APPRENDRE À PROGRAMMER EN RUST : POUR LES NULS

Table des matières

Introduction

Ce livre est une introduction à la programmation en Rust pour les débutants. Il est conçu pour les personnes qui n'ont aucune expérience en programmation, mais qui veulent apprendre à programmer. Le livre est divisé en plusieurs chapitres, chacun couvrant un aspect différent de la programmation en Rust. Chaque chapitre contient des exemples de code, des exercices et des quiz pour vous aider à apprendre.

Pourquoi choisir Rust ?

Rust est un langage de programmation moderne conçu pour être sûr, rapide et concurrent. Il est idéal pour les applications qui nécessitent une performance élevée et une sécurité de la mémoire. Rust est également un langage polyvalent qui peut être utilisé pour une variété de tâches, allant de la programmation système à la programmation web.

À qui s'adresse ce livre ?

Ce livre s'adresse à toute personne intéressée par la programmation en Rust, qu'elle soit débutante ou expérimentée. Il est conçu pour les personnes qui veulent apprendre à programmer en Rust, mais qui n'ont aucune expérience en programmation. Le livre couvre les bases de la programmation en Rust, y compris les types de données, les fonctions, les boucles, les structures de contrôle, la gestion de la mémoire, la programmation concurrente et la gestion des erreurs.

Comment utiliser ce livre

Ce livre est conçu pour être utilisé comme un guide d'apprentissage autonome. Chaque chapitre couvre un aspect différent de la programmation en Rust, et contient des exemples de code, des exercices et des quiz pour vous aider à apprendre. Vous pouvez lire le livre de manière linéaire, en suivant les chapitres dans l'ordre, ou vous pouvez sauter directement aux chapitres qui vous intéressent.

Sécurité de la mémoire

Rust est conçu pour être sûr, ce qui signifie qu'il est difficile de faire des erreurs de programmation qui pourraient causer des problèmes de sécurité ou de stabilité. Rust utilise un système de gestion de la mémoire innovant qui garantit que les programmes ne peuvent pas accéder à la mémoire de manière incorrecte ou dangereuse. Rust est un langage de programmation moderne conçu pour être sûr, rapide et concurrent. Il est idéal pour les applications qui nécessitent une performance élevée et une sécurité de la mémoire. Rust est également un langage polyvalent qui peut être utilisé pour une variété de tâches, allant de la programmation système à la programmation web.

Concurrence sans crainte

Rust est conçu pour faciliter la programmation concurrente, ce qui signifie que vous pouvez écrire des programmes qui exécutent plusieurs tâches en parallèle sans craindre les problèmes de concurrence. Rust utilise un modèle de mémoire partagée innovant qui garantit que les programmes concurrents sont sûrs et fiables. La concurrence est un aspect important de la programmation moderne, car de nombreux programmes doivent exécuter plusieurs tâches en parallèle pour être efficaces. Rust est conçu pour faciliter la programmation concurrente, ce qui signifie que vous pouvez écrire des programmes qui exécutent plusieurs tâches en parallèle sans craindre les problèmes de concurrence.

Performance

Rust est conçu pour être rapide, ce qui signifie qu'il peut exécuter des programmes à une vitesse élevée. Rust utilise un système de compilation avancé qui optimise automatiquement le code pour obtenir les meilleures performances possibles. Cela signifie que vous pouvez écrire des programmes en Rust qui sont aussi rapides que des programmes écrits dans des langages de programmation bas niveau comme le C ou le C++.

Cas d'utilisation et entreprises qui utilisent Rust

Rust est utilisé par de nombreuses entreprises pour développer des logiciels critiques, y compris des systèmes d'exploitation, des navigateurs web, des bases de données et des outils de développement. Rust est également utilisé par de nombreuses startups pour développer des applications web, des jeux vidéo et des logiciels de sécurité. Rust est un langage polyvalent qui peut être utilisé pour une variété de tâches, allant de la programmation système à la programmation web.

Installation et environnement de développement

Avant de commencer à programmer en Rust, vous devez installer Rust selon votre système d'exploitation et configurer votre environnement de développement. Dans ce chapitre, nous allons vous montrer comment installer Rust et `cargo` (le gestionnaire de paquets de Rust), configurer votre éditeur de code et compiler et exécuter un programme Rust simple. La documentation officielle de Rust est également une ressource utile pour l'installation et la configuration de l'environnement de développement.

Installation de Rust et de cargo (gestionnaire de paquets)

Pour installer Rust et cargo, vous pouvez utiliser le site officiel de Rust. Vous y trouverez des instructions détaillées pour installer Rust sur Windows, macOS et Linux. Une fois que vous avez installé Rust, vous aurez accès à la commande `rustc` (le compilateur Rust) et à la commande `cargo` (le gestionnaire de paquets de Rust).

Configuration de l'éditeur de code

Rust est pris en charge par de nombreux éditeurs de code, y compris Visual Studio Code, IntelliJ IDEA, RustRover, Sublime Text, Vim et Emacs. Vous pouvez trouver des extensions pour ces éditeurs qui ajoutent des fonctionnalités de Rust, comme la coloration syntaxique, l'auto-complétion, le débogage et la gestion de projet. Vous pouvez également utiliser un éditeur de code en ligne comme Repl.it pour écrire et exécuter des programmes Rust sans installer quoi que ce soit sur votre ordinateur.

Compilation et exécution d'un programme Rust simple

Pour créer un nouveau projet Rust, vous pouvez utiliser la commande : `cargo new nom_du_projet`, puis la commande `cargo run` pour compiler et exécuter le programme. Par exemple, voici un programme Rust simple qui affiche "Hello, world!" à l'écran :

```
cargo new hello_world
cd hello_world
```

Cette commande crée un nouveau projet Rust appelé `hello_world`. Ensuite, vous pouvez ouvrir le fichier `src/main.rs` dans votre éditeur de code et y écrire le code suivant :

Voyons la structure de ce programme de notre projet `hello_world` :

```
hello_world/
│
├── Cargo.toml
│
└── src/
    └── main.rs
```

Détails des composants

- **`Cargo.toml`** : Ce fichier contient les métadonnées de votre projet et les dépendances. Pour un projet `hello_world`, le contenu de `Cargo.toml` pourrait ressembler à cela :

```
[package]
name = "hello_world"
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-
lang.org/cargo/reference/manifest.html

[dependencies]
```

- **src/** : Ce dossier contient le code source de votre projet.
- **main.rs** : C'est le point d'entrée de votre application. Pour un simple "Hello, world!", le contenu de main.rs serait :

```
fn main() {
    println!("Hello, world!");
}
```

Étudions ce code en détail :

- `fn main() { ... }` : C'est la fonction principale de votre programme. Tous les programmes Rust commencent par une fonction `main`. Le code entre les accolades `{ ... }` est le corps de la fonction `main`.
- `println!("Hello, world!");` : C'est une macro Rust qui affiche du texte à l'écran. Dans ce cas, la macro `println!` affiche "Hello, world!" à l'écran.

Pour compiler et exécuter ce programme, vous pouvez utiliser la commande `cargo run` dans le terminal :

```
cargo run
```

Dans le terminal, vous verrez la sortie suivante :

```
   Compiling hello_world v0.1.0 (file:///path/to/your/project/hello_world)
    Finished dev [unoptimized + debuginfo] target(s) in 0.34s
     Running `target/debug/hello_world`
Hello, world!
```

Félicitations ! Vous venez de compiler et d'exécuter votre premier programme Rust. Vous êtes maintenant prêt à commencer à apprendre les bases de la programmation en Rust.

Fondamentaux de Rust

Dans ce chapitre, nous allons couvrir les bases de la programmation en Rust. Nous allons commencer par les variables et la mutabilité, puis nous aborderons les types de données, les fonctions, les commentaires et les exemples de programmes Rust simples.

Les bases de Rust

Les bases de Rust comprennent les variables et la mutabilité, les types de données, les fonctions et les commentaires. Ces concepts sont essentiels pour comprendre la programmation en Rust.

Variables et mutabilité

En Rust, les variables sont créées en utilisant le mot-clé `let`. Par défaut, les variables sont immuables, ce qui signifie qu'une fois qu'elles ont une valeur, cette valeur ne peut pas être modifiée. Voici un exemple de déclaration de variable immuable en Rust :

```
fn main() {
    let x = 5;
    println!("La valeur de x est : {}", x);
}
```

Dans cet exemple, `x` est une variable immuable qui contient la valeur `5`. Si vous essayez de modifier la valeur de `x`, vous obtiendrez une erreur de compilation.

Par exemple, le code suivant générera une erreur de compilation :

```
fn main() {
    let x = 5;
    x = 10; // Erreur : tentative de modification d'une variable immuable
    println!("La valeur de x est : {}", x);
}
```

Pour rendre une variable mutable, vous pouvez utiliser le mot-clé `mut` lors de sa déclaration. Voici un exemple de déclaration de variable mutable en Rust :

La `mutabilité` est une caractéristique importante de Rust qui garantit que les programmes sont sûrs et fiables. En Rust, les variables sont immuables par défaut, ce qui signifie qu'une fois qu'elles ont une valeur, cette valeur ne peut pas être modifiée. Cela garantit que les programmes ne peuvent pas accéder à la mémoire de manière incorrecte ou dangereuse.

```
fn main() {
    let mut x = 5;
    x = 10; // Pas d'erreur : x est une variable mutable
```

```
    println!("La valeur de x est : {}", x);
}
```

Dans cet exemple, `x` est une variable mutable qui contient la valeur `5`. Vous pouvez modifier la valeur de `x` en utilisant l'opérateur d'affectation `=`. Le code ci-dessus affichera "La valeur de x est : 10" à l'écran.

Types de données (Scalars et compound types)

Rust prend en charge plusieurs types de données, y compris les types scalaires (entiers, flottants, booléens, caractères) et les types composés (tuples, tableaux, chaînes de caractères, pointeurs). Voici quelques exemples de types de données en Rust :

Les types de données scalaires comprennent les entiers, les flottants, les booléens et les caractères

Les types de données scalaires sont des types de données qui représentent une seule valeur. Les types de données composés sont des types de données qui représentent plusieurs valeurs regroupées. Les types de données scalaires comprennent les entiers, les flottants, les booléens et les caractères.

- Entiers : `i8`, `i16`, `i32`, `i64`, `u8`, `u16`, `u32`, `u64` :
  - Les entiers signés (`i`) peuvent stocker des nombres négatifs et positifs.
  - Les entiers non signés (`u`) ne peuvent stocker que des nombres positifs.
  - Le nombre après le `i` ou le `u` indique la taille de l'entier en bits (par exemple, `i8` est un entier signé de 8 bits).
  - Exemple : `let x: i32 = 5;` (déclare une variable `x` de type `i32` avec la valeur `5` avec une taille de 32 bits).

Exemple :

```
fn main() {
    let x: i32 = 5;
    let y: i8 = -10;
    let z: u8 = 10;
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
    println!("La valeur de y est : {}", y); // La valeur de y est : -10
    println!("La valeur de z est : {}", z); // La valeur de z est : 10
}
```

```
fn main() {
    let x: u32 = -5; // Un entier non signé de 32 bits avec une valeur
négative (erreur de compilation)
    println!("La valeur de x est : {}", x); // Erreur : tentative de
stocker un nombre négatif dans un entier non signé
}
```

Pour comprendre la taille des types de données, vous pouvez consulter la documentation officielle de Rust ou avoir la notion du code binaire sachant que 1 octet = 8 bits.

- Flottants : `f32`, `f64` :
  - Les flottants sont utilisés pour stocker des nombres à virgule flottante ou réels.
  - `f32` est un flottant de 32 bits et `f64` est un flottant de 64 bits.
  - Exemple : `let x: f64 = 5.0;` (déclare une variable `x` de type `f64` avec la valeur `5.0`).

Exemple :

```
fn main() {
    let x: f64 = 5.5; // Un flottant de 64 bits
    println!("La valeur de x est : {}", x); // La valeur de x est : 5.5
}
```

- Booléens : `bool` :
  - Les booléens sont utilisés pour stocker des valeurs de vérité (vrai ou faux). - Exemple : `let x: bool = true;` (déclare une variable `x` de type `bool` avec la valeur `true`). Exemple :

```
fn main() {
    let x: bool = true;
    let y: bool = false;
    println!("La valeur de x est : {}", x); // La valeur de x est : true
    println!("La valeur de y est : {}", y); // La valeur de y est : false
}
```

- Caractères : `char` :
  - Les caractères sont utilisés pour stocker des caractères Unicode. Ce qui signifie que Rust prend en charge les caractères de toutes les langues et les symboles. Un seul caractère Unicode est stocké dans un type `char`.
  - Exemple : `let x: char = 'a';` (déclare une variable `x` de type `char` avec la valeur `'a'`). Exemple :

```
fn main() {
    let x: char = 'a';
    println!("La valeur de x est : {}", x); // La valeur de x est : a
}
```

Les types de données composés comprennent les tuples, les tableaux, les chaînes de caractères et les pointeurs

Les types de données composés sont des types de données qui représentent plusieurs valeurs regroupées. Les types de données composés comprennent les tuples, les tableaux, les chaînes de caractères et les pointeurs.

- Tuples :

- Les tuples sont utilisés pour stocker plusieurs valeurs de types différents. Un tuple est déclaré en écrivant les valeurs entre parenthèses et en les séparant par des virgules.
- Exemple : `let x: (i32, f64, bool) = (5, 5.5, true);` (déclare une variable `x` de type tuple `(i32, f64, bool)` avec les valeurs `5`, `5.5` et `true`).
- Pour accéder aux valeurs d'un tuple, vous pouvez utiliser la notation de point `.` suivie de l'index de la valeur dans le tuple (en commençant par `0`).

Exemple :

```
fn main() {
    let x: (i32, f64, bool) = (5, 5.5, true);
    println!("La première valeur de x est : {}", x.0); // La première
valeur de x est : 5
    println!("La deuxième valeur de x est : {}", x.1); // La deuxième
valeur de x est : 5.5
    println!("La troisième valeur de x est : {}", x.2); // La troisième
valeur de x est : true
}
```

Cette manière de déclarer les tuples est appelée `déstructuration` des tuples.

- Tableaux :
  - Les tableaux sont utilisés pour stocker plusieurs valeurs de même type. Un tableau est déclaré en écrivant les valeurs entre crochets et en les séparant par des virgules.
  - Exemple : `let x: [i32; 5] = [1, 2, 3, 4, 5];` (déclare une variable `x` de type tableau `[i32; 5]` avec les valeurs `1`, `2`, `3`, `4` et `5`).
  - Pour accéder aux valeurs d'un tableau, vous pouvez utiliser la notation de crochets `[]` suivie de l'index de la valeur dans le tableau (en commençant par `0`).

Exemple :

```
fn main() {
    let x: [i32; 5] = [1, 2, 3, 4, 5]; // Un tableau de 5 entiers de 32
bits
    println!("La première valeur de x est : {}", x[0]); // La première
valeur de x est : 1
    println!("La deuxième valeur de x est : {}", x[1]); // La deuxième
valeur de x est : 2
    println!("La troisième valeur de x est : {}", x[2]); // La troisième
valeur de x est : 3
}
```

Voyons en détail cette ligne de code : `let x: [i32; 5] = [1, 2, 3, 4, 5];`

- `let x` : déclare une variable `x`.
- `: [i32; 5]` : indique que `x` est un tableau de 5 entiers de 32 bits.

- `= [1, 2, 3, 4, 5]` : initialise `x` avec les valeurs `1`, `2`, `3`, `4` et `5`.
- `x[0]` : accède à la première valeur de `x` (qui est `1`).
- `x[1]` : accède à la deuxième valeur de `x` (qui est `2`).
- `x[2]` : accède à la troisième valeur de `x` (qui est `3`).
- `x[3]` : accède à la quatrième valeur de `x` (qui est `4`).
- `x[4]` : accède à la cinquième valeur de `x` (qui est `5`).
- `x[5]` : accède à la sixième valeur de `x` (qui est `erreur de compilation`).
- Chaînes de caractères : littérales (`&str`) et dynamiques (`String`) : Les chaînes de caractères sont utilisées pour stocker du texte. En Rust, les chaînes de caractères sont de deux types : les chaînes de caractères littérales (`&str`) et les chaînes de caractères dynamiques (`String`).

Chaînes de caractères littérales (`&str`) :

Les chaînes de caractères littérales (`&str`) sont des chaînes de caractères statiques qui sont stockées dans le programme lui-même. - Exemple : `let x: &str = "Hello, world!";` (déclare une variable `x` de type chaîne de caractères littérale `&str` avec la valeur `"Hello, world!"`).

Exemple :

```
fn main() {
    let x: &str = "Hello, world!"; // Une chaîne de caractères littérale
    println!("La valeur de x est : {}", x); // La valeur de x est : Hello,
world!
}
```

Chaînes de caractères dynamiques (`String`) :

Les chaînes de caractères dynamiques (`String`) sont des chaînes de caractères qui peuvent être modifiées à l'exécution. - Exemple : `let x: String = String::from("Hello, world!");` (déclare une variable `x` de type chaîne de caractères dynamique `String` avec la valeur `"Hello, world!"`).

Exemple :

```
fn main() {
    let x: String = String::from("Hello, world!"); // Une chaîne de
caractères dynamique
    println!("La valeur de x est : {}", x); // La valeur de x est : Hello,
world!
}
```

Voyons en détail cette ligne de code : `let x: String = String::from("Hello, world!");`

- `let x` : déclare une variable `x` de type `String`.:

- `= String::from("Hello, world!")` : initialise `x` avec la valeur `"Hello, world!"`.

`String::from()` est une fonction associée au type String en Rust, utilisée pour créer une nouvelle instance de String à partir d'une chaîne de caractères littérale. En Rust, les chaînes de caractères littérales, spécifiées avec des guillemets doubles (comme `"hello"`), sont de type `&str`. Le type &str représente une vue immuable sur une chaîne de caractères stockée quelque part en mémoire (généralement dans le segment de données du programme, donc immuable et de durée de vie statique). En revanche, le type String est une chaîne de caractères allouée sur le tas, modifiable et de taille dynamique.

La nécessité de `String::from()` découle de ces différences. Lorsque vous avez besoin d'une chaîne de caractères que vous pouvez modifier ou dont la taille peut changer, vous utilisez un String plutôt qu'un `&str`.

Exemple, pour convertir un `&str` en `String`, vous pouvez utiliser `String::from()` comme suit :

```
fn main() {
    let x: &str = "Hello, world!"; // Une chaîne de caractères littérale
    let y: String = String::from(x); // Convertit x en une chaîne de
caractères dynamique
    println!("La valeur de y est : {}", y); // La valeur de y est : Hello,
world!
}
```

Dans cet exemple : `let y: String = String::from(x);` convertit la chaîne de caractères littérale `x` en une chaîne de caractères dynamique `y`.

- Pointeurs :
    - Les pointeurs sont utilisés pour stocker l'adresse mémoire d'une valeur. En Rust, les pointeurs sont de deux types : les références (`&`) et les pointeurs intelligents (`Box`, `Rc`, `Arc`, etc.).
    - Les références (`&`) sont des pointeurs qui permettent d'accéder à une valeur sans la posséder. Les références sont utilisées pour éviter la copie de valeurs et pour garantir la sécurité de la mémoire.
    - Les pointeurs intelligents (`Box`, `Rc`, `Arc`, etc.) sont des pointeurs qui permettent de posséder une valeur et de la partager entre plusieurs propriétaires. Les pointeurs intelligents sont utilisés pour gérer la mémoire de manière dynamique et pour garantir la sécurité de la mémoire.

Les pointeurs : les références (`&`)

Les références (`&`) sont des pointeurs qui permettent d'accéder à une valeur sans la posséder. Les références sont utilisées pour éviter la copie de valeurs et pour garantir la sécurité de la mémoire.

Exemple :

```
fn main() {
    let x: i32 = 5;
    let y: &i32 = &x; // Une référence à x
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
```

```
    println!("La valeur de y est : {}", y); // La valeur de y est : 5
}
```

Dans cet exemple, `let y: &i32 = &x;` crée une référence `y` à la valeur `x`. La référence `y` permet d'accéder à la valeur `x` sans la posséder. Cela signifie que `y` ne possède pas la valeur `x`, mais qu'il peut y accéder.

Les pointeurs intelligents (`Box`, `Rc`, `Arc`, `RefCell`, `Mutex`, `Cow`)

En Rust, les pointeurs intelligents sont des types de données qui non seulement agissent comme des pointeurs, mais offrent également des fonctionnalités supplémentaires, telles que la gestion de la mémoire ou la sécurité à l'exécution. Voici une liste des pointeurs intelligents les plus couramment utilisés en Rust :

1- **Box** : Le type Box permet d'allouer des données sur le tas plutôt que sur la pile. Un Box pointe vers des données allouées dans le tas et détruit ces données lorsque le Box sort de la portée, ce qui permet de gérer la mémoire automatiquement et en toute sécurité. Box est souvent utilisé pour :

`Box<T>` : `T` est un type de données quelconque. Box est utilisé pour :

- Créer des types récursifs.
- Posséder des données de taille inconnue à la compilation.
- Transférer la propriété des données sans faire une copie.

```
fn main() {
    let x: Box<i32> = Box::new(5); // Un pointeur intelligent Box à un
entier de 32 bits
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

Détaillons ce code : `let x: Box<i32> = Box::new(5);`

- `let x` : déclare une variable `x` de type `Box<i32>` avec une taille de 32 bits.
- `= Box::new(5)` : initialise `x` avec la valeur `5`.
- `Box::new()` : crée une nouvelle instance de Box à partir de la valeur `5`.

2- **Rc** : Le type `Rc` fournit un comptage de références immuables, permettant à plusieurs parties du code de "posséder" des données. Lorsque le dernier Rc pointant vers des données est détruit, les données sont également détruites. Rc est utilisé pour :

`Rc<T>` veut dire "Reference Count" (comptage de références). Rc est utilisé pour :

- Partager des données immuables entre plusieurs parties d'un programme.
- Utilisé uniquement dans des scénarios à thread unique.

```
fn main() {
    let x: Rc<i32> = Rc::new(5); // Un pointeur intelligent Rc à un entier
de 32 bits
```

```
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

Détaillons ce code : `let x: Rc<i32> = Rc::new(5);`

- `let x` : déclare une variable `x` de type `Rc<i32>` avec une taille de 32 bits.
- `= Rc::new(5)` : initialise `x` avec la valeur `5`.
- `Rc::new()` : crée une nouvelle instance de Rc à partir de la valeur `5`.

3- **Arc** : Similaire à Rc, mais Arc est sûr à utiliser dans des contextes multithread grâce à un comptage de références atomique (d'où le nom "Atomic Reference Count"). Arc est utilisé pour :

`Arc<T>` veut dire "Atomic Reference Count" (comptage de références atomique). Arc est utilisé pour :

- Partager des données immuables entre plusieurs `threads` (fils d'exécution).
- Assurer que les données restent en vie tant qu'au moins un pointeur y accède, dans un environnement multithread.

```
fn main() {
    let x: Arc<i32> = Arc::new(5); // Un pointeur intelligent Arc à un
entier de 32 bits
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

Détaillons ce code : `let x: Arc<i32> = Arc::new(5);`

- `let x` : déclare une variable `x` de type `Arc<i32>` avec une taille de 32 bits.
- `= Arc::new(5)` : initialise `x` avec la valeur `5`.
- `Arc::new()` : crée une nouvelle instance de Arc à partir de la valeur `5`.

4- **RefCell** : Fournit une mutabilité intérieure, permettant de modifier les données même lorsque l'instance RefCell est immuable. RefCell suit les emprunts à l'exécution, paniquant si les règles d'emprunt sont violées. `RefCell` est souvent utilisé pour :

- Contourner les règles d'emprunt statiques de Rust lorsque vous savez que le code respectera les règles d'emprunt à l'exécution.
- Modifier des données partagées entre différentes parties d'un programme sans recourir à la mutabilité globale.

```
fn main() {
    let x: RefCell<i32> = RefCell::new(5); // Un pointeur intelligent
RefCell à un entier de 32 bits
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

Détaillons ce code : `let x: RefCell<i32> = RefCell::new(5);`

- `let x` : déclare une variable `x` de type `RefCell<i32>` avec une taille de 32 bits.
- `= RefCell::new(5)` : initialise `x` avec la valeur `5`.
- `x.borrow_mut()` : permet de modifier la valeur de `x` en utilisant la méthode `borrow_mut()`.

5- **Mutex** : Un type de synchronisation qui permet de protéger les données avec un accès exclusif dans un contexte multithread. Mutex est similaire à Arc, mais il ajoute un verrouillage pour s'assurer qu'un seul thread peut accéder aux données à la fois. Mutex est utilisé pour :

`Mutex<T>` veut dire "Mutual Exclusion" (exclusion mutuelle). Mutex est utilisé pour :

- Protéger des données partagées entre threads.
- Assurer qu'un seul thread à la fois peut modifier les données partagées.

```
fn main() {
    let x: Mutex<i32> = Mutex::new(5); // Un pointeur intelligent Mutex à
un entier de 32 bits
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

Détaillons ce code : `let x: Mutex<i32> = Mutex::new(5);`

- `let x` : déclare une variable `x` de type `Mutex<i32>` avec une taille de 32 bits.
- `= Mutex::new(5)` : initialise `x` avec la valeur `5`.
- `Mutex::new()` : crée une nouvelle instance de Mutex à partir de la valeur `5`.

6- **Cow** : Clone On Write (COW) est un type d'énumération qui peut encapsuler soit une référence immuable vers des données (&T), soit une version possédée (T) des données. Cow permet une flexibilité entre performance et propriété des données, clonant les données uniquement si une modification est nécessaire. Cow est utilisé pour :

- Optimiser les cas où des données pourraient être modifiées, mais sont souvent lues sans modification.
- Réduire les coûts de clonage en ne clonant que lorsque c'est absolument nécessaire.

```
fn main() {
    let x: Cow<i32> = Cow::Borrowed(&5); // Un pointeur intelligent Cow à
un entier de 32 bits
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

Détaillons ce code : `let x: Cow<i32> = Cow::Borrowed(&5);`

- `let x` : déclare une variable `x` de type `Cow<i32>` avec une taille de 32 bits.
- `= Cow::Borrowed(&5)` : initialise `x` avec la valeur `5`.
- `Cow::Borrowed()` : crée une nouvelle instance de Cow à partir de la valeur `5`.

Détaillons ces types de pointeurs intelligents :

- **Box** : Le type Box permet d'allouer des données sur le tas plutôt que sur la pile. Un Box pointe vers des données allouées dans le tas et détruit ces données lorsque le Box sort de la portée, ce qui permet de gérer la mémoire automatiquement et en toute sécurité. Box est souvent utilisé pour :
    - Créer des types récursifs.
    - Posséder des données de taille inconnue à la compilation.
    - Transférer la propriété des données sans faire une copie.
- **Rc** : Le type `Rc` fournit un comptage de références immuables, permettant à plusieurs parties du code de "posséder" des données. Lorsque le dernier Rc pointant vers des données est détruit, les données sont également détruites. Rc est utilisé pour :
    - Partager des données immuables entre plusieurs parties d'un programme.
    - Utilisé uniquement dans des scénarios à thread unique.
- **Arc** : Similaire à Rc, mais Arc est sûr à utiliser dans des contextes multithread grâce à un comptage de références atomique (d'où le nom "Atomic Reference Count"). Arc est utilisé pour :
    - Partager des données immuables entre plusieurs `threads` (fils d'exécution).
    - Assurer que les données restent en vie tant qu'au moins un pointeur y accède, dans un environnement multithread.
- **RefCell** : Fournit une mutabilité intérieure, permettant de modifier les données même lorsque l'instance RefCell est immuable. RefCell suit les emprunts à l'exécution, paniquant si les règles d'emprunt sont violées. `RefCell` est souvent utilisé pour :
    - Contourner les règles d'emprunt statiques de Rust lorsque vous savez que le code respectera les règles d'emprunt à l'exécution.
    - Modifier des données partagées entre différentes parties d'un programme sans recourir à la mutabilité globale.
    - Assurer qu'un seul thread à la fois peut modifier les données partagées.
    - Optimiser les cas où des données pourraient être modifiées, mais sont souvent lues sans modification.
    - Réduire les coûts de clonage en ne clonant que lorsque c'est absolument nécessaire.
- **Cow** : Clone On Write (COW) est un type d'énumération qui peut encapsuler soit une référence immuable vers des données (&T), soit une version possédée (T) des données. Cow permet une flexibilité entre performance et propriété des données, clonant les données uniquement si une modification est nécessaire. Cow est utilisé pour :
    - Optimiser les cas où des données pourraient être modifiées, mais sont souvent lues sans modification.
    - Réduire les coûts de clonage en ne clonant que lorsque c'est absolument nécessaire.

Ces pointeurs intelligents jouent un rôle essentiel dans la gestion de la mémoire et la manipulation des données en Rust, offrant à la fois flexibilité et sécurité.

Fonctions

Les fonctions sont des blocs de code qui effectuent une tâche spécifique. En Rust, les fonctions sont déclarées en utilisant le mot-clé `fn`, suivi du nom de la fonction, de ses paramètres et de son type de retour. Voici un exemple de déclaration de fonction en Rust :

Syntaxe de base d'une fonction en Rust :

```
fn main() {
    fn nom_de_la_fonction(paramètre1: Type, paramètre2: Type) ->
Type_de_retour {
        // Corps de la fonction
    }
}
```

Exemple :

```
fn main() {
    fn add(x: i32, y: i32) -> i32 {
        x + y
    }
    let result = add(5, 10);
    println!("Le résultat est : {}", result); // Le résultat est : 15
}
```

Dans cet exemple, `fn add(x: i32, y: i32) -> i32 { x + y }` déclare une fonction `add` qui prend deux paramètres `x` et `y` de type `i32` et renvoie un résultat de type `i32`. Le corps de la fonction `add` est `x + y`, qui ajoute les deux paramètres `x` et `y` et renvoie le résultat.

Pour appeler une fonction en Rust, vous pouvez utiliser le nom de la fonction suivi de ses paramètres entre parenthèses. Par exemple, `add(5, 10)` appelle la fonction `add` avec les paramètres `5` et `10` et stocke le résultat dans la variable `result`.

Commentaires

Les commentaires sont des annotations dans le code source qui ne sont pas exécutées par le programme, mais qui fournissent des informations supplémentaires sur le code. En Rust, les commentaires peuvent être de deux types : les commentaires de ligne (`//`) et les commentaires de bloc (`/* ... */`). Voici un exemple de commentaires en Rust :

Exemple :

```
fn main() {
    // Ceci est un commentaire de ligne
    let x = 5; // Ceci est un commentaire de fin de ligne
    /*
    Ceci est un commentaire de bloc
    qui peut s'étendre sur plusieurs lignes
```

```
    */
}
```

Dans cet exemple, `// Ceci est un commentaire de ligne` est un commentaire de ligne, `let x = 5; // Ceci est un commentaire de fin de ligne` est un commentaire de fin de ligne et `/* ... */` est un commentaire de bloc.

Exemples de programmes Rust simples

Voici quelques exemples de programmes Rust simples pour illustrer les concepts de base de Rust :

- Programme Rust simple pour afficher "Hello, world!" à l'écran :

```rust
fn main() {
    println!("Hello, world!");
}
```

- Programme Rust simple pour ajouter deux nombres et afficher le résultat :

```rust
fn main() {
    fn add(x: i32, y: i32) -> i32 {
        x + y
    }
    let result = add(5, 10);
    println!("Le résultat est : {}", result); // Le résultat est : 15
}
```

- Programme Rust simple pour déclarer une variable et afficher sa valeur :

```rust
fn main() {
    let x = 5;
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
}
```

- Programme Rust simple pour déclarer un tableau et afficher ses valeurs :

```rust
fn main() {
    let x: [i32; 5] = [1, 2, 3, 4, 5]; // Un tableau de 5 entiers de 32
bits
    println!("La première valeur de x est : {}", x[0]); // La première
valeur de x est : 1
    println!("La deuxième valeur de x est : {}", x[1]); // La deuxième
valeur de x est : 2
    println!("La troisième valeur de x est : {}", x[2]); // La troisième
```

```
valeur de x est : 3
}
```

- Programme Rust simple pour déclarer une chaîne de caractères et afficher sa valeur :

```
fn main() {
    let x: &str = "Hello, world!"; // Une chaîne de caractères littérale
    println!("La valeur de x est : {}", x); // La valeur de x est : Hello,
world!
}
```

- Programme Rust simple pour déclarer une référence et afficher sa valeur :

```
fn main() {
    let x: i32 = 5;
    let y: &i32 = &x; // Une référence à x
    println!("La valeur de x est : {}", x); // La valeur de x est : 5
    println!("La valeur de y est : {}", y); // La valeur de y est : 5
}
```

Contrôle de flux

Le contrôle de flux est une technique utilisée pour gérer l'exécution des instructions dans un programme. En Rust, le contrôle de flux est réalisé à l'aide de structures de contrôle telles que `if`, `else`, `loop`, `while`, `for`, `match`, `break`, `continue`, `return`, etc. Ces structures de contrôle permettent de prendre des décisions, de répéter des instructions, de gérer les erreurs et de contrôler le flux d'exécution du programme.

Structures de contrôle

Les structures de contrôle sont des instructions qui permettent de contrôler le flux d'exécution d'un programme. En Rust, les structures de contrôle les plus couramment utilisées sont `if`, `else`, `loop`, `while`, `for`, `match`, `break`, `continue`, `return`, etc. Ces structures de contrôle permettent de prendre des décisions, de répéter des instructions, de gérer les erreurs et de contrôler le flux d'exécution du programme.

`if`, `else` et `else if`

- `if` : L'instruction `if` est utilisée pour exécuter un bloc de code si une condition est vraie.

Syntaxe de base de l'instruction `if` en Rust :

```
fn main() {
    if condition {
        // Bloc de code à exécuter si la condition est vraie
    }
}
```

Exemple :

```
fn main() {
    let x = 5;
    if x > 0 {
        println!("x est positif");
    }
}
```

Dans cet exemple, `if x > 0 { println!("x est positif"); }` exécute le bloc de code `println!("x est positif");` si la condition `x > 0` est vraie.

- `else` : L'instruction `else` est utilisée pour exécuter un bloc de code si la condition de l'instruction `if` est fausse.

Syntaxe de base de l'instruction `else` en Rust :

```
fn main() {
    if condition {
        // Bloc de code à exécuter si la condition est vraie
    } else {
        // Bloc de code à exécuter si la condition est fausse
    }
}
```

Exemple :

```
fn main() {
    let x = 5;
    if x > 0 {
        println!("x est positif");
    } else {
        println!("x est négatif ou nul");
    }
}
```

Dans cet exemple, `if x > 0 { println!("x est positif"); } else { println!("x est négatif ou nul"); }` exécute le bloc de code `println!("x est positif");` si la condition `x > 0` est vraie, sinon il exécute le bloc de code `println!("x est négatif ou nul");`.

- `else if` : L'instruction `else if` est utilisée pour exécuter un bloc de code si une autre condition est vraie.

Syntaxe de base de l'instruction `else if` en Rust :

```
fn main() {
    if condition1 {
        // Bloc de code à exécuter si la condition1 est vraie
    } else if condition2 {
        // Bloc de code à exécuter si la condition2 est vraie
    } else {
        // Bloc de code à exécuter si aucune des conditions n'est vraie
    }
}
```

Exemple :

```
fn main() {
    let x = 5;
    if x > 0 {
        println!("x est positif");
    } else if x < 0 {
        println!("x est négatif");
    } else {
        println!("x est nul");
    }
}
```

Dans cet exemple, `if x > 0 { println!("x est positif"); } else if x < 0 { println!("x est négatif"); } else { println!("x est nul"); }` exécute le bloc de code `println!("x est positif");` si la condition `x > 0` est vraie, sinon il exécute le bloc de code `println!("x est négatif");` si la condition `x < 0` est vraie, sinon il exécute le bloc de code `println!("x est nul");`.

`loop`

L'instruction `loop` est utilisée pour exécuter un bloc de code en boucle indéfiniment jusqu'à ce qu'une condition de sortie soit rencontrée.

Syntaxe de base de l'instruction `loop` en Rust :

```
fn main() {
    loop {
        // Bloc de code à exécuter en boucle
    }
}
```

Exemple :

```
fn main() {
    let mut x = 0;
    loop {
        println!("x = {}", x);
        x += 1;
        if x == 5 {
            break;
        }
    }
}
```

Dans cet exemple, `loop { println!("x = {}", x); x += 1; if x == 5 { break; } }` exécute le bloc de code `println!("x = {}", x); x += 1;` en boucle jusqu'à ce que la condition `x == 5` soit vraie, puis il sort de la boucle à l'aide de l'instruction `break`.

- let mut x = 0; : déclare une variable `x` et l'initialise avec la valeur `0`.
- loop { ... } : exécute le bloc de code en boucle indéfiniment.
- println!("x = {}", x); : affiche la valeur de `x`.
- x += 1; : incrémente la valeur de `x` de `1`.
- if x == 5 { break; } : vérifie si la valeur de `x` est égale à `5`, puis sort de la boucle à l'aide de l'instruction `break`.
- break; : sort de la boucle.

`while`

L'instruction `while` est utilisée pour exécuter un bloc de code en boucle tant qu'une condition est vraie.

Syntaxe de base de l'instruction `while` en Rust :

```
fn main() {
    while condition {
        // Bloc de code à exécuter en boucle tant que la condition est
vraie
    }
}
```

Exemple :

```
fn main() {
    let mut x = 0;
    while x < 5 {
        println!("x = {}", x);
        x += 1;
    }
}
```

Dans cet exemple, `while x < 5 { println!("x = {}", x); x += 1; }` exécute le bloc de code `println!("x = {}", x); x += 1;` en boucle tant que la condition `x < 5` est vraie.

- let mut x = 0; : déclare une variable `x` et l'initialise avec la valeur `0`.
- while x < 5 { ... } : exécute le bloc de code en boucle tant que la condition `x < 5` est vraie.
- println!("x = {}", x); : affiche la valeur de `x`.
- x += 1; : incrémente la valeur de `x` de `1`.

`for`

L'instruction `for` est utilisée pour exécuter un bloc de code pour chaque élément d'une collection (par exemple, un tableau, une plage, etc.).

Syntaxe de base de l'instruction `for` en Rust :

```
fn main() {
    for element in collection {
        // Bloc de code à exécuter pour chaque élément de la collection
    }
}
```

Exemple :

```
fn main() {
    let numbers = [1, 2, 3, 4, 5];
    for number in numbers.iter() {
        println!("number = {}", number);
    }
}
```

Dans cet exