## Intro

Le but de ce TP est de prendre en main rust et sa syntaxe par le biais de petits exercices.

## Prérequis

Comme demandé lors de la précédente séance, vous devez avoir un environnent rust / cargo correctement configuré.

# Démarrer le projet

Créez un projet avec cargo.

```
$ cargo new tp1
$ cd tp1
$ cargo run
```

Vous devez observer `cargo` qui construit votre projet et l'exécute en affichant `Hello, world!`.

Le fichier `Cargo.toml` décrit votre projet. Nous discuterons de celui-ci plus tard.

Votre code est dans `src/main.rs`.

Pensez à visionner votre code.

# Travail à fournir

En vous aidant du code réalisé pendant la séance d'introduction, ainsi que de toute les ressources que vous désirez, vous aller implémenter différentes fonctions / types / ... en rust.

Le but de cette séance est de vous faire écrire du rust, alors allez y. Lisez les messages du compilateur. Faites des essais. Posez des questions.

Trouvez un moyen de "valider" vos créations. Le plus simple est bien souvent d'utiliser `println!("{}", valeur)` ou *valeur* est le résultat pertinent d'un calcul.

Ne bloquez pas. Vous avez un chargé de TP, il est là pour répondre à vos questions. Voyez le comme un moteur de recherche interactif très élaboré.

Vous faites du code "jetable", n'hésitez pas à tout mettre dans votre fonction `main`.

Suivez les conseils du compilateur rust, ils sont souvent judicieux. Si le compilateur vous parle de problème de "*liveness*", ou de "*borrowing*", essayez de suivre les conseils, essayez de développer une intuition, mais cela sera traité dans un futur TP. Dans le doute, demandez au chargé de TP qui vous proposera une solution temporaire.

Dans le doute, ajoutez `Clone` dans votre liste de `derive` de vos types (`#[derive(Clone)]`), et utilisez `.clone()`.

# Fonctions

Implémentez une fonction `mad(a:i32, b:i32, c:i32) -> i32` qui retourne la valeur `a * b + c`. Celle-ci prendra trois entiers `i32` en entrée et en renverra un en sortie.

Implémentez une fonction `sum_from_to(a:i32, b:i32) -> i32` qui renvoie la somme des nombres entre `a` et `b`. Il existe plusieurs façon de faire ce calcul, essayez en plusieurs (et nommez les avec des noms différents) :

1. La version mathématique, vous connaissez la formule.
2. Avec une boucle `while`. Vous aller devoir utiliser des `let mut` pour créer des variables modifiables.
3. Avec une boucle `for` et des "ranges" `1..20`. Observez comment les bornes du *range* sont incluses (ou pas).
4. Avec une récursion (terminale ou non)

# Creation de struct

Nous allons représenter une bibliothèque.

Pour cela nous allons créer la `struct Livre` qui contient les champs suivants:

- `titre`, qui sera une `String`.
- `annee_publication`, qui sera un `u32`.

Créez le type, puis créez plusieurs valeurs représentatives dans la fonction `main` (il faudra utiliser `to_string()`). Je vous encourage vivement à dériver `Debug` pour tous ces types.

Créer ensuite une fonction, `age_livre`, celle-ci prend un livre en paramètre et retourne l'age de celui-ci (*i.e.* l'année en cours moins l'année de publication). Nous sommes en 2021.

**Note:** Vous aurez sans doute un problème de `borrow` ici si vous faites plusieurs opérations sur le même livre. Utilisez `.clone()` sur vos livres.

# Création d'enum

Vous créerez ensuite un `enum Genre` (*i.e.* un type qui peut avoir plusieurs valeurs) représentants les "genres" d'un livre. Les valeurs possible étant:

- `Fiction`
- `Histoire`
- `Fantasy`
- `Informatique`

Ajoutez ensuite un champs `genre` au type `Livre` et complétez vos valeurs. Vous remarquerez que `Genre` doit `derive` au moins autant de comportement que `Livre`.

Nous allons maintenant écrire une fonction `note_livre` qui associe une note à un `Livre` grâce à un algorithme très précis :

La note d'un livre est la somme de:

- la longueur de son titre (en nombre de caractères) et de
- son année de publication

Tout cela multiplié par une valeur qui dépend de son genre:

- `Fiction`: 12
- `Histoire`: 2
- `Fantasy`: 36
- `Informatique`: 41

Ne vous formalisez pas trop sur l'équation. Le but est de vous faire écrire une fonction qui travail sur tous les champs du `Livre`. Vous devrez utiliser un `match` sur le champs `genre`.

# Enum avancé

Nous voulons créer une fonction de division, `division` (*i.e.* `a / b`) mais qui soit safe, ou "totale". C'est à dire qu'elle prend une valeur bien déterminée pour toutes ses valeurs d'entrée.

Hors, vous savez que diviser par `0` pose problème. L'appel à la fonction `division(a, b)` dans le cas où l'argument `b` est 0 devra renvoyer une valeur particulière.

Nous proposons tout d'abord de représenter le résultat de la fonction division par un type `enum`:

```
enum ResultatDivision
{
  DivisionParZero,
  DivisionCorrecte(f32)
}
```

Ce type possède deux cas, `DivisionParZero`, qui représente le résultat d'une division par zéro. Et `DivisionCorrecte` qui contiendra le résultat de la division sous la forme d'un `f32` (un nombre flottant).

- Implémentez le type `ResultatDivision` (*i.e.* copiez le code du dessus). Je vous encourage à `derive`, au moins `Debug`.
- Implémentez la fonction `division(a: f32, b: f32) -> ResultatDivision`.

**Note:* Le type `ResultatDivison` existe déjà de façon plus générique et s'appelle `Option<f32>`. Cherchez la documentation pour utiliser ce type à la place.