
                Err(_) => { // si jamais la conversion échoue (si l'utilisateur n'a pas rentré u
                    println!("Veuillez entrer un nombre valide !");
                    None
                }
            }
        },
        _ => { // une erreur s'est produite, on doit avertir l'utilisateur !
            println!("Erreur lors de la récupération de la saisie...");
            None
        }
    }
}
```

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

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

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

    println!("Génération du nombre...");
    let nombre_aleatoire = rand::thread_rng().gen_range(1, 101);
    println!("C'est parti !");
    while tentative > 0 {
        print!("Entrez un nombre : "); // on ne veut pas de retour à la ligne !
        io::stdout().flush(); // si on n'utilise pas cette méthode, on ne verra pas l'affichage
        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 main et le tour est joué !

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

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

Run the following code:

```
extern crate rand;

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();

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

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

    println!("Génération du nombre...");
    let nombre_aleatoire = rand::thread_rng().gen_range(1, 101);
    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() == true {
        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 venir en parler sur #rust-fr ou directement sur github en ouvrant une issue.

## Améliorations

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

- Un mode 2 joueurs.
- 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" hexadecimal (base 16)
println!("{:X}", nombre); // affiche le nombre en "grand" hexadecimal (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 !).

### 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 et 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:
```
use std::io::Write; // on importe le trait Write...

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

Et oui, encore une autre macro ! Ne vous en faites pas, c'est la dernière (pour l'instant... *sourire machiavélique*) ! 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 ?!". Avant d'aborder cette partie, il faut que je vous parle des structures en Rust.

# 2. Les structures

Comme certains d'entre vous vont s'en rendre compte, elles sont à la fois très ressemblantes et très différentes de ce que vous pourriez croiser dans d'autres langages. Ce chapitre est assez lourd 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 tuples.
- Les structures unitaires (on dit aussi structure opaque).
- Les structures "classiques" (comme en C).
- Les structures tuples (un mélange entre les tuples et les structures "classiques").

Exemple de déclaration pour chacune d'entre elles :

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

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

// Une structure unitaire
struct Unitaire;

// Une structure tuple
struct StructureTuple(usize);
```

Maintenant voyons comment on les instancie :

```
Run the following code:
let t = Tuple(0, 2, false); // Le tuple

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

let st = StructureTuple(1); // La structure tuple

let u = Unitaire; // La structure unitaire
```

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 bien de les suivre lorsqu'on le peut, ça 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.

Avec les exemples que je vous ai donné au-dessus, je pense que certains d'entre vous se demande à quoi peut bien servir la "structure tuple" ? Hé bien pas à grand chose dans la majorité des cas, mais il y en a un où elle est très utile :

Run the following code:
```
struct Distance(isize);

let distance = Distance(23);

let Distance(longueur) = distance;
println!("La distance est {}", longueur);
```

Elle permet de "masquer" un type, ce qui peut se révéler pratique dans certains cas.

"D'accord. Et la structure unitaire ?"

Celle-là par-contre, vous risquez de ne pas vous en servir avant longtemps voire peut-être même jamais. Elle est utilisée en général quand on ne sait pas ce qu'elle contient.

Maintenant je présume que vous vous demandez : "Comment peut-on utiliser une structure sans savoir ce qu'elle contient ? o_O". Quand on porte une bibliothèque depuis un autre langage par exemple. Une fonction peut vous retourner un type dont vous n'avez pas accès aux champs mais qui est utilisé partout (les structures présentes dans la bibliothèque GTK+ en sont un très bon exemple).

## Déstructuration

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

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),
}
```

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 };

match origin {
    Point { y, .. } => println!("(..,{})", y),
}
```

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

Maintenant que les explications sont faites, voyons voir 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 structure lui-même. Exemple :

Run the following code:
```
pub struct Distance {
    // Ce champs 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 dites ? Après tout, on irait aussi vite de le faire nous-même ! Dans le cas présent, il n'y en a pas beaucoup, c'est vrai. 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(10);

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

Une chose importante à noter est qu'une fonction membre 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** : c'est la durée de vie de l'objet. Nous aborderons cela 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 };
let mut point2 = Point3d { y: 1, .. point }; // et ici on prend x et z de 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** à 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 pour plus d'explications, il va vous falloir lire le chapitre suivant !

# 3. Les traits

Commençons par donner une rapide définition : un trait est un ensemble de méthodes que l'objet sur lequel il est appliqué doit implémenter.

Dans le chapitre précédent, 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. On en retrouve même sur de simples types 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 implémenté "par défaut". Si ce n'est pas le cas, vous devez importer un trait pour utiliser les fonctions qui y sont associées. Exemple :

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

println!("{}", f64::from_str("3.6").unwrap());
```

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 {
    pub 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 ? :D

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 été inventés pour permettre de faire du polymorphisme. Un exemple (encore un !) :

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 !

## Aller plus loin

J'en profite maintenant pour vous montrer quelques utilisation de traits comme **Range** (que l'on avait déjà rapidement abordé dans le chapitre des boucles). 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:
```
let v: &[u8] = &[0; 10]; // on crée un slice contenant 10 '\0'

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.

# 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

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 trait demandé.

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());
}
```

C'est pas trop génial ?

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);
```

Pas mal hein ? Et pourtant... Ce code ne compile pas !

Pourquoi ?!

Tout simplement parce que les traits ne peuvent prendre en compte, dans les méthodes par défaut, le travail sur des valeurs contenues dans l'objet pris en paramètre (référencé dans cet exemple par **self**).

Imaginez une baleine et un chien. Tous deux sont des animaux, pas vrai ? Pourtant, ils ont très peu de points communs...

Ainsi, chacun aura des caractéristiques que l'autre pourrait ne pas avoir (la couleur des poils par exemple). Programmer une fonction par défaut sur un attribut qu'un objet implémentant le trait ne possède pas pourrait poser problème !

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 réimplém
    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, à chaque structure héritant d'un trait, la réimplémenter... Cela dit, les macros pourraient grandement faciliter cette étape laborieuse, mais nous verrons cela plus tard.

## 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** permet d'ajouter des clauses à la généricité et est plus lisible sur les fonctions/types prenant beaucoup de paramètres génériques :

Run the following code:
```
use std::fmt::Debug;

trait UnTrait<A, B> {
    fn copy_a(&mut self, &A);
    fn clone_b(&mut self, &B);
}

struct UneStruct<A, B> {
    a: A,
    b: B,
}

impl<A, B> UnTrait<A, B> for UneStruct<A, B>
    where A: Copy + Debug,
          B: Clone {
    fn copy_a(&mut self, a: &A) {
        self.a = *a;
        println!("copy_a: {:?}", a);
    }

    fn clone_b(&mut self, b: &B) {
        self.b = b.clone();
    }
}
```

# 5. Propriété (ou ownership)

Jusqu'à présent, de temps à autres, 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 en devenir le propriétaire. 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);
```

Il est cependant 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<
    (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 **Copy** 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 !

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 doit 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 f() {
    let mut v = 10i32; // on crée une variable

    v += 12; // on fait des opérations dessus
    v *= 2;
    // ...

    // quand on sort de la fonction, v n'existe plus
}

fn main() {
    let v: i32 = 12; // cette variable n'a rien à voir avec celle dans la fonction f
    let v2: f32 = 0;

    f();
    // on quitte la fonction, v et v2 n'existent plus
}
```

Ainsi, ce code devient invalide :

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

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

Ici, le compilateur vous dira que la variable **x** ne vit pas assez longtemps. **x** ayant été déclarée après **reference**, 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.

# 6. Durée de vie (ou lifetime)

Il existe plusieurs types de durée de vie. Jusqu'à présent, nous n'avons vu que les plus basique mais sachez qu'il en existe encore deux autres :

- Les durées de vie statiques.
- Les durées de vie associées.

Les durées de vie statiques permettent aux références de référencer des variables statiques ou du contenu "constant" :

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

// avec du contenu constant
let variable_const: &'static str = "Ceci est une str constante !";
```

Rien de bien difficile ici, l'autre est un peu plus complexe mais aussi moins visible. Imaginons que vous écriviez une classe dont l'une des variables membre devait être modifiée à l'extérieur de la structure. Vous vous contentez de renvoyer une "&mut self.ma_variable", je me trompe ? 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 'z, peu importe). Ici, on retourne donc une référence avec une durée de vie **'a** sur une variable.

Je tenais à vous parler de ce dernier point pour que vous compreniez bien comment tout cela fonctionne, le premier code sans l'ajout des paramètres de durée de vie était tout à fait fonctionnel, mais moins clair.

# 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 le trait **Deref** en **Rust**. Dans l'exemple précédent, le trait est implémenté sur le type `&mut i32`. Cependant, il est aussi possible de faire :

Run the following code:
```
let x = String::new();
let deref_x = *x; // ce qui renvoie une erreur car le type str n'implémente pas le trait Sized
```

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:
```
use std::ops::Deref; // on importe le trait

struct UneStruct {
    value: u32
}

impl Deref for UneStruct {
    type Target = u32; // pour préciser quel type on retourne !

    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 aussi ce trait sans le savoir quand vous faites :

Run the following code:
```
fn affiche_la_str(s: &str) { // on obtient une &str
    println!("affichage : {}", s);
}

let x = "toto".to_owned(); // on a donc une String("toto")
affiche_la_str(&x); // on passe une &String
```

Normalement, vous devriez vous demander : "Pourquoi si on passe `&String` on obtient `&str` ?!". Hé bien sachez que Rust implémente un système **d'auto-déréférencement**. Ça 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 l'appelle
- `&(*(String.deref()))`
- `&(*(&str))`
- `&(str)`
- `&str`

Et voilà, le compilateur a bien le type attendu !

# 8. Sized et String vs str

Ce chapitre approfondi ce dont je vous ai déjà parlé dans un chapitre précédent, 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 en mémoire une adresse 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:
let x: [i32] = [0, 1, 2];
```

Ce qui donne :

```
error: mismatched types:
expected `[i32]`,
   found `[i32; 3]`
# ...
error: the trait `core::marker::Sized` is not implemented for the type `[i32]` [E0277]
```

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. Du coup 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 :

```
Run the following code:
let x: &[i32] = &[0, 1, 2];
let x = "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 longueur), elles contiennent aussi une capacité qui représente la quantité de mémoire réservée (mais pas

nécessairement utilisée).

Pour résumer un peu le tout, **String** est une structure permettant de modifier le contenu d'une "vue" constante représentée par le type **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";
let y: String = x.to_owned(); // on aurait aussi pu utiliser String::from
let z: &str = &y;
```

## Vec vs slice

C'est plus ou moins le même fonctionnement : le type **Vec** permet de modifier le contenu d'une vue (non constante) représentée par les **slice**. Exemple :

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

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

# 9. 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 dans ce chapitre (mais ça ne fait jamais de mal après tout !).

Pour ceux qui n'ont jamais utilisé de closures, c'est une fonction anonyme qui capture son 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| 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| 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 l'appelle**.

Mais à quoi ça peut bien servir ? Imaginons que vous avez une super interface graphique et que vous voulez effectuer une action lors d'un événement, disons lorsque le bouton est cliqué. Hé bien 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);
});
```

Facile, non ? Cela dit, vous pourriez aussi vous servir des closures pour trier un vecteur d'objets ou d'autres choses similaires.

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)
}
```

Nous avons donc vu les bases des closures. C'est une partie importantes, 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".

# 10. 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 de **internet.rs**, il vous suffit de faire comme ceci :

Run the following code:
```
use internet::*; // cela veut dire que l'on inclut TOUT ce que contient ce fichier
// ou comme ceci
use internet::{LaStruct, la_fonction};

// très important, le mod internet doit venir après !
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:
```
// vue.rs

pub use self::LaStruct;

// internet.rs

pub use super::LaStruct;

// ou bien

::LaStruct; // "::" voulant dire "dans le scope supérieur"

// ou bien encore

super::LaStruct; // super voulant aussi dire dans "le scope supérieur"
```

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".
- 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 rééexportés.

Un dernier exemple plus concret :

```
 - le_project
    |
    |- 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 reexporte "UnElement" de un_fichier.rs
pub use un_fichier::UnElement;

// On reexporte "UnAutreElement" de module1/file1.rs
pub use module1::file1::UnAutreElement;

// On reexporte "Element" de module1/file1.rs
pub use module1::Element;
// On aurait pu le reexporter de cette facon aussi : "pub use module1::file1::Element;"

// on reexporte "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 declarer 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 clotûre 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 !

# 11. 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.

## Fonctionnement

Nous rentrons maintenant dans le vif du sujet : une macro est définie au travers d'une série de règles qui sont des **conditions de 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 !");
    }
}

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

Et on obtient :

```
Bonjour !
```

Merveilleux ! Bon jusque là, rien de bien difficile. Mais ne vous inquiétez pas, ça arrive !

## Les arguments

Bien évidemment, les macros peuvent recevoir des arguments. Petit exemple :

```
Run the following code:
macro_rules! dire_quelque_chose {
    ($x:expr) => {
        println!("Il dit : '{}'", $x);
    };
}

fn main() {
    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** appellée **x** en paramètre. Après il nous a juste suffit de l'afficher. Maintenant on va ajouter la possibilité de passer un deuxième argument :

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);
    };
}

fn main() {
    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 "match" sur les arguments.

## Les différents types d'argument

Comme vous vous en doutez, il y a d'autres types en plus des **expr**. En voici la liste complète :

- **ident** : une identification. 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** : un motif (ou "pattern"). Exemples : "Some(t)"; "(17, 'a')"; "_".
- **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 "meta item", comme les attributs. Exemple : "cfg(target_os = "windows")".
- **tt** : un élément un peu plus global, il peut contenir toute une expression.

## 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()
    }
}

fn main() {
    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 pu mettre un ';' si on l'avait voulu. D'ailleurs pourquoi ne pas essayer ?

Run the following code:
```
macro_rules! vector {
    (
        $($x:expr);*
    ) => {
        [ $($x),* ].to_vec()
    }
}

fn main() {
    let mut v: Vec<u32> = vector!(1; 2; 3);

    v.push(6);
    println!("{:?}", &v);
}
```

Dans le cas présent, on récupère le tout dans un slice qui est ensuite transformé 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);
    )
}

fn main() {
    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 goût. 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. Exemple :

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() {
    let temp = Temperature { degree: 0u32 };
    // error: expected f32, found u32
    let point = Point { x: 0u32, y: 0u32, z: 0u32 };
    // error: expected f32, found u32 (pour les 3 champs)
}
```

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 (car 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:
```
#[macro_use]
extern crate nom_de_la_dependance;
```

## Quelques macros utiles

En bonus, je vous donne une petite liste de macros qui pourraient vous être utiles :

- panic!
- assert!
- assert_eq!
- try!
- 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).

# 12. 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 pour avoir une adresse constante quand on utilise une FFI (Foreign Function Interface), comme des pointeurs sur objet/fonction. Je n'en parlerai pas dans cette partie du cours.

Il existe le mot-clé **box** pour en créer une, par-contre, la feature le permettant est encore instable donc le code suivant ne fonctionnera pas si vous utilisez la version stable du compilateur :

Run the following code:
```
#![feature(box_syntax)]

let a: Box<[u8; 100000]> = Box::new([0; 100000]); // Bof...
let b: Box<[u8; 100000]> = box new([0; 100000]; // Mieux ! Mais ne marche qu'en nightly...
```

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 :

Run the following code:

```
use std::fmt::Display;

struct List<T> {
    a: T,
    next: Option<Box<List<T>>>, // None signifiera qu'on est à la fin de la chaîne
}

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 problème. Il me semblait donc important de vous en parler.

# III. Aller plus loin

## 1. Utiliser du code compilé en C avec les FFI

Rust permet d'exécuter du code compilé en C au travers des Foreign Function Interface (aussi appelée FFI). Ce chapitre va vous montrer comment faire.

### Les bases

La première chose à faire est d'ajouter une dépendance à la bibliothèque **libc** :

Cargo.toml :

```
[dependencies]
libc = "0.2"
```

Fichier principal :

```
Run the following code:
extern crate libc;
```

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:
extern crate libc;

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, les types fournis par la **libc** ont l'avantage d'être plus clairs et surtout de correspondre au type C. 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 :

fichier principal :

```
Run the following code:
#[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 l'utilité des **structures unitaires** !

