Programme d'apprentissage Rust - De Zéro à Héros

12 semaines pour maîtriser Rust à travers des projets immersifs

---

📅 SEMAINE 1 : Les Fondations de la Citadelle

Thème : Variables, Types et Fonctions

Jour 1 : Le Registre des Novices

Contexte : Vous êtes apprenti à la Citadelle d'Oldtown. Votre premier devoir est de créer un registre pour enregistrer les nouveaux novices.

Objectif : Apprendre les variables, mutabilité et types de base

Consignes :

1. Créez des variables pour stocker : nom (String), âge (u8), maison d'origine (String)
2. Affichez ces informations avec `println!`
3. Modifiez l'âge (utilisez `mut`)
4. Créez des constantes pour : ANNEE_FONDATION_CITADELLE = 1850

Code à écrire :

```rust
fn main() {
    // Créez vos variables ici
    let nom = "Samwell Tarly";
    // ... complétez le reste
}
```

Jour 2 : Le Compteur de Bougies

Contexte : Les maesters doivent calculer combien de bougies sont nécessaires pour éclairer la bibliothèque chaque soir.

Objectif : Opérations arithmétiques et types numériques

Consignes :

1. Créez une fonction `calculer_bougies(salles: u32, heures: u32) -> u32`
2. Chaque salle nécessite 5 bougies par heure
3. Calculez le total et affichez le résultat
4. Testez avec 12 salles pendant 8 heures

Jour 3 : Le Classificateur de Chaînes

Contexte : Organisez les chaînes des maesters par couleur (argent, or, acier).

Objectif : Conditions if/else et comparaisons

Consignes :

1. Créez une fonction `classifier_chaine(matiere: &str) -> &str`
2. Retournez le domaine d'expertise : "argent" → "Médecine", "or" → "Économie", "acier" → "Guerre"
3. Utilisez match ou if/else
4. Testez avec plusieurs matières

Jour 4 : Le Calendrier Lunaire

Contexte : Calculez les phases de la lune pour prédire les marées.

Objectif : Boucles (loop, while, for)

Consignes :

1. Créez une boucle for qui affiche les jours de 1 à 28
2. Marquez les jours de pleine lune (jour 14) et nouvelle lune (jour 1 et 28)
3. Utilisez if dans la boucle pour identifier ces jours spéciaux
4. Affichez : "Jour X : Phase lunaire"

Jour 5 : L'Inventaire des Grimoires

Contexte : Créez un système pour compter et catégoriser les livres de la bibliothèque.

Objectif : Tuples et premiers concepts de groupement

Consignes :

1. Créez des tuples pour représenter des livres : (titre, nombre_pages, est_ancien)
2. Créez une fonction qui prend un tuple et retourne si le livre est "rare" (ancien ET > 500 pages)
3. Testez avec au moins 5 livres différents

Jour 6 : Le Tableau d'Honneur

Contexte : Affichez les scores des novices à leurs examens.

Objectif : Arrays et itération

Consignes :

1. Créez un array avec 7 scores (u8) : `[85, 92, 78, 95, 88, 90, 73]`
2. Calculez la moyenne avec une boucle
3. Trouvez le score le plus élevé
4. Comptez combien de novices ont réussi (score >= 80)

Jour 7 : PROJET - Le Système d'Alchimie du Maester

Contexte : Créez un programme complet pour gérer les recettes alchimiques de la Citadelle.

Objectif : Intégrer tous les concepts de la semaine

Consignes :

1. Créez une structure de données pour une recette (nom, ingrédients nécessaires, temps de préparation en minutes)
2. Créez au moins 5 recettes stockées dans un array
3. Fonction 1 : `afficher_recettes()` - affiche toutes les recettes
4. Fonction 2 : `recettes_rapides(temps_max: u32)` - trouve les recettes sous un certain temps
5. Fonction 3 : `calculer_temps_total()` - somme tous les temps de préparation
6. Menu interactif (simulé avec match) permettant de choisir quelle fonction exécuter

Bonus : Ajoutez une fonction pour trouver la recette la plus longue

---

📅 SEMAINE 2 : La Propriété et l'Emprunt

Thème : Ownership, Borrowing, References

Jour 8 : Le Messager Unique

Contexte : Les corbeaux messagers ne peuvent porter qu'un seul message à la fois.

Objectif : Comprendre l'ownership de base

Consignes :

1. Créez une fonction `envoyer_message(message: String)`
2. Essayez d'utiliser la même String deux fois - observez l'erreur
3. Corrigez en clonant le message
4. Comparez le comportement avec un type Copy (u32)

Jour 9 : La Bibliothèque Partagée

Contexte : Plusieurs maesters doivent consulter le même livre sans le posséder.

Objectif : References immuables (&T)

Consignes :

1. Créez une fonction `lire_titre(livre: &String) -> usize` qui retourne la longueur
2. Appelez cette fonction plusieurs fois avec le même livre
3. Le livre reste accessible après les appels
4. Créez 3 fonctions qui empruntent le même livre simultanément

---

Jour 10 : L'Annotateur de Manuscrits

Contexte : Un maester doit annoter un manuscrit ancien.

Objectif : References mutables (&mut T)

Consignes :

1. Créez une fonction `ajouter_annotation(texte: &mut String, note: &str)`
2. La fonction doit ajouter la note à la fin du texte
3. Créez un manuscrit initial et ajoutez 3 annotations successives
4. Testez qu'une seule référence mutable existe à la fois

Jour 11 : Le Validateur de Parchemins

Contexte : Vérifiez l'authenticité des parchemins sans les altérer.

Objectif : Slices de String (&str)

Consignes :

1. Créez une fonction `extraire_signature(parchemin: &str) -> &str` qui retourne les 10 premiers caractères
2. Créez `verifier_sceau(parchemin: &str) -> bool` qui vérifie si le parchemin contient "Sceau Royal"
3. Testez avec plusieurs parchemins
4. Utilisez des slices pour extraire différentes parties

Jour 12 : La Copie des Cartes

Contexte : Les cartographes doivent copier des portions de cartes anciennes.

Objectif : Slices de tableaux et vecteurs (introduction aux Vec)

Consignes :

1. Créez un Vec<i32> représentant des coordonnées : `vec![10, 25, 30, 45, 50, 65, 70]`
2. Créez une fonction `extraire_region(carte: &[i32], debut: usize, fin: usize) -> &[i32]`
3. Calculez la moyenne d'une région spécifique
4. Trouvez la plus grande coordonnée dans une slice

Jour 13 : Le Coffre-fort de la Citadelle

Contexte : Gérez l'accès aux documents secrets avec des règles strictes.

Objectif : Règles d'emprunt complexes

Consignes :

1. Créez une structure simulant des tentatives d'accès concurrents
2. Fonction `consulter_document(doc: &String)` - peut être appelée plusieurs fois
3. Fonction `modifier_document(doc: &mut String)` - exclusive
4. Démontrez qu'on ne peut pas avoir une référence mutable et immuable simultanément

5. Utilisez des scopes {} pour gérer les durées de vie

Jour 14 : PROJET - Le Système de Prêt de la Bibliothèque

Contexte : Créez un système de gestion de prêts de livres respectant les règles d'ownership de Rust.

Objectif : Maîtriser ownership, borrowing et references

Consignes :

1. Créez une struct `Livre { titre: String, auteur: String, disponible: bool }`
2. Créez un Vec<Livre> avec 10 livres
3. Fonction `afficher_livres(bibliotheque: &Vec<Livre>)` - affiche tous les livres
4. Fonction `emprunter_livre(bibliotheque: &mut Vec<Livre>, titre: &str) -> Result<String, String>`
   - Change disponible à false si trouvé
   - Retourne Ok("Livre emprunté") ou Err("Livre non disponible")
5. Fonction `retourner_livre(bibliotheque: &mut Vec<Livre>, titre: &str)`
6. Fonction `livres_disponibles(bibliotheque: &Vec<Livre>) -> Vec<&str>` - retourne les titres disponibles
7. Créez un scénario avec 5 emprunts et 2 retours

Bonus : Ajoutez un système de réservation avec une file d'attente

---

📅 SEMAINE 3 : Structures et Enums

Thème : Structs, Enums, Pattern Matching

Jour 15 : Le Dossier du Maester

Contexte : Créez un profil complet pour chaque maester.

Objectif : Structures de base

Consignes :

1. Créez une struct `Maester { nom: String, age: u8, chaines: Vec<String>, specialite: String }`
2. Créez 3 instances de maesters différents
3. Créez une fonction `afficher_maester(m: &Maester)`
4. Fonction `compter_chaines(m: &Maester) -> usize`

Jour 16 : Les Méthodes du Grand Maester

Contexte : Ajoutez des comportements aux maesters.

Objectif : Implémentation de méthodes (impl)

Consignes :

1. Reprenez la struct Maester
2. Créez un bloc `impl Maester {}`
3. Méthode `new(nom: String, age: u8) -> Self` (constructeur)
4. Méthode `ajouter_chaine(&mut self, matiere: String)`
5. Méthode `est_experimente(&self) -> bool` (retourne true si >= 5 chaînes)
6. Testez toutes les méthodes

Jour 17 : Les Ordres Militaires

Contexte : Représentez les différents ordres de chevalerie.

