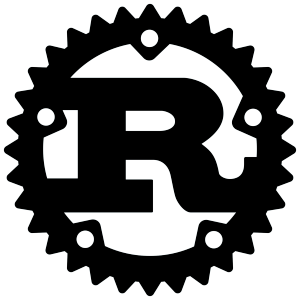
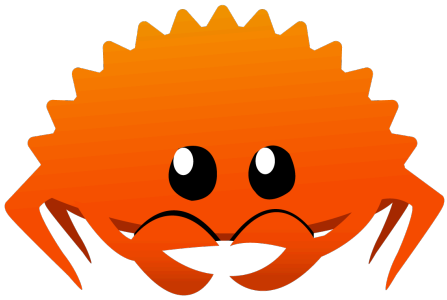
TP Rust





`https://rust-lang.org/`

1. Présentation de cargo

Rust est un langage compilé. On utilise `rustc` pour compiler un programme. Mais (en tout cas ici), on ne l'utilisera pas directement. On utilisera plutôt le gestionnaire de projets de rust, **cargo**.

Pour installer rustc + cargo, la façon la plus conventionnelle de faire est d'installer `rustup`, qui est un outil qui permet de gérer les nouvelles versions du langage, de la bibliothèque standard, de `rustc`, et de `cargo`. Il permet aussi de faire de la *cross-compilation* (compiler un programme sur une machine pour le porter sur une autre avec un autre OS). Tout ceci est assez transparent pour l'utilisateur, et vous n'aurez pas besoin de le faire ici.

1.1. Créer un nouveau projet

Nous allons créer un nouveau projet rust. Pour cela, dans la console, tapez la commande `cargo init premier`.

Cargo nous dit qu'il a créé un projet pour un binaire (par opposition à une bibliothèque). Il dit aussi qu'il a créé un fichier `Cargo.toml`. Ce fichier est important : il contient toute la description de notre projet, et notamment ses dépendences (bibliothèques externes dont on a besoin). Cela permet à quiconque dispose du `Cargo.toml` de compiler l'application.

Regardons le dossier `premier` qui a été créé par cargo. Il contient le fichier `Cargo.toml`, ainsi qu'un répertoire `src`, où se trouvera le code source de notre application.

1.2. Hello, world

Le répertoire `src` contient lui-même un fichier `main.rs`, qui est le fichier source principal. Ce `main.rs` contient un programme d'exemple, le fameux "Hello, world".

Regardez le contenu du fichier `main.rs`.

Nous allon maintenant exécuter ce programme. Il y a trois façons de faire :

- compiler et exécuter à la volée, pour un cycle rapide de développement - compilation - exécution lors du développement,
- compiler en mode *dev*, avec une compilation relativement rapide et un exécutable qui comporte peu d'optimisations et beaucoup de tests de sécurité, pour débugger le code, toujours pendant la phase de développement,
- compiler en mode *release*, un mode plus lent, où le compilateur optimise au maximum le code, lorsque le programme est fonctionnel et doit être mis en production.

Utilisons le premier, le plus simple. Tapez :

```
cargo run
```

Le compilateur compile le programme et l'exécute dans la foulée. Faites quelques modifications, et retapez la commande.

Passons au mode *dev*. Tapez :

```
cargo build
```

Le programme est compilé, sans être exécuté. Cargo a créé un nouveau répertoire à la racine du projet : `target` (cible). C'est là que se trouve notre exécutable. Dans `target`, il y a un répertoire `debug`: c'est là qu'est le programme compilé en mode *debug*.

Dans le répertoire `debug`, il y a tout un tas de fichiers auxquels on ne s'intéressera pas, et un fichier en particulier qui s'appelle `premier` (c'est le nom que l'on a donné à notre projet) qui est notre exécutable.

À la racine du projet, tapez :

```
./target/debug/premier
```

Votre programme doit s'exécuter.

Testons maintenant le mode `release` :

```
cargo build --release
```

La compilation prend un peu plus de temps (pas beaucoup vu la taille du programme, mais pour un vrai programme, la différence peut être significative). Testons le programme :

```
./target/release/premier
```

1.3. Calculateur de moyennes

On ne va pas faire de saisie au clavier cette fois, car cela nécessite de maîtriser des concepts pas encore abordés.

On va faire un calculateur de moyennes. On aura dans un tableau un certain nombre de notes, que l'on entrera en dur dans le programme. Par exemple, dans la fonction `main()`, déclarez la variable suivante :

```
let notes = [7, 12, 10, 4, 5, 13, 20];
```

Notez qu'il s'agit d'un tableau de `i32` (sur la pile), pas d'un vecteur (sur le tas).

Écrivez une fonction `moyenne(notes: &[i32]) -> i32`, qui renvoie la moyenne des notes, arrondie (ce sont des entiers), et testez-la en l'appelant dans votre fonction `main()` et en exécutant avec `cargo run`.

Modifiez la liste des notes, et relancez le programme. Essayez avec une liste vide : votre programme va *paniquer* : une erreur à l'exécution s'est produite, donc le programme se termine (proprement). On essaie d'éviter les *panic* (surtout dans le code mis en production), mais on n'a pas encore vu les outils adéquats.

Écrivez une fonction `min_et_max(notes: &[i32]) -> (i32, i32)` qui renvoie un tuple contenant les notes la plus basse et la plus haute de la liste. Affichez les résultats.

Écrivez une fonction `harmoniser` qui prend une liste de notes et :
- met toutes les notes entre 0 et 20,
- ajoute un point à tout le monde, sauf à ceux qui ont déjà 20.

Il faudra sans doute rendre la variable `notes` mutables pour que cela marche.

1.4. Les vecteurs entrent en piste

Moditiez notes. Jusqu'à présent, c'était un tableau : il était sur la pile, de taille fixe et connue à la compilation. Maintenant, nous allons en faire un vecteur (un `Vec<i32>`). Il suffit d'utiliser la macro `vec!` sur notre tableau :

```
let notes = vec![7, 12, 10, 4, 5, 13, 20];
```

Testez toutes les fonctions déja écrites. Normalement, il n'y a rien à changer : quand on passe une référence vers un tableau ou une référence vers un vecteur, le type associé est toujours `&[i32]`. C'est très pratique.

On peut aussi faire comme suit, en appelant la méthode `to_vec()` sur le tableau :

```
let notes = [7, 12, 10, 4, 5, 13, 20].to_vec();
```

On souhaite maintenant avoir les notes de plusieurs étudiants. Créez une variable `toutes_les_notes` qui contiendra autant de listes de notes qu'il y a d'étudiants. Par exemple, on aura trois étudiants, qui auront eu les notes `[10, 12, 14]`, `[7, 20]` et `[11, 14, 17, 3]`. Notez qu'ils n'ont pas forcément tous le même nombre de notes. Pouvez-vous utiliser un tableau de tableaux ? Un tableau de vecteurs ? Un vecteur de tableaux ? Un vecteur de vecteurs ? Pourquoi ?

Écrivez une fonction `stats` qui prend votre liste d'étudiants en paramètres, et qui affiche :
- la meilleure moyenne,
- la meilleure note de toute la promo,
- la pire note de toute la promo.

1.5. Fibonacci

Écrivez une fonction `fib(n: i32) -> i32` qui calcule le nième terme de la suite de Fibonacci. Rappel :

$$\text{fib}(0) = \text{fib}(1) = 1$$

et

$$\forall n \geq 2 \ \text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$$

.

Dans votre fonction `main`, calculez la valeur de `fib(45)`. Nous allons comparer les modes *dev* et *release*. Compilez et lancez le programme, d'abord en utilisant le mode *dev*, puis le mode *release*.

2. Autres commandes de cargo

Lancez la commande `cargo` seule (sans argument). Vous apercevez la liste (partielle) de toutes les commandes disponibles sous cargo.

Notamment, notez :

- `cargo add` pour ajouter des dépendences
- `cargo test` et `cargo bench`, pour tester le bon fonctionnement du programme et l'efficacité des solutions retenues (il faut évidemment écrire les fonctions associées en amont)
- `cargo check` vérifie que le programme est correct, sans le compiler
- `cargo clippy`(non listé) fait des vérifications plus poussées; il est très pénible, mais très puissant

Lancez `cargo clippy` sur tous vos programmes de ce TP. Corrigez tous les warnings, ou bien essayez de justifier pourquoi vous ne corrigerez pas tel ou tel warning. Notez qu'il est possible de

supprimer tel ou tel warning via des directives de compilation. Faites donc en sorte que `cargo clippy` n'affiche aucun warning ni aucune erreur.

3. Morpion

On va créer un programme rust qui permet de jouer au morpion.

Le morpion se joue à deux, dans une grille de 3 lignes, 3 colonnes. On joue chacun son tour dans une des 9 cases. Le but est d'aligner 3 pions de sa couleur, en ligne, colonne, ou diagonale. Une partie se conclut soit par la victoire d'un des deux joueurs, soit par un match nul.

3.1. Type `Joueur`

Il y a deux joueurs : celui qui place des ronds, celui qui place des croix. On créera donc une énumération :

```
#[derive(Debug, Copy, Clone)]
enum Joueur {
  Rond,
  Croix,
}
```

On dérive `Debug` pour pouvoir afficher les joueurs en mode debug, et `Copy` et `Clone` pour copier les valeurs de ce type facilement.

Écrivez une méthode `autre() -> Joueur` qui, à un joueur, associe l'autre joueur.

3.2. Type `Grille`

Une grille contient 9 cases (3 lignes et 3 colonnes). Chaque case est soit vide, soit utilisée par un des deux joueurs.

Implémentez le type `Grille`.

Implémentez les méthodes suivantes :

```
// Renvoie une grille vierge.
fn new() -> Grille;

// Affiche la grille sur la sortie standard.
// Les croix seront représentées par X
// Les ronds seront représentés par O
// Les cases libres seront représentées par .
fn afficher_grille(&self);

// Renvoie le gagnant, s'il y en a un.
fn winner(&self) -> Option<Joueur>;

// Vrai ssi la partie est finie : match nul ou victoire d'un des deux joueurs.
fn game_over(&self) -> bool;

// Joue à la ligne indiquée et à la colonne indiquée.
fn jouer(&mut self, line: usize, col: usize) -> Result<(), MorpionError>;
```

Notez que l'on a besoin d'un type `MorpionError` en cas d'erreur. Le voici :

```
enum MorpionError {
  InvalidLine(i32),   // L'argument indique la ligne concernée
  InvalidColumn(i32), // L'argument indique la colonne concernée
  SpotTaken(i32,i32), // Ligne et colonne
```

```
  InputError,         // Erreur de saisie utilisateur
}
```

3.3. Demander un nombre à l'utilisateur

Nous allons besoin de demander des valeurs à l'utilisateur. Recopiez la fonction suivante :

```rust
fn demander_nombre(msg: &str) -> Result<usize, MorpionError> {
    println!("{msg}");
    let mut input = String::new();
    io::stdin()
        .read_line(&mut input)
        .expect("Échec de la lecture de l'entrée utilisateur");
    match input.trim().parse::<usize>() {
        Ok(n) => Ok(n),
        Err(e) => Err(InputError)
    }
}
```

Pour cela, il est nécessaire d'utiliser `std::io`. Ajoutez la ligne d'import suivante tout au début du fichier :

```rust
use std::io;
```

3.4. Faire jouer deux humains

Maintenant, écrivez la fonction `main`. Nous allons faire joueur deux joueurs humains. Chacun son tour, chaque joueur pourra choisir une ligne et une colonne. Puis on affichera la grille.

Une fois la partie terminée, on affichera qui est le vainqueur (s'il y en a un).

3.5. Jouer contre la machine

Maintenant, la machine jouera contre l'humain. C'est la machine qui commence.

3.5.1. IA de base

On va faire une "IA" basée sur des règles simples pour débuter. Elle va suivre les règles suivantes :

- s'il y a une case qui nous permet de gagner, on y joue,
- s'il y a une case qui permettrait à l'adversaire de gagner au coup suivant, on y joue,
- si la case centrale est vide, on joue dedans,
- sinon, on joue dans une case libre quelconque (la première que l'on trouve, par exemple).

3.5.2. IA d base, mais avec un random

Plutôt que de choisir la première case libre que l'on trouve (dans le cas 4 ci-dessus), choisissez une case au hasard.

Oh non, comment on fait un random en rust ? En fait, ce n'est pas si simple que ça.

Il faut installer la dépendance `rand` :

```
cargo add rand
```

Cette commande modifie votre `Cargo.toml` et ajoute une ligne dans les dépendances :

```
[dependencies]
rand = "0.8"
```

Désormais, nous pouvons créer un générateur de nombre pseudo-aléatoires :

```rust
let mut rng = rand::Rng();
```

On ne va le créer qu'une seule fois, et l'utiliser chaque fois que l'on veut un nombre aléatoire :

```
let n = rng.random_range(1..=6); // Ceci est un dé
```

Modifiez votre programme pour que, dans le cas 4 ci-dessus, on choisisse une case libre au hasard.

3.5.3. IA basée sur un parcours exhaustif

Il n'y a pas tant de partie de morpion possible : comme il n'y a que 9 coups possibles, et qu'à chaque coup il y a une case de moins, il n'y a "que" $9! = 362880$ parties possibles. On peut donc toutes les tester (dans le pire des cas) sans que ce ne soit déraisonnable. C'est une version simplifiée de l'algorithme du minimax.

Écrivez une fonction récursive `meilleur_coup` qui prend en paramètre une grille, un joueur, et renvoie les coordonnées du meilleur coup possible pour ce joueur. Attention, ce n'est pas une fonction facile à écrire ! Cela va vous prendre du temps. Beaucoup de temps.