On va dire que la bibliothèque en C ressemble à ça :

```
#define NOT_OK 0
#define OK 1

struct Handler; // on ne sait pas ce que la structure contient

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 de
// objets venant du C.
#[repr(C)]
pub struct FFIHandler;

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.

handler.rs :

Run the following code:

```
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_stri
        }
    }
}

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 là comment ajouter un callback, convertir une **String** entre C et Rust et nous avons surtout vu à quoi servaient les **structures unitaires** !

# 2. Documentation et rustdoc

En plus du compilateur, Rust possède un générateur de documentation. Toute la documentation en ligne de l'API (disponible ici) a été générée avec cet outil. Vous allez voir qu'il est très facile à mettre en oeuvre.

## Génération de la documentation

Commençons par le commencement : la **génération**. Si vous utilisez **Cargo**, rien de plus simple :

```
> cargo doc
```

Et c'est tout. Votre documentation se trouvera dans le dossier **target/doc/le_nom_de_votre_programme/**. Pour l'afficher, ouvrez le fichier **index.html** qui s'y trouve avec votre navigateur internet préféré, ou lancez la commande :

```
> cargo doc --open
```

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**.

## Ajouter des explications

Pour le moment, la documentation que je vous ai fait générer ne contient que du code sans rien d'autre. Pas tip top pour de la doc donc. Au final, ce serait bien qu'on ait une explication, comme ici :


Pour cela, rien de plus simple, il suffit d'utiliser les "///" :

```
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 champs sert à faire ceci
    un_champs: 32,
    /// et ce champs sert à faire cela
    un_autre_champs: i32
}
```

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 décrire. Ce code ne fonctionnera pas :

```
Run the following code:
enum Option<T> {
    None,
    Some(T), /// Some value `T`
}
```

Voilà pour les bases.

## Documenter un module

Il existe encore un autre niveau de commentaire qui sert à décrire le contenu d'un module, le "//!" ou "/*!". Il doit être mis avant que le code du module ne commence et ne peut être mis qu'une seule fois (par module). Cela donne :

## Crate libc [-] [+] [src]

Unstable: use `libc` from crates.io

[-] Bindings for the C standard library and other platform libraries

**NOTE:** These are *architecture and libc* specific. On Linux, these bindings are only correct for glibc.

This module contains bindings to the C standard library, organized into modules by their defining standard. Additionally, it contains some assorted platform-specific definitions. For convenience, most functions and types are reexported, so `use libc::*` will import the available C bindings as appropriate for the target platform. The exact set of functions available are platform specific.

*Note:* Because these definitions are platform-specific, some may not appear in the generated documentation.

We consider the following specs reasonably normative with respect to interoperating with the C standard library (libc/msvcrt):

- ISO 9899:1990 ('C95', 'ANSI C', 'Standard C'), NA1, 1995.
- ISO 9899:1999 ('C99' or 'C9x').
- ISO 9945:1988 / IEEE 1003.1-1988 ('POSIX.1').
- ISO 9945:2001 / IEEE 1003.1-2001 ('POSIX:2001', 'SUSv3').
- ISO 9945:2008 / IEEE 1003.1-2008 ('POSIX:2008', 'SUSv4').

Note that any reference to the 1996 revision of POSIX, or any revs between 1990 (when '88 was approved at ISO) and 2001 (when the next actual revision-revision happened), are merely additions of other chapters (1b and 1c) outside the core interfaces.

Despite having several names each, these are *reasonably* coherent point-in-time, list-of-definition sorts of specs. You can get each under a variety of names but will wind up with the same definition in each case.

See standards(7) in linux-manpages for more details.

Our interface to these libraries is complicated by the non-universality of conformance to any of them. About the only thing universally supported is the first (C95), beyond that definitions quickly become absent on various platforms.

We therefore wind up dividing our module-space up (mostly for the sake of sanity while editing, filling-in-details and eliminating duplication) into definitions common-to-all (held in modules named c95, c99, posix88, posix01 and posix08) and definitions that appear only on *some* platforms (named 'extra'). This would be things like significant OSX foundation kit, or Windows library kernel32.dll, or various fancy glibc, Linux or BSD extensions.

In addition to the per-platform 'extra' modules, we define a module of 'common BSD' libc routines that never quite made it into POSIX but show up in multiple derived systems. This is the 4.4BSD r2 / 1995 release, the final one from Berkeley after the lawsuits died down and the CSRG dissolved.

### Reexports

Petit exemple rapide :

Run the following code:

```
// copyright
// blablabla

//! Ce module fait ci.
//! Il fait aussi ça.
//!
//! #Titre
//! blabla
//! etc...

// du code...
pub mod un_module {
    //! Encore un module !
    //! Who dares summon the Rust documentation maker ?!
}
```

Si vous êtes un peu fainéant, vous pouvez aussi l'écrire de cette façon :

```
Run the following code:
// copyright
// blablabla

/*!
Ce module fait ci.
Il fait aussi ça.

#Titre
blabla
etc...
!*/
```

Cependant, il est plus rare de la voir dans les codes.

Voilà, vous savez maintenant gérer des documentations en Rust !

# 3. Ajouter des tests

Ce chapitre parlera des tests et de la métadonnée `#[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 :

## La métadonnée #[test]

Pour indiquer au compilateur que cette fonction est un test unique, il faut ajouter `#[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));
}
```

Plutôt facile, non ? Vous pouvez aussi mettre cette balise sur un module :

Run the following code:
```
fn some_func(valeur1: i32, valeur2: i32) -> i32 {
    valeur1 + valeur2
}

#[cfg(test)] // on ne compile que si on est en mode "test"
mod tests {
    use super::some_func;

    #[test] // on doit le remettre pour bien spécifier au compilateur que c'est un test
    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 cette balise :

```
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). 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 dossiers à part. Commencez par créer un dossier **tests** puis créez un fichier .rs.

Dans ce fichier, il vous faudra importer votre bibliothèque pour pouvoir tester ses fonctions :

```
Run the following code:
extern crate ma_lib;

#[test]
fn test_some_func() {
    assert_eq!(3, ma_lib::some_func(1, 2));
}
```

### Écrire des suites de tests

Si vous shouhaitez 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 vous le savez déjà, 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

```
/// ```
/// 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.

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 **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;
/// ```
```

C'est généralement inutile mais sait-on jamais...

Une dernière option pour la route (un peu étrange) :

```
```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

Tout autre option sera considérée comme un langage et passera le code en **ignore** invisible (vous ne le verrez pas apparaitre dans la liste des codes testés).

## Cacher des lignes

Pour certaines, vous pourriez vouloir cacher des lignes lors du rendu du code mais les gardez 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 !

# 4. 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

Les **RefCell** sont utiles 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 fenêtre 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 problèmatique ! 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) {
        for vue in &mut self.vues { // nous voulons &mut Vue et pas juste Vue
            vue.borrow_mut().pos.x += 1;
        }
        for layout in &mut self.layouts { // pareil que pour la boucle précédente
            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

Pour faire simple, le type **Rc** est un compteur de référence d'un objet constant (d'où son nom d'ailleurs). Exemple :

```
Run the following code:
use std::rc::Rc;

let r = Rc::new(5);
println!("{}", *r);
```

Juste 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(5);
println!("{:?} = {}", (&*r) as *const i32, *r);
let r2 = r.clone();
*Rc::make_mut(&mut r) = 10;
println!("{:?} = {}\n{:?} = {}", (&*r2) as *const i32, *r2, (&*r) as *const i32, *r);
```

Comme vous vous en serez rendu compte, l'objet contenu par `r` a changé. Pour éviter qu'une copie soit faite lorsque vous manipulez un type, il vous faudra passer par les types **Cell** ou **RefCell**. Cela pourra vous être très utile si vous avez des soucis avec des closures notamment.

# 5. Les threads

Commençons par un exemple tout bête :

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:
```
let mut data = vec![1u32, 2, 3];

for i in 0..3 {
    // on lance le thread
    thread::spawn(move || {
        data[i] += 1;
    });
}

// on attend 50 millisecondes, le temps que les threads finissent leur travail
thread::sleep_ms(50);
```

Vous devriez obtenir une magnifique erreur :

```
error: capture of moved value: `data`
        data[i] += 1;
```

Le système de propriété que vous haïssez sans doute 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 ça. Pour contourner ce problème, plusieurs solutions s'offrent à vous :

## Mutex

Le type **Mutex** permet donc d'échanger des informations entre threads. Une solution naïve serait de les utiliser de cette façon :

Run the following code:
```
use std::thread;
use std::sync::Mutex;

fn main() {
    let mut data = Mutex::new(vec![1u32, 2, 3]); // on crée notre mutex

    for i in 0..3 {
        let data = data.lock().unwrap(); // on lock
        // on lance le thread
        thread::spawn(move || {
            data[i] += 1;
        });
    }

    // on attend 50 millisecondes, le temps que les threads finissent leur travail
    thread::sleep_ms(50);
}
```

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 d'utiliser les données partagées de manière sûre ! C'est ici que rentre en jeu le type **Arc** !

## Arc

Vous l'aurez deviné (ou peut-être pas), 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]));

    for i in 0..3 {
        // on incrémente le compteur interne de Arc
        let data = data.clone();
        thread::spawn(move || {
            let mut ret = data.lock(); // on 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);
                }
            }
        });
    }

    // on attend 50 millisecondes, le temps que les threads finissent leur travail
    thread::sleep_ms(50);
}
```

Nous avons vu comment partager des données entre threads mais il nous reste cette ligne dont on voudrait bien se débarrasser :

Run the following code:
```
thread::sleep_ms(50);
```

les channels sont la solution à notre problème !

## Les channels

Nous aimerions donc bien pouvoir continuer l'exécution de notre programme mais uniquement après que les threads aient terminé. On crée un channel via la fonction **mpsc::channel**. Exemple :

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(());
        });
    }

    for _ in 0..10 {
        // on attend le signal de fin du thread
        rx.recv();
    }
}
```

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 données pour savoir que tous les threads se sont terminés.

Dans le code que je viens de vous montrer, on ne s'en sert que comme d'un signal en envoyant des données vides. Il est cependant possible d'envoyer des données, du moment qu'elles implémentent le trait **Send**. Exemple :

Run the following code:
```
use std::thread;
use std::sync::mpsc;

fn main() {
    // on crée le channel
    let (tx, rx) = mpsc::channel();

    for _ in 0..10 {
        let tx = tx.clone();

        thread::spawn(move || {
            let answer = 42u32;

            // on envoie la donnée dans le channel
            tx.send(answer);
        });
    }

    match rx.recv() {
        Ok(data) => println!("Le channel vient de recevoir : {}", data),
        Err(e) => println!("Une erreur s'est produite : {:?}", e)
    };
}
```

Et voilà ! 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)
};
```

Magique !

## Empoisonnement de Mutex

Vous savez maintenant comment partager les données de manière sûre entre des threads. Il reste cependant un petit détail à savoir 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 lock
    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 maintenant d'utiliser les threads de manière bien plus simple. Je vous recommande au moins d'y jeter un coup d'oeil :

- rayon
- crossbeam

# 6. Le réseau

Je présenterai ici surtout tout ce qui a attrait à des échanges réseaux en mode "connecté" plus simplement appelé TCP. Je pense qu'après ça, vous serez tout à fait en mesure d'utiliser d'autres protocoles réseaux comme l'UDP (qui est un mode "non-connecté") sans trop de problèmes.

Commençons par écrire le code d'un client :

## Le client

Pour le moment, je vais vous demander de 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 (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 montre les différentes façon 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").unwrap();

    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]


> ./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 ? Hé bien il vous suffit de boucler sur l'appel de la méthode **accept** et de gérer chaque client dans un thread. **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 changement donc. Maintenant comment pourrait-on faire pour gérer plusieurs clients en même temps ? 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 !

# 7. 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 T {
    fn yop() {
        println!("yop");
    }
}

fn main() {
    T::yop()
}
```

## Toujours plus vers le fonctionnel avec le slice pattern !

J'ai trouvé cette fonctionnalité assez sympa (mais encore instable pour le moment) donc je la mets ici :

Run the following code:

```
#![feature(slice_patterns)]

fn sum(values: &[i32]) -> i32 {
    match values {
        [head, tail..] => head + sum(tail),
        [] => 0,
    }
}

fn main() {
    println!("Sum: {}", sum(&[1, 2, 3, 4]));
}
```