Objectif : Enums de base

Consignes :

1. Créez un enum `Ordre { GuetDeNuit, Garde, Kingsguard, SanOrdre }`

2. Créez une fonction `decrire_ordre(ordre: Ordre) -> String` avec match
3. Créez une struct `Chevalier { nom: String, ordre: Ordre }`
4. Testez avec 4 chevaliers différents

Jour 18 : Les Messages du Royaume

Contexte : Gérez différents types de messages avec données associées.

Objectif : Enums avec données

Consignes :

1. Créez un enum `Message { Texte(String), Chiffre(u32), Urgent { contenu: String, priorite: u8 } }`
2. Fonction `traiter_message(msg: Message)` avec match
3. Différents traitements selon le type
4. Créez et traitez 5 messages variés

Jour 19 : Le Trésor Royal (Option)

Contexte : Gérez des coffres qui peuvent être vides ou contenir de l'or.

Objectif : Option<T> et pattern matching

Consignes :

1. Fonction `ouvrir_coffre(numero: u32) -> Option<u32>` qui retourne Some(or) ou None
2. Fonction `calculer_richesse(coffres: Vec<Option<u32>>) -> u32`
3. Utilisez match, if let, et unwrap_or
4. Testez avec un vec de 10 coffres (certains vides)

Jour 20 : Le Verdict du Jugement (Result)

Contexte : Gérez les résultats des procès avec succès ou erreurs.

Objectif : Result<T, E> et gestion d'erreurs

Consignes :

1. Fonction `juger_crime(gravite: u8) -> Result<String, String>`
2. Retourne Ok("Innocent") si < 3, Err("Coupable") si >= 3
3. Créez une fonction qui traite plusieurs jugements et compte les verdicts
4. Utilisez ?, match, et unwrap_or_else

Jour 21 : PROJET - Le Système de Lignées de Westeros

Contexte : Pour aider un maester de la Citadelle, créez un système qui trace les alliances et les lignées des grandes maisons.

Objectif : Intégrer structs, enums, methods et pattern matching

Consignes :

1. Enum `Maison { Stark, Lannister, Targaryen, Baratheon, Tyrell, Martell, Greyjoy, Arryn }`
2. Enum `StatutAlliance { Allie, Ennemi, Neutre }`
3. Struct `Seigneur { nom: String, maison: Maison, age: u8, vassaux: Vec<String> }`
4. Struct `Alliance { maison1: Maison, maison2: Maison, statut: StatutAlliance }`
5. Struct `Royaume { seigneurs: Vec<Seigneur>, alliances: Vec<Alliance> }`

Implémentez les méthodes suivantes sur Royaume :

- `new() -> Self` - crée un royaume vide
- `ajouter_seigneur(&mut self, seigneur: Seigneur)`
- `declarer_alliance(&mut self, m1: Maison, m2: Maison, statut: StatutAlliance)`
- `trouver_seigneur(&self, nom: &str) -> Option<&Seigneur>`

- `compter_allies(&self, maison: Maison) -> u32`
- `maisons_ennemies(&self, maison: Maison) -> Vec<Maison>`
- `puissance_militaire(&self, maison: Maison) -> u32` (compte les vassaux)

Scénario de test :

- Créez 8 seigneurs (un par maison)
- Déclarez 10 alliances variées
- Testez toutes les fonctions
- Affichez un rapport complet du royaume

Bonus : Créez une fonction `survie_complot(reine: Maison) -> bool` qui détermine si la reine a assez d'alliés (>= 3) pour survivre

---

📅 SEMAINE 4 : Collections et Itérateurs

Thème : Vec, HashMap, Iterators

Jour 22 : L'Armée du Roi

Contexte : Gérez les troupes avec des vecteurs dynamiques.

Objectif : Maîtriser Vec<T>

Consignes :

1. Créez un Vec<String> de soldats avec vec![]
2. Ajoutez 5 soldats avec push()
3. Retirez le dernier avec pop()
4. Insérez un général en position 0 avec insert()
5. Parcourez avec for et affichez chaque soldat
6. Utilisez len() et is_empty()

Jour 23 : Le Recensement des Maisons

Contexte : Comptez les membres de chaque maison avec une HashMap.

Objectif : HashMap de base

Consignes :

1. Créez une HashMap<String, u32> (nom maison -> nombre de membres)
2. Insérez 7 maisons avec insert()
3. Fonction `ajouter_membre(carte: &mut HashMap<String, u32>, maison: &str)`
4. Utilisez entry() et or_insert()
5. Affichez toutes les maisons avec for (cle, valeur)

Jour 24 : Le Glossaire des Termes Anciens

Contexte : Créez un dictionnaire de mots en Haut Valyrien.

Objectif : HashMap avancé avec String keys

Consignes :

1. HashMap<String, String> pour mot -> traduction
2. Ajoutez 15 mots/traductions
3. Fonction `traduire(dico: &HashMap<String, String>, mot: &str) -> Option<&String>`
4. Fonction `mots_commencant_par(dico: &HashMap<String, String>, lettre: char) -> Vec<String>`
5. Comptez combien de traductions contiennent "dragon"

Jour 25 : La Caravane Marchande

Contexte : Transformez des données de marchands avec des itérateurs.

Objectif : Iterator basics (map, filter, collect)

Consignes :

1. Vec de prix : `vec![10, 25, 5, 50, 15, 30, 8]`
2. Utilisez map() pour appliquer une taxe de 20%
3. Utilisez filter() pour garder seulement les prix > 20
4. Utilisez collect() pour récupérer un Vec
5. Calculez la somme avec sum()

Jour 26 : Les Statistiques de Batailles

Contexte : Analysez les résultats de batailles historiques.

Objectif : Chaînage d'itérateurs complexe

Consignes :

1. Struct `Bataille { nom: String, annee: u32, victimes: u32 }`
2. Créez un Vec de 10 batailles
3. Trouvez la bataille la plus meurtrière avec max_by_key()
4. Comptez les batailles après l'an 280 avec filter().count()
5. Calculez la moyenne de victimes avec map().sum() / len
6. Créez un Vec des noms de batailles triées par année avec sort_by()

Jour 27 : L'Inventaire Multi-Niveau

Contexte : Gérez des coffres contenant des objets variés.

Objectif : Collections imbriquées

Consignes :

1. HashMap<String, Vec<String>> - nom de coffre -> liste d'objets
2. Ajoutez 5 coffres avec 3-5 objets chacun
3. Fonction `compter_total_objets(inventaire: &HashMap<String, Vec<String>>) -> usize`
4. Fonction `chercher_objet(inventaire: &HashMap<String, Vec<String>>, objet: &str) -> Option<String>`
   - Retourne le nom du coffre contenant l'objet
5. Créez un Vec de tous les objets uniques (pas de doublons)

Jour 28 : PROJET - La Conspiration de la Cour

Contexte : Pour un maester astucieux, créez un système d'analyse politique qui révèle les complots à la cour en traçant les interactions et alliances secrètes.

Objectif : Maîtriser les collections et itérateurs dans un contexte réaliste

Structures de données :

1. Struct `Noble { nom: String, maison: String, influence: u32, loyaute_couronne: u8 }` (loyauté sur 10)
2. Struct `Interaction { noble1: String, noble2: String, type_rencontre: String, secret: bool }`
3. Struct `Cour { nobles: Vec<Noble>, interactions: Vec<Interaction>, alliances: HashMap<String, Vec<String>> }`

Implémentez sur Cour :

- `new() -> Self`
- `ajouter_noble(&mut self, noble: Noble)`
- `enregistrer_interaction(&mut self, inter: Interaction)`
- `former_alliance(&mut self, noble1: &str, noble2: &str)` - met à jour la HashMap
- `influence_totale_maison(&self, maison: &str) -> u32` - somme les influences

- `nobles_influents(&self) -> Vec<&Noble>` - retourne ceux avec influence > 50
- `rencontres_secretes(&self) -> Vec<(&str, &str)>` - paires de nobles ayant des interactions secrètes
- `detecter_conspiration(&self, seuil_loyaute: u8) -> HashMap<String, Vec<String>>`
  - Regroupe les nobles peu loyaux par maison
  - Utilise filter(), group_by logic, collect()
- `rapport_securite(&self) -> String` - synthèse complète

Scénario :

1. Créez 15 nobles de 6 maisons différentes avec influences et loyautés variées
2. Enregistrez 20 interactions (10 publiques, 10 secrètes)
3. Formez 8 alliances
4. Exécutez toutes les analyses
5. Générez le rapport de sécurité final

Bonus : Fonction `nobles_a_surveiller(&self) -> Vec<String>` qui identifie les nobles avec basse loyauté ET beaucoup d'interactions secrètes

---

📅 SEMAINE 5 : Traits et Génériques

Thème : Traits, Generics, Trait Bounds

Jour 29 : Le Protocole des Messagers

Contexte : Différents messagers ont tous besoin de pouvoir "délivrer".

Objectif : Créer et implémenter un trait simple

Consignes :

1. Créez un trait `Messager { fn delivrer(&self) -> String; }`
2. Struct `Corbeau { destination: String }`

3. Struct `Cavalier { nom: String, vitesse: u32 }`
4. Implémentez Messager pour les deux
5. Fonction `envoyer_message<T: Messager>(m: T)` qui appelle delivrer()

Jour 30 : La Forge Générique

Contexte : Créez un système qui fonctionne pour différents types d'armes.

Objectif : Fonctions génériques de base

Consignes :

1. Fonction générique `afficher<T: std::fmt::Display>(item: T)`
2. Fonction `creer_paire<T>(a: T, b: T) -> (T, T)` where T: Clone
3. Testez avec différents types : i32, String, etc.
4. Struct générique `Coffre<T> { contenu: Vec<T> }`

Jour 31 : Le Registre Universel

Contexte : Un registre qui peut stocker n'importe quel type de données.

Objectif : Structs génériques avec méthodes

Consignes :

1. Struct `Registre<T> { entrees: Vec<T> }`
2. Impl méthodes : `new()`, `ajouter(&mut self, item: T)`, `obtenir(&self, index: usize) -> Option<&T>`
3. Impl méthode qui nécessite trait bound : `afficher_tout(&self)` where T: Display
4. Testez avec Registre<String> et Registre<u32>

Jour 32 : Les Combattants de l'Arène

Contexte : Différentes créatures peuvent combattre si elles respectent certains critères.

Objectif : Trait bounds multiples

Consignes :

1. Trait `Combattant { fn force(&self) -> u32; fn nom(&self) -> &str; }`
2. Struct `Chevalier` et `Dragon` implémentant Combattant
3. Fonction `duel<T, U>(c1: &T, c2: &U) -> String` where T: Combattant, U: Combattant
4. Retourne le nom du gagnant basé sur force()

Jour 33 : Les Archives Comparables

Contexte : Triez et comparez des documents anciens.

Objectif : Traits standards (Clone, PartialEq, PartialOrd)

Consignes :

1. Struct `Document { titre: String, importance: u8 }` avec derives
2. `#[derive(Clone, PartialEq, PartialOrd)]`
3. Créez un Vec<Document> et triez-le avec sort()
4. Fonction `trouver_duplicatas(docs: &[Document]) -> Vec<Document>`
5. Testez l'égalité avec ==

Jour 34 : Le Convertisseur Universel

Contexte : Convertissez entre différents systèmes de mesure.

Objectif : Traits From et Into

Consignes :

1. Struct `Lieues(u32)` et `Kilometres(u32)`
2. Implémentez `From<Lieues> for Kilometres` (1 lieue = 4 km)
3. Utilisez automatiquement Into
4. Fonction générique qui accepte `T: Into<Kilometres>`
5. Testez les conversions dans les deux sens

Jour 35 : PROJET - Le Bestiaire de Westeros

Contexte : Le maester Luwin vous demande de créer un système flexible pour cataloguer toutes les créatures de Westeros, des loups-garous aux dragons.

Objectif : Combiner traits, génériques et collections

Traits requis :

1. `Creature { fn nom(&self) -> &str; fn niveau_danger(&self) -> u8; fn habitat(&self) -> String; }`
2. `Combattant { fn attaque(&self) -> u32; fn defense(&self) -> u32; }`
3. `Volant { fn altitude_max(&self) -> u32; }`

Structs (implémentez les traits appropriés) :

1. `Loup { nom: String, meute: String, danger: u8, attaque: u32, defense: u32 }`
2. `Dragon { nom: String, couleur: String, danger: u8, attaque: u32, defense: u32, altitude: u32 }`
3. `Geant { nom: String, tribu: String, danger: u8, attaque: u32, defense: u32 }`
4. `Corbeau { nom: String, messager: bool, altitude: u32 }` (non combattant)

Struct générique principale :

```
rust
```

```
struct Bestiaire<T: Creature> {
    creatures: Vec<T>,
    region: String,
}
```

Implémentez sur Bestiaire<T> :

- `new(region: String) -> Self`
- `ajouter(&mut self, creature: T)`
- `creature_la_plus_dangereuse(&self) -> Option<&T>`
- `compter(&self) -> usize`

Fonctions génériques indépendantes :

- `simuler_combat<T, U>(c1: &T, c2: &U) -> String` where T: Combattant, U: Combattant
  - Retourne le vainqueur basé sur (attaque + defense)
- `creatures_volantes<T: Creature + Volant>(bestiaire: &Bestiaire<T>) -> Vec<&T>`
- `rapport_danger<T: Creature>(creatures: &[T]) -> HashMap<u8, usize>`
  - Groupe les créatures par niveau de danger

Scénario :

1. Créez 3 bestiaires différents : Bestiaire<Loup>, Bestiaire<Dragon>, Bestiaire<Geant>
2. Ajoutez 5 créatures dans chaque bestiaire
3. Trouvez la créature la plus dangereuse de chaque bestiaire
4. Organisez 3 combats entre créatures de différents bestiaires
5. Créez un bestiaire de corbeaux et listez tous les volants
6. Générez un rapport de danger pour tous les dragons

Bonus : Créez un trait `Apprivoisable` avec méthode `peut_etre_dresse(&self) -> bool` et implémentez-le pour Loup et Corbeau uniquement

📅 SEMAINE 6 : Gestion d'Erreurs Avancée

Thème : Error Handling, Custom Errors, Propagation

Jour 36 : Le Coffre Verrouillé

Contexte : Tentez d'ouvrir des coffres avec différents résultats.

Objectif : Result<T, E> approfondi et match

Consignes :

1. Fonction `ouvrir_coffre(code: u32) -> Result<String, String>`
2. Retourne Ok avec trésor si code == 1234
3. Retourne Err avec message d'erreur sinon
4. Fonction `essayer_codes(codes: Vec<u32>) -> Vec<Result<String, String>>`
5. Gérez les résultats avec match, is_ok(), is_err()

Jour 37 : La Chaîne de Commandement

Contexte : Les ordres passent par plusieurs niveaux, chacun pouvant échouer.

Objectif : L'opérateur ? pour propagation d'erreurs

Consignes :

1. Fonction `valider_ordre(ordre: &str) -> Result<(), String>` qui vérifie si l'ordre est valide
2. Fonction `transmettre_ordre(ordre: &str) -> Result<(), String>` qui utilise ?
3. Fonction `executer_mission() -> Result<String, String>` qui chaîne plusieurs opérations avec ?
4. Testez avec ordres valides et invalides
5. Observez comment l'erreur remonte automatiquement

Jour 38 : Les Erreurs Typées

Contexte : Créez des types d'erreurs spécifiques pour différentes situations.

Objectif : Custom error types avec enum

Consignes :

1. Enum `ErreurBibliotheque { LivreIntrouvable, LivreDejaEmprunte, PermissionRefusee }`
2. Implémentez `std::fmt::Display` pour l'enum
3. Fonction `emprunter_livre(titre: &str, age_lecteur: u8) -> Result<String, ErreurBibliotheque>`
4. Utilisez match pour gérer chaque type d'erreur différemment
5. Testez tous les cas d'erreur

Jour 39 : La Validation en Cascade

Contexte : Validez des données avec plusieurs étapes pouvant échouer.

Objectif : Chaînage avec and_then, or_else, map_err

Consignes :

1. Fonction `parser_age(s: &str) -> Result<u8, String>`
2. Fonction `valider_age(age: u8) -> Result<u8, String>` (doit être entre 18 et 100)
3. Chaînez les deux avec `and_then()`
4. Utilisez `map_err()` pour transformer les messages d'erreur
5. Fonction `traiter_inscription(age_str: &str) -> Result<String, String>` qui utilise tout

Jour 40 : Le Système de Logs

Contexte : Enregistrez les erreurs dans un journal sans arrêter l'exécution.

Objectif : unwrap_or, unwrap_or_else, expect

Consignes :

1. Fonction `lire_config(cle: &str) -> Result<String, String>`
2. Utilisez `unwrap_or()` pour valeur par défaut
3. Utilisez `unwrap_or_else()` avec une closure pour valeur calculée
4. Utilisez `expect()` avec message personnalisé pour cas critiques
5. Créez une fonction qui traite 10 configurations et ne plante jamais

---

Jour 41 : Le Convertisseur Robuste

Contexte : Convertissez des données avec gestion d'erreurs complète.

Objectif : Conversion de types d'erreurs

Consignes :

1. Utilisez `parse()` pour convertir String en nombres
2. Gérez `ParseIntError` et `ParseFloatError`
3. Créez un enum personnalisé englobant ces erreurs
4. Implémentez `From<ParseIntError>` pour votre enum
5. Fonction qui parse plusieurs types et agrège les erreurs

Jour 42 : PROJET - Le Système de Navigation Maritime

Contexte : Pour la Flotte de Fer, créez un système de planification de routes maritimes qui doit gérer de nombreuses erreurs possibles : tempêtes, pirates, provisions insuffisantes, cartes incorrectes.

Objectif : Maîtriser la gestion d'erreurs complète dans un contexte réaliste

Custom Error Types :

```
rust
```

```
#[derive(Debug)]
enum ErreurNavigation {
    TempeteDetectee { zone: String, intensite: u8 },
    PiratesSur le Zone { nombre: u32 },
    ProvisionsInsuffisantes { jours_manquants: u32 },
    CartePasTrouvee { destination: String },
    DistanceTropLongue { distance: u32, max: u32 },
}
```

Structs :

1. `Port { nom: String, position: (i32, i32), securite: u8 }`
2. `Navire { nom: String, provisions: u32, rayon_action: u32 }`
3. `Route { depart: String, arrivee: String, distance: u32, danger: u8 }`

Implémentez Display pour ErreurNavigation

Fonctions requises (toutes retournent Result) :

1. `calculer_distance(p1: (i32, i32), p2: (i32, i32)) -> Result<u32, ErreurNavigation>`
   - Retourne erreur si distance > 1000
2. `verifier_provisions(navire: &Navire, distance: u32) -> Result<(), ErreurNavigation>`
   - 1 jour de provision par 100 km
   - Erreur si insuffisant
3. `evaluer_securite(route: &Route) -> Result<(), ErreurNavigation>`
   - Erreur si danger > 7
4. `trouver_port(ports: &[Port], nom: &str) -> Result<&Port, ErreurNavigation>`
   - Erreur si port introuvable
5. `planifier_route(navire: &Navire, ports: &[Port], depart: &str, arrivee: &str) -> Result<Route, ErreurNavigation>`
   - Utilise ? pour propager toutes les erreurs

- Vérifie tout en cascade
- Retourne la route si tout est OK

6. `simuler_traversee(navire: &Navire, route: &Route, meteo: u8) -> Result<String, ErreurNavigation>`
   - Simule la traversée
   - Erreur aléatoire de tempête si meteo > 5
7. `rapport_erreurs(erreurs: Vec<ErreurNavigation>) -> String`
   - Génère rapport lisible

Scénario principal :

1. Créez une flotte de 5 navires avec caractéristiques variées
2. Créez une carte de 10 ports dans l'océan
3. Planifiez 15 routes différentes
4. Pour chaque route :
   - Essayez de planifier avec `planifier_route()`
   - Si succès, simulez la traversée
   - Collectez toutes les erreurs
5. Générez un rapport final avec :
   - Nombre de routes réussies
   - Liste détaillée de toutes les erreurs rencontrées
   - Recommandations pour améliorer la flotte

Bonus : Fonction `route_alternative(navire: &Navire, ports: &[Port], depart: &str, arrivee: &str) -> Result<Vec<Route>, ErreurNavigation>` qui trouve un chemin avec escales si la route directe échoue

---

📅 SEMAINE 7 : Lifetimes et Smart Pointers

Thème : Lifetimes, Box, Rc, RefCell

Jour 43 : Les Références Durables

Contexte : Gérez des références qui doivent vivre assez longtemps.

Objectif : Syntaxe des lifetimes de base

Consignes :

1. Fonction `plus_long<'a>(s1: &'a str, s2: &'a str) -> &'a str`
2. Retourne la chaîne la plus longue
3. Struct `Parchemin<'a> { contenu: &'a str }`
4. Méthode `afficher(&self)` sur Parchemin
5. Testez avec différentes durées de vie

Jour 44 : L'Archiviste

Contexte : Une struct qui garde des références à des données externes.

Objectif : Lifetimes dans les structs

Consignes :

1. Struct `Archiviste<'a> { nom: &'a str, documents: Vec<&'a str> }`
2. Impl méthodes avec lifetimes corrects
3. Méthode `ajouter_document(&mut self, doc: &'a str)`
4. Méthode `premier_document(&self) -> Option<&str>`
5. Créez plusieurs archivistes avec lifetimes imbriqués

Jour 45 : La Tour Infinie

Contexte : Créez une structure de données récursive (arbre).

Objectif : Box<T> pour récursion

Consignes :

1. Enum `Tour { Etage { hauteur: u32, suivant: Box<Tour> }, Sommet }`
2. Fonction `construire_tour(etages: u32) -> Tour`
3. Fonction récursive `hauteur_totale(tour: &Tour) -> u32`
4. Créez une tour de 10 étages
5. Calculez la hauteur totale

Jour 46 : La Bibliothèque Partagée

Contexte : Plusieurs lecteurs accèdent au même livre simultanément.

Objectif : Rc<T> pour ownership partagé

Consignes :

1. `use std::rc::Rc;`
2. Créez un `Rc<String>` contenant un texte
3. Clonez la référence 5 fois
4. Vérifiez avec `Rc::strong_count()`
5. Struct `Lecteur { livre: Rc<String> }`
6. Créez 3 lecteurs partageant le même livre

Jour 47 : Le Registre Mutable

Contexte : Modifiez des données à travers une référence immuable.

Objectif : RefCell<T> pour interior mutability

Consignes :

1. `use std::cell::RefCell;`
2. Struct `Compteur { valeur: RefCell<u32> }`
3. Méthode `incrementer(&self)` qui modifie via `borrow_mut()`
4. Méthode `obtenir(&self) -> u32` qui lit via `borrow()`
5. Testez le compteur avec une référence immuable

Jour 48 : L'Arbre Généalogique

Contexte : Créez un graphe de relations familiales.

Objectif : Rc<RefCell<T>> combinés

Consignes :

1. `use std::rc::Rc; use std::cell::RefCell;`
2. Struct `Personne { nom: String, enfants: Vec<Rc<RefCell<Personne>>> }`
3. Créez une hiérarchie de 3 générations
4. Fonction récursive `compter_descendants(p: &Rc<RefCell<Personne>>) -> u32`
5. Ajoutez des enfants après création initiale

Jour 49 : PROJET - Le Réseau de Corbeaux Messagers

Contexte : Le Grand Maester vous confie la création d'un système de communication par corbeaux qui forme un réseau complexe entre les châteaux de Westeros.

Objectif : Maîtriser Box, Rc, RefCell et lifetimes dans un système réaliste

Structures de données :

rust

```rust
use std::rc::Rc;
use std::cell::RefCell;

struct Message<'a> {
    contenu: &'a str,
    priorite: u8,
    expediteur: &'a str,
}

struct Corbeau {
    id: u32,
    nom: String,
    vitesse: u32,
    fatigue: RefCell<u32>,
}

struct Tour {
    nom: String,
    position: (i32, i32),
    corbeaux: Vec<Rc<Corbeau>>,
    connexions: Vec<Rc<RefCell<Tour>>>,
}

struct Reseau {
    tours: Vec<Rc<RefCell<Tour>>>,
    messages_envoyes: RefCell<Vec<String>>,
}
```

Fonctions sur Tour (méthodes) :

1. `new(nom: String, position: (i32, i32)) -> Self`
2. `ajouter_corbeau(&mut self, corbeau: Rc<Corbeau>)`
3. `connecter(&mut self, autre_tour: Rc<RefCell<Tour>>)`

4. `corbeau_disponible(&self) -> Option<Rc<Corbeau>>`
    - Trouve un corbeau avec fatigue < 50

Fonctions sur Reseau (méthodes) :

1. `new() -> Self`
2. `ajouter_tour(&mut self, tour: Rc<RefCell<Tour>>)`
3. `trouver_tour(&self, nom: &str) -> Option<Rc<RefCell<Tour>>>`
4. `envoyer_message<'a>(&self, msg: Message<'a>, origine: &str, destination: &str) -> Result<String, String>`
    - Trouve les tours
    - Vérifie la disponibilité d'un corbeau
    - Augmente la fatigue du corbeau
    - Enregistre dans messages_envoyes
5. `statistiques(&self) -> String`
    - Nombre total de tours
    - Nombre total de corbeaux
    - Messages envoyés
    - Fatigue moyenne des corbeaux

Fonctions utilitaires (indépendantes) :

1. `creer_corbeau(id: u32, nom: String, vitesse: u32) -> Rc<Corbeau>`
2. `distance_tours(t1: &Tour, t2: &Tour) -> f64`
3. `chemin_le_plus_court(reseau: &Reseau, origine: &str, destination: &str) -> Option<Vec<String>>`
    - Retourne les noms des tours dans le chemin (algorithme simple)

Scénario :

1. Créez un réseau de 8 tours représentant les grands châteaux :
    - Winterfell, King's Landing, Casterly Rock, The Eyrie, Highgarden, Dorne, Pyke, The Wall
2. Positionnez-les sur une carte 2D (coordonnées)
3. Créez 20 corbeaux et distribuez-les (2-3 par tour)
4. Connectez les tours en réseau (chaque tour connectée à 2-4 autres)
5. Envoyez 15 messages variés :
    - Messages urgents (priorité 10)
    - Messages routiniers (priorité 1-5)
    - Certains doivent échouer (corbeau fatigué ou indisponible)
6. Simulez la fatigue : après 5 messages, augmentez la fatigue de tous les corbeaux
7. Générez un rapport complet des statistiques
8. Affichez le chemin le plus court entre Winterfell et King's Landing

Bonus : Implémentez un système de "repos" où les corbeaux récupèrent : fonction `repos_nocturne(reseau: &Reseau)` qui réduit la fatigue de tous les corbeaux de 20 points

---

📅 SEMAINE 8 : Modules et Organisation

Thème : Modules, Crates, Visibility, Tests

Jour 50 : La Hiérarchie des Commandements

Contexte : Organisez le code en modules logiques.

Objectif : Modules de base avec mod

Consignes :

1. Créez un module `armee` avec `mod armee {}`
2. À l'intérieur, struct `Soldat` et fonction `recruter()`

3. Rendez-les publics avec `pub`
4. Utilisez depuis main avec `armee::recruter()`
5. Créez un sous-module `armee::cavalerie`

Jour 51 : Les Fichiers de la Citadelle

Contexte : Séparez le code en plusieurs fichiers.

Objectif : Modules dans des fichiers séparés

Consignes :

1. Créez `bibliotheque.rs` avec struct `Livre`
2. Dans main.rs : `mod bibliotheque;`
3. Créez un dossier `maison/` avec `mod.rs`
4. Dans maison/mod.rs, déclarez `pub mod stark;`
5. Créez `maison/stark.rs` avec fonctions publiques
6. Utilisez tout depuis main.rs

Jour 52 : Les Secrets et le Public

Contexte : Contrôlez l'accès aux fonctionnalités.

Objectif : Visibilité (pub, pub(crate), private)

Consignes :

1. Module avec fonction publique et privée
2. Struct avec champs publics et privés
3. Utilisez `pub(crate)` pour visibilité interne
4. Créez des méthodes getter/setter pour champs privés

5. Testez les différents niveaux d'accès

Jour 53 : Les Chemins du Royaume

Contexte : Naviguez dans la hiérarchie des modules.

Objectif : use, as, pub use

Consignes :

1. Importez avec `use crate::module::fonction;`
2. Renommez avec `use crate::module::Type as AutreNom;`
3. Ré-exportez avec `pub use crate::module::fonction;`
4. Utilisez `super::` pour module parent
5. Créez des imports groupés avec `{}`

Jour 54 : Les Tests de la Garde

Contexte : Vérifiez que vos fonctions marchent correctement.

Objectif : Tests unitaires avec #[cfg(test)]

Consignes :

1. Créez un module `#[cfg(test)] mod tests {}`
2. Fonction de test avec `#[test]`
3. Utilisez `assert_eq!`, `assert!`, `assert_ne!`
4. Test qui doit paniquer : `#[should_panic]`
5. Testez au moins 5 fonctions différentes
6. Exécutez avec `cargo test`

Jour 55 : Les Tests d'Intégration

Contexte : Testez comment les modules fonctionnent ensemble.

Objectif : Tests d'intégration et documentation

Consignes :

1. Créez dossier `tests/` à la racine
2. Fichier `tests/integration_test.rs`
3. Tests qui utilisent le crate comme bibliothèque externe
4. Ajoutez des doc tests dans `///` comments
5. Testez avec `cargo test --doc`

Jour 56 : PROJET - La Citadelle Complète (Multi-Module)

Contexte : Créez une simulation complète de la Citadelle avec organisation professionnelle en modules.

Objectif : Organiser un grand projet avec modules, tests et documentation

Structure du projet :

```
citadelle/
├── src/
│   ├── main.rs
│   ├── lib.rs
│   ├── maester/
│   │   ├── mod.rs
│   │   ├── apprentissage.rs
│   │   └── specialites.rs
│   ├── bibliotheque/
│   │   ├── mod.rs
│   │   ├── livres.rs
│   │   └── catalogage.rs
│   └── commun/
│       ├── mod.rs
│       └── types.rs
└── tests/
    └── integration.rs
```

Module commun (types.rs) :

```rust
pub struct Nom(String);
pub enum Specialite { Medecine, Histoire, Architecture, Astronomie, Alchimie }
```

Module maester (mod.rs, apprentissage.rs, specialites.rs) :

- Struct `Maester { nom: Nom, age: u8, chaines: Vec<Specialite> }`
- Struct `Apprenti { nom: Nom, progression: u8 }`
- Fonctions pour gérer l'apprentissage et les promotions
- Méthodes privées d'aide et méthodes publiques

Module bibliotheque (mod.rs, livres.rs, catalogage.rs) :

- Struct `Livre { titre: String, auteur: String, specialite: Specialite }`

- Struct `Catalogue { livres: Vec<Livre> }`
- Système de recherche et classification

lib.rs :

- Ré-exporte les types principaux avec `pub use`
- Documentation du crate avec `//!`

main.rs :

- Utilise tous les modules
- Crée une simulation interactive

Tests requis (dans chaque module et tests/) :

1. Tests unitaires pour chaque fonction publique
2. Tests d'apprentissage : vérifier promotions
3. Tests de catalogue : recherche, ajout, suppression
4. Tests d'intégration : scénario complet maester + livres
5. Au moins 20 tests au total

Fonctionnalités complètes :

1. Système de progression d'apprenti → maester (0-100%)
2. Attribution de chaînes selon examens réussis
3. Catalogue de 50 livres classés par spécialité
4. Recherche multi-critères dans le catalogue
5. Statistiques de la Citadelle (nombre maesters, spécialités, etc.)
6. Système de mentorat (maester → apprentis)

Consignes spécifiques :

- Tous les types principaux doivent avoir documentation `///`

- Utilisez `pub(crate)` pour fonctions internes
- Modules doivent être indépendants autant que possible
- Tests doivent couvrir > 80% du code
- Ajoutez des exemples de code dans la documentation

Scénario de démonstration (main.rs) :

1. Initialisez la Citadelle avec 10 maesters et 15 apprentis
2. Créez le catalogue avec 50 livres
3. Simulez 3 mois d'apprentissage (progressions)
4. Promouvez les apprentis qualifiés
5. Recherchez tous les livres d'Alchimie
6. Assignez des mentors aux apprentis
7. Générez un rapport complet

Bonus : Créez un module `export` qui génère un fichier JSON avec toutes les données de la Citadelle

---

📅 SEMAINE 9 : Closures et Programmation Fonctionnelle

Thème : Closures, Functional Programming, Iterator Patterns

Jour 57 : Les Fonctions Anonymes

Contexte : Créez des calculs rapides sans nommer la fonction.

Objectif : Syntaxe des closures de base

Consignes :

1. Créez une closure simple : `let double = |x| x * 2;`
2. Closure avec types explicites : `let ajouter = |x: i32, y: i32| -> i32 { x + y };`

3. Closure qui capture une variable de l'environnement
4. Testez avec différents nombres de paramètres
5. Passez une closure comme argument à une fonction

Jour 58 : Le Tri Personnalisé

Contexte : Triez des données selon différents critères.

Objectif : Closures avec sort_by et autres méthodes

Consignes :

1. Vec de tuples `(nom, age, rang)`
2. Triez par âge avec `sort_by(|a, b| ...)`
3. Triez par rang avec une closure différente
4. Utilisez `sort_by_key()` pour simplifier
5. Créez une fonction qui accepte une closure de tri personnalisée

---

Jour 59 : Les Calculateurs Spécialisés

Contexte : Créez des fonctions qui retournent des closures.

Objectif : Retourner des closures, traits Fn, FnMut, FnOnce

Consignes :

1. Fonction `creer_multiplicateur(facteur: i32) -> impl Fn(i32) -> i32`
2. Retourne une closure qui multiplie par facteur
3. Créez plusieurs multiplicateurs différents
4. Fonction avec `FnMut` qui modifie une variable capturée
5. Fonction avec `FnOnce` qui consomme la variable capturée

Jour 60 : La Chaîne de Transformations

Contexte : Transformez des données par étapes successives.

Objectif : Chaînage d'itérateurs avancé

Consignes :

1. Vec<i32> de départ
2. `.iter().filter().map().collect()` en une seule chaîne
3. Utilisez `fold()` pour accumulation
4. Utilisez `take_while()` et `skip_while()`
5. Créez un pipeline complexe de 5+ opérations

Jour 61 : Les Itérateurs Personnalisés

Contexte : Créez votre propre itérateur pour une structure.

Objectif : Implémenter le trait Iterator

Consignes :

1. Struct `Compteur { courant: u32, max: u32 }`
2. Implémentez `Iterator` pour Compteur
3. Implémentez la méthode `next(&mut self) -> Option<Self::Item>`
4. Utilisez votre itérateur dans une boucle for
5. Utilisez les méthodes d'itérateur dessus (map, filter, etc.)

Jour 62 : Le Système de Callbacks

Contexte : Créez un système d'événements avec callbacks.

Objectif : Stocker et appeler des closures

Consignes :

1. Struct `Observateur { callbacks: Vec<Box<dyn Fn(String)>> }`
2. Méthode `ajouter_callback(&mut self, f: Box<dyn Fn(String)>)`
3. Méthode `notifier(&self, message: String)`
4. Enregistrez 5 callbacks différents
5. Déclenchez la notification et observez tous les callbacks

Jour 63 : PROJET - Le Système d'Intelligence de la Main du Roi

Contexte : Créez un système d'analyse de renseignements pour la Main du Roi, utilisant des closures pour filtrer, analyser et transformer des rapports d'espions.

Objectif : Maîtriser closures, itérateurs et programmation fonctionnelle

Structures :

rust

```rust
#[derive(Debug, Clone)]
struct Rapport {
    espion: String,
    lieu: String,
    importance: u8,
    contenu: String,
    date: u32,
    verifie: bool,
}

struct SystemeIntelligence {
    rapports: Vec<Rapport>,
    filtres: Vec<Box<dyn Fn(&Rapport) -> bool>>,
    analyseurs: Vec<Box<dyn Fn(&Rapport) -> String>>,
    alertes: Vec<Box<dyn Fn(String)>>,
}
```

Méthodes sur SystemeIntelligence :

1. `new() -> Self`
2. `ajouter_rapport(&mut self, rapport: Rapport)`
3. `ajouter_filtre<F>(&mut self, filtre: F)` where F: Fn(&Rapport) -> bool + 'static
   - Stocke une closure de filtrage
4. `ajouter_analyseur<F>(&mut self, analyseur: F)` where F: Fn(&Rapport) -> String + 'static
   - Stocke une closure d'analyse
5. `ajouter_alerte<F>(&mut self, alerte: F)` where F: Fn(String) + 'static
   - Stocke un callback d'alerte
6. `rapports_filtres(&self) -> Vec<Rapport>`
   - Applique tous les filtres en chaîne
   - Utilise fold ou filter successifs
7. `analyser(&self) -> Vec<String>`
   - Applique tous les analyseurs aux rapports filtrés
   - Retourne les résultats
8. `declencher_alertes(&self, messages: Vec<String>)`
   - Appelle tous les callbacks d'alerte
9. `rapport_synthese(&self) -> String`
   - Utilise des itérateurs pour créer statistiques

Fonctions utilitaires (créateurs de filtres et analyseurs) :

1. `filtre_importance(min: u8) -> impl Fn(&Rapport) -> bool`
   - Retourne closure qui filtre par importance
2. `filtre_lieu(lieu: String) -> impl Fn(&Rapport) -> bool`
3. `filtre_recents(jours: u32, jour_actuel: u32) -> impl Fn(&Rapport) -> bool`

4. `analyseur_menaces() -> impl Fn(&Rapport) -> String`
    - Analyse si contient mots-clés: "complot", "trahison", "assassinat"
5. `analyseur_statistiques(lieu: String) -> impl Fn(&Rapport) -> String`
    - Compte rapports par lieu
6. `combinateur_filtres<F1, F2>(f1: F1, f2: F2) -> impl Fn(&Rapport) -> bool` where F1: Fn(&Rapport) -> bool, F2: Fn(&Rapport) -> bool
    - Combine deux filtres avec AND logique

Scénario :

1. Créez un SystemeIntelligence
2. Générez 50 rapports variés :
    - Différents espions (10 noms)
    - Différents lieux (King's Landing, Winterfell, Casterly Rock, etc.)
    - Importances de 1-10
    - Contenus variés (certains avec mots-clés dangereux)
    - Dates sur 30 jours
    - 70% vérifiés
3. Ajoutez 5 filtres dynamiques :
    - Importance > 7
    - Lieu = "King's Landing"
    - Rapports des 7 derniers jours
    - Seulement les vérifiés
    - Combinaison de deux filtres
4. Ajoutez 4 analyseurs :
    - Détecteur de menaces
    - Compteur par lieu

- Vérificateur d'urgence
- Extracteur de noms mentionnés

5. Ajoutez 3 alertes (closures qui "envoient" messages) :
   - Alerte console simple
   - Alerte prioritaire (si contient "urgent")
   - Compteur d'alertes (closure qui modifie compteur externe)
6. Exécutez le pipeline complet :
   - Filtrez les rapports
   - Analysez-les
   - Déclenchez les alertes appropriées
7. Générez rapport final avec :
   - Nombre total de rapports
   - Nombre après filtrage
   - Toutes les analyses
   - Recommandations

Pipeline fonctionnel complexe : Créez une fonction `analyse_complete` qui :

```rust
```

```rust
fn analyse_complete(systeme: &SystemeIntelligence) -> Vec<(String, Vec<String>)> {
    systeme.rapports_filtres()
        .iter()
        .map(|r| (r.lieu.clone(), r))
        .fold(HashMap::new(), |mut acc, (lieu, rapport)| {
            acc.entry(lieu).or_insert(Vec::new()).push(rapport);
            acc
        })
        .into_iter()
        .map(|(lieu, rapports)| {
            let analyses = rapports.iter()
                .filter(|r| r.importance > 5)
                .map(|r| format!("{}: {}", r.espion, r.contenu))
                .collect();
            (lieu, analyses)
        })
        .collect()
}
```

Bonus : Créez un système de "mots-clés surveillés" avec une closure configurable qui peut être mise à jour dynamiquement

---

📅 SEMAINE 10 : Concurrence de Base

Thème : Threads, Message Passing, Shared State

Jour 64 : Les Patrouilles Parallèles

Contexte : Envoyez plusieurs patrouilles explorer simultanément.

Objectif : Créer des threads basiques

Consignes :

1. `use std::thread;`
2. Créez un thread avec `thread::spawn(|| { ... })`

3. Utilisez `join()` pour attendre la fin
4. Créez 5 threads qui affichent des messages
5. Observez l'ordre d'exécution non-déterministe

Jour 65 : Les Messagers Concurrents

Contexte : Plusieurs messagers envoient des nouvelles simultanément.

Objectif : Channels et message passing

Consignes :

1. `use std::sync::mpsc;`
2. Créez un canal avec `mpsc::channel()`
3. Envoyez des messages depuis plusieurs threads
4. Recevez tous les messages dans le thread principal
5. Testez avec 10 threads envoyant 5 messages chacun

Jour 66 : Le Trésor Partagé

Contexte : Plusieurs threads accèdent au même coffre de manière sûre.

Objectif : Mutex<T> pour synchronisation

Consignes :

1. `use std::sync::{Arc, Mutex};`
2. Créez un `Arc<Mutex<u32>>` représentant de l'or
3. Clonez l'Arc pour chaque thread
4. Chaque thread ajoute de l'or avec `lock()`
5. Vérifiez le total final (doit être exact)

Jour 67 : La Bibliothèque Concurrente

Contexte : Plusieurs lecteurs accèdent simultanément aux données.

Objectif : Arc<T> et RwLock<T>

Consignes :

1. `use std::sync::RwLock;`
2. `Arc<RwLock<Vec<String>>>` pour une liste de livres
3. Plusieurs threads lecteurs avec `read()`
4. Un thread écrivain avec `write()`
5. Démontrez que plusieurs lectures peuvent être simultanées

Jour 68 : La Coordination des Armées

Contexte : Synchronisez plusieurs threads à un point de rendez-vous.

Objectif : Barrier et autres primitives

Consignes :

1. `use std::sync::Barrier;`
2. Créez une barrière pour 5 threads
3. Chaque thread fait du travail puis attend à la barrière
4. Tous reprennent ensemble après la barrière
5. Testez avec des durées de travail variables

Jour 69 : Le Traitement Parallèle

Contexte : Traitez de grandes quantités de données en parallèle.

Objectif : Patterns de concurrence pratiques

Consignes :

1. Vec de 1000 nombres
2. Divisez en 4 chunks
3. Traitez chaque chunk dans un thread séparé
4. Collectez les résultats avec channels
5. Comparez le temps avec version séquentielle (conceptuellement)

Jour 70 : PROJET - La Forge Royale Concurrente

Contexte : La forge royale doit produire des armes pour l'armée. Créez un système de production parallèle avec forgerons, stockage partagé et livraisons.

Objectif : Maîtriser threads, channels, Mutex et Arc dans un système réaliste

Structures :

```rust
```

```rust
use std::sync::{Arc, Mutex};
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

#[derive(Debug, Clone)]
enum Arme {
    Epee { qualite: u8 },
    Lance { qualite: u8 },
    Arc { qualite: u8 },
}

#[derive(Debug)]
struct Stock {
    armes: Vec<Arme>,
    capacite_max: usize,
}

struct Forgeron {
    id: u32,
    nom: String,
    specialite: String,
    vitesse: u64, // millisecondes par arme
}

struct Forge {
    stock: Arc<Mutex<Stock>>,
    commandes_restantes: Arc<Mutex<u32>>,
}
```

Implémentations requises :

1. **Stock :**
   - `new(capacite: usize) -> Self`
   - `ajouter_arme(&mut self, arme: Arme) -> Result<(), String>`

- `retirer_arme(&mut self) -> Option<Arme>`
- `est_plein(&self) -> bool`
- `nombre_armes(&self) -> usize`

2. **Forgeron :**
   - `new(id: u32, nom: String, specialite: String, vitesse: u64) -> Self`
   - `forger(&self) -> Arme` - crée une arme selon la spécialité
3. **Forge :**
   - `new(capacite_stock: usize, commandes: u32) -> Self`

Fonctions de threads :

1. `thread_forgeron(forgeron: Forgeron, stock: Arc<Mutex<Stock>>, commandes: Arc<Mutex<u32>>)`
   - Boucle : vérifie s'il reste des commandes
   - Forge une arme (avec sleep pour simuler le temps)
   - Attend si le stock est plein
   - Ajoute l'arme au stock
   - Décrémente les commandes restantes
2. `thread_livreur(stock: Arc<Mutex<Stock>>, sender: mpsc::Sender<Arme>)`
   - Boucle : retire des armes du stock
   - Envoie via le channel
   - Simule le temps de livraison
3. `thread_inspecteur(receiver: mpsc::Receiver<Arme>, stats: Arc<Mutex<Statistiques>>)`
   - Reçoit les armes livrées
   - Vérifie la qualité
   - Met à jour les statistiques

Struct Statistiques :

```rust
struct Statistiques {
    total_livrees: u32,
    par_type: HashMap<String, u32>,
    qualite_moyenne: f64,
}
```

Scénario principal :

1. Créez une forge avec :
   - Stock capacité 20 armes
   - 100 commandes à remplir
2. Créez 5 forgerons :
   - 2 spécialisés en épées (vitesse 100ms)
   - 2 spécialisés en lances (vitesse 120ms)
   - 1 spécialisé en arcs (vitesse 150ms)
3. Lancez les threads :
   - 5 threads forgerons
   - 2 threads livreurs
   - 1 thread inspecteur
   - 1 thread moniteur (affiche état toutes les secondes)
4. Channel pour communication :
   - Forgerons → Stock (via Mutex)
   - Livreurs → Inspecteur (via mpsc)
5. Thread moniteur affiche régulièrement :
   - Nombre d'armes dans le stock

- Commandes restantes
- Armes livrées

6. Attendez que toutes les commandes soient remplies
7. Rapport final :
    - Total d'armes produites par type
    - Qualité moyenne par type
    - Temps total (simulé)
    - Efficacité de chaque forgeron

Défis supplémentaires :

- Gérez correctement les deadlocks potentiels
- Assurez-vous que les threads se terminent proprement
- Utilisez drop(guard) pour libérer les locks rapidement
- Implémentez une shutdown propre avec un signal

Bonus :

1. Ajoutez un système de "rupture de matériaux" où parfois la forge doit attendre des matériaux (channel supplémentaire)
2. Thread "fournisseur" qui envoie des matériaux périodiquement
3. Forgerons doivent attendre les matériaux avant de forger

📅 SEMAINE 11 : Async/Await et Futures

Thème : Asynchronous Programming, Tokio, Futures

Jour 71 : Les Promesses du Royaume

Contexte : Introduisez les concepts de programmation asynchrone.

Objectif : Comprendre Future et async/await de base

Consignes :

1. Ajoutez tokio : `tokio = { version = "1", features = ["full"] }`
2. Créez une fonction `async fn attendre_corbeau() -> String`
3. Utilisez `tokio::time::sleep()` pour simuler l'attente
4. Fonction main avec `#[tokio::main]`
5. Appelez avec `.await`

Jour 72 : Les Missions Parallèles

Contexte : Exécutez plusieurs tâches asynchrones simultanément.

Objectif : Concurrence avec tokio::join! et spawn

Consignes :

1. Créez 3 fonctions async différentes
2. Utilisez `tokio::join!` pour exécuter ensemble
3. Utilisez `tokio::spawn` pour tâches indépendantes
4. Collectez les résultats avec handles
5. Comparez avec exécution séquentielle

Jour 73 : Le Marché Asynchrone

Contexte : Gérez des transactions concurrentes de manière asynchrone.

Objectif : Channels asynchrones (tokio::sync::mpsc)

Consignes :

1. `use tokio::sync::mpsc;`

2. Créez un canal async
3. Plusieurs tâches envoient des messages
4. Une tâche reçoit et traite
5. Utilisez `select!` pour gérer plusieurs canaux

Jour 74 : La Sentinelle Partagée

Contexte : Partagez des données entre tâches asynchrones.

Objectif : Mutex asynchrone et Arc

Consignes :

1. `use tokio::sync::Mutex;`
2. Créez `Arc<Mutex<T>>` pour données partagées
3. Plusieurs tâches async accèdent aux données
4. Utilisez `.lock().await`
5. Comparez avec Mutex standard de std

Jour 75 : Le Timeout des Négociations

Contexte : Limitez le temps d'attente des opérations.

Objectif : Timeout et select!

Consignes :

1. Utilisez `tokio::time::timeout()`
2. Gérez les Result de timeout
3. Utilisez `tokio::select!` pour choisir entre plusieurs futures
4. Créez une fonction qui tente plusieurs stratégies

5. Implémentez retry avec timeout

Jour 76 : Les Streams de Données

Contexte : Traitez des flux continus d'informations.

Objectif : Async streams et StreamExt

Consignes :

1. `use tokio_stream::{self as stream, StreamExt};`
2. Créez un stream avec `stream::iter()`
3. Utilisez `.map()`, `.filter()` sur le stream
4. Consommez avec `.next().await`
5. Créez un stream personnalisé

Jour 77 : PROJET - Le Réseau d'Espionnage Asynchrone

Contexte : Créez un réseau d'espions distribuant des informations en temps réel à travers Westeros, utilisant async/await pour gérer des milliers d'agents simultanément.

Objectif : Maîtriser la programmation asynchrone dans un système complexe

Configuration Cargo.toml :

```toml
[dependencies]
tokio = { version = "1", features = ["full"] }
tokio-stream = "0.1"
```

Structures :

```rust
```

```rust
use tokio::sync::{mpsc, Mutex};
use tokio::time::{sleep, Duration, timeout};
use std::sync::Arc;
use std::collections::HashMap;

#[derive(Debug, Clone)]
struct Intelligence {
    agent_id: String,
    location: String,
    info: String,
    priorite: u8,
    timestamp: u64,
}

struct Agent {
    id: String,
    nom: String,
    location: String,
    actif: bool,
    vitesse_collecte: u64, // ms entre rapports
}

struct MaitreEspion {
    agents: Arc<Mutex<HashMap<String, Agent>>>,
    canal_intelligence: mpsc::Sender<Intelligence>,
    statistiques: Arc<Mutex<Stats>>,
}

struct Stats {
    total_rapports: u32,
    par_location: HashMap<String, u32>,
    alertes_urgentes: u32,
}
```

Fonctions async requises :

1. **Agent :**

rust

```rust
async fn agent_collecte(
    agent: Agent,
    sender: mpsc::Sender<Intelligence>,
    duree_mission: u64
) {
    // Boucle de collecte d'intelligence
    // Sleep selon vitesse_collecte
    // Envoie périodiquement des Intelligence
    // S'arrête après duree_mission secondes
}
```

2. **Centre de traitement :**

rust

```rust
async fn centre_analyse(
    mut receiver: mpsc::Receiver<Intelligence>,
    stats: Arc<Mutex<Stats>>,
    alerte_sender: mpsc::Sender<String>
) {
    // Reçoit les intelligences
    // Met à jour les stats
    // Envoie alertes si priorité > 7
}
```

3. **Système d'alertes :**

rust

```rust
async fn gestionnaire_alertes(
    mut receiver: mpsc::Receiver<String>
) {
    // Traite les alertes urgentes
    // Simule notifications
}
```

4. **Surveillance réseau :**

rust

```rust
async fn moniteur_reseau(
    agents: Arc<Mutex<HashMap<String, Agent>>>,
    stats: Arc<Mutex<Stats>>,
    interval: u64
) {
    // Affiche périodiquement l'état du réseau
    // Nombre d'agents actifs
    // Stats en temps réel
}
```

5. **Coordinateur de missions :**

rust

```rust
async fn coordinateur_mission(
    maitre: Arc<MaitreEspion>,
    missions: Vec<(String, String, u64)> // (agent_id, target_location, duree)
) -> Result<(), String> {
    // Assigne des missions aux agents
    // Avec timeout pour chaque assignation
    // Utilise tokio::select! pour gérer plusieurs assignations
}
```

6. **Collecteur distribué :**

rust

```rust
async fn collecte_parallele(
    agents: Vec<Agent>,
    sender: mpsc::Sender<Intelligence>
) {
    // Lance tous les agents en parallèle avec tokio::spawn
    // Collecte tous les handles
    // Attend la fin de tous avec join_all ou futures::future::join_all
}
```

Scénario principal (dans main) :

rust

```rust
#[tokio::main]
async fn main() {
    // 1. Créez 20 agents dans différentes locations
    let agents = creer_agents();

    // 2. Setup des channels
    let (intel_tx, intel_rx) = mpsc::channel(100);
    let (alerte_tx, alerte_rx) = mpsc::channel(32);

    // 3. Stats partagées
    let stats = Arc::new(Mutex::new(Stats::new()));

    // 4. Lancez les tâches async :
    let handle_analyse = tokio::spawn(centre_analyse(intel_rx, stats.clone(), alerte_tx));
    let handle_alertes = tokio::spawn(gestionnaire_alertes(alerte_rx));
    let handle_moniteur = tokio::spawn(moniteur_reseau(agents.clone(), stats.clone(), 2000));

    // 5. Lancez 20 agents en parallèle
    let handles_agents = agents.into_iter().map(|agent| {
        let tx = intel_tx.clone();
        tokio::spawn(agent_collecte(agent, tx, 30))
    }).collect::<Vec<_>>();

    // 6. Attendez que tous les agents finissent (30 secondes)
    for handle in handles_agents {
        let _ = handle.await;
    }

    // 7. Fermez les channels
    drop(intel_tx);

    // 8. Attendez les autres tâches
    let _ = tokio::join!(handle_analyse, handle_alertes);

    // 9. Rapport final
```

```
    afficher_rapport_final(stats).await;
}
```

Fonctionnalités avancées :

1. **Gestion de timeout :**

rust

```rust
async fn mission_avec_timeout(
    agent: Agent,
    timeout_ms: u64
) -> Result<Vec<Intelligence>, String> {
    timeout(Duration::from_millis(timeout_ms),
            collecter_intelligence(agent))
        .await
        .map_err(|_| "Timeout".to_string())?
}
```

2. **Select entre plusieurs sources :**

rust

```rust
async fn select_intelligence(
    rx1: &mut mpsc::Receiver<Intelligence>,
    rx2: &mut mpsc::Receiver<Intelligence>
) {
    tokio::select! {
        Some(intel) = rx1.recv() => traiter(intel),
        Some(intel) = rx2.recv() => traiter(intel),
        else => println!("Tous les canaux fermés"),
    }
}
```

3. **Stream processing :**

rust

```rust
use tokio_stream::StreamExt;

async fn analyser_stream(intelligence_stream: impl Stream<Item = Intelligence>) {
    let urgent = intelligence_stream
        .filter(|i| i.priorite > 7)
        .map(|i| format!("URGENT: {}", i.info));

    tokio::pin!(urgent);

    while let Some(msg) = urgent.next().await {
        println!("{}", msg);
    }
}
```

Tests et métriques :

1. Nombre total d'intelligences collectées
2. Distribution par location
3. Temps de réponse moyen pour alertes urgentes
4. Taux de succès des missions avec timeout
5. Agents les plus productifs

Bonus :

1. Implémentez un système de "résurrection" : si un agent "meurt" (panic), relancez-le automatiquement
2. Ajoutez un rate limiter pour éviter de surcharger le système
3. Créez un dashboard en temps réel (affichage console avancé)

📅 SEMAINE 12 : Projet Final Intégrateur

Thème : Tout intégrer dans un projet complet

Jour 78-84 : PROJET FINAL - Le Trône de Fer : Simulateur Politique Complet

Contexte : Le Grand Conseil vous charge de créer un simulateur complet de la politique de Westeros. Le système doit gérer les alliances, les trahisons, les armées, l'économie, et prédire qui survivra au Jeu des Trônes.

Objectif : Intégrer TOUS les concepts appris en 12 semaines

JOUR 78 : Architecture et Structures de Base

Modules à créer :

```
jeu_des_trones/
├── src/
│   ├── main.rs
│   ├── lib.rs
│   ├── modeles/
│   │   ├── mod.rs
│   │   ├── maison.rs
│   │   ├── seigneur.rs
│   │   ├── armee.rs
│   │   └── economie.rs
│   ├── systemes/
│   │   ├── mod.rs
│   │   ├── alliances.rs
│   │   ├── combats.rs
│   │   └── politique.rs
│   ├── simulation/
│   │   ├── mod.rs
│   │   ├── engine.rs
│   │   └── evenements.rs
│   └── utils/
│       ├── mod.rs
│       └── erreurs.rs
└── tests/
    ├── integration_test.rs
    └── simulation_test.rs
```

Structures fondamentales :

rust

```rust
// modeles/maison.rs
#[derive(Debug, Clone, PartialEq)]
pub struct Maison {
    pub nom: String,
    pub devise: String,
    pub region: Region,
    pub richesse: u64,
    pub prestige: u32,
}

#[derive(Debug, Clone, PartialEq)]
pub enum Region {
    Nord,
    Terres Ouest,
    Val,
    Terres Couronne,
    Reach,
    Terres Orage,
    Dorne,
    Iles Fer,
}

// modeles/seigneur.rs
pub struct Seigneur {
    pub nom: String,
    pub maison: String,
    pub age: u8,
    pub traits: Vec<Trait>,
    pub competences: Competences,
    pub relations: HashMap<String, Relation>,
}

#[derive(Debug, Clone)]
pub enum Trait {
    Ambitieux,
    Loyal,
    Courageux,
```

```
    Ruse,
    Cruel,
    Juste,
    Diplomatique,
}

pub struct Competences {
    pub combat: u8,
    pub diplomatie: u8,
    pub intrigue: u8,
    pub gestion: u8,
    pub commandement: u8,
}

// modeles/armee.rs
pub struct Armee {
    pub infanterie: u32,
    pub cavalerie: u32,
    pub archers: u32,
    pub moral: u8,
    pub equipement: u8,
}

// systemes/alliances.rs
pub struct Alliance {
    pub maisons: Vec<String>,
    pub type_alliance: TypeAlliance,
    pub force: u8,
    pub duree: u32,
}

#[derive(Debug, Clone)]
pub enum TypeAlliance {
    Mariage,
    Pacte,
    Vassalage,
```

```
    Militaire,
}
```

Consignes Jour 78 :

1. Créez toute la structure de modules
2. Implémentez tous les types de base
3. Ajoutez derives appropriés (Debug, Clone, etc.)
4. Créez des constructeurs (new) pour chaque struct
5. Tests unitaires pour chaque module

JOUR 79 : Systèmes d'Alliances et Relations

Implémentez :

```rust
```

```rust
// systemes/alliances.rs
pub struct GestionnaireAlliances {
    alliances: Vec<Alliance>,
    trahisons: Vec<Trahison>,
}

impl GestionnaireAlliances {
    pub fn former_alliance(&mut self, m1: &str, m2: &str, type_a: TypeAlliance) -> Result<(), ErreurP
    pub fn rompre_alliance(&mut self, m1: &str, m2: &str) -> Result<(), ErreurPolitique>
    pub fn evaluer_stabilite(&self, maison: &str) -> u8
    pub fn allies_de(&self, maison: &str) -> Vec<String>
    pub fn ennemis_de(&self, maison: &str) -> Vec<String>
    pub fn detecter_complot(&self) -> Vec<Complot>
}

pub struct Complot {
    pub meneurs: Vec<String>,
    pub cible: String,
    pub probabilite_succes: f64,
}

// systemes/politique.rs
pub struct SystemePolitique {
    seigneurs: HashMap<String, Seigneur>,
    influences: HashMap<String, u32>,
}

impl SystemePolitique {
    pub fn calculer_influence(&self, maison: &str) -> u32
    pub fn simuler_negociation(&self, s1: &Seigneur, s2: &Seigneur) -> ResultatNegociation
    pub fn evenement_politique(&mut self) -> Evenement
}
```

Consignes Jour 79 :

1. Implémentez le système d'alliances complet

2. Créez l'algorithme de détection de complots

3. Système de calcul d'influence basé sur : richesse, armée, alliances, prestige

4. Tests : formez 10 alliances, rompez-en 3, détectez complots

5. Utilisez traits personnalisés (Negociable, Influent, etc.)

JOUR 80 : Système de Combat et Armées

Implémentez :

rust

```rust
// systemes/combats.rs
pub struct SystemeCombat {
    batailles_historique: Vec<Bataille>,
}

pub struct Bataille {
    pub nom: String,
    pub lieu: String,
    pub attaquant: String,
    pub defenseur: String,
    pub armee_attaque: Armee,
    pub armee_defense: Armee,
    pub resultat: Option<ResultatBataille>,
}

pub struct ResultatBataille {
    pub vainqueur: String,
    pub pertes_attaquant: u32,
    pub pertes_defenseur: u32,
    pub butin: u64,
}

impl SystemeCombat {
    pub fn simuler_bataille(&mut self, b: &mut Bataille) -> ResultatBataille
    pub fn calculer_force_militaire(&self, armee: &Armee) -> u32
    pub fn appliquer_bonus_terrain(&self, armee: &Armee, terrain: Terrain) -> f64
    pub fn calculer_pertes(&self, force1: u32, force2: u32) -> (u32, u32)
}
```

Algorithme de combat :

1. Force brute = infanterie + (cavalerie * 1.5) + (archers * 1.2)
2. Multiplicateurs : moral (0.5-1.5), équipement (0.7-1.3), terrain
3. Compare les forces et détermine le vainqueur
4. Calcule les pertes proportionnellement

5. Génère événements aléatoires (embuscade, trahison, etc.)

Consignes Jour 80 :

1. Implémentez la simulation de bataille complète
2. Créez 5 types de terrain différents avec bonus
3. Simulez 20 batailles avec résultats variés
4. Historique des batailles avec statistiques
5. Tests de tous les cas (victoire écrasante, pyrrhique, etc.)

JOUR 81 : Économie et Ressources

Implémentez :

rust

```rust
// modeles/economie.rs
pub struct Economie {
    pub or: u64,
    pub nourriture: u32,
    pub ressources: HashMap<Ressource, u32>,
    pub revenus_annuels: u64,
    pub depenses_annuelles: u64,
}

#[derive(Debug, Clone, Hash, Eq, PartialEq)]
pub enum Ressource {
    Fer,
    Bois,
    Pierre,
    Tissu,
}

impl Economie {
    pub fn calculer_revenus(&self, maison: &Maison, armee: &Armee) -> u64
    pub fn payer_armee(&mut self, armee: &Armee) -> Result<(), ErreurEconomique>
    pub fn commercer(&mut self, autre: &mut Economie, ressource: Ressource, quantite: u32, prix: u6
    pub fn investir_dans_armee(&mut self, armee: &mut Armee, montant: u64) -> Result<(), ErreurEc
}

pub struct Marche {
    prix: HashMap<Ressource, u64>,
    offre: HashMap<Ressource, u32>,
    demande: HashMap<Ressource, u32>,
}

impl Marche {
    pub fn ajuster_prix(&mut self)
    pub fn simuler_commerce(&mut self, economies: &mut [Economie])
}
```

Consignes Jour 81 :

1. Système économique complet avec inflation/déflation
2. Commerce entre maisons avec négociation de prix
3. Coût d'entretien des armées
4. Investissements et retours
5. Crises économiques aléatoires

JOUR 82 : Moteur de Simulation Asynchrone

Implémentez :

rust

```rust
// simulation/engine.rs
use tokio::sync::{Mutex, mpsc};
use std::sync::Arc;

pub struct MoteurSimulation {
    etat: Arc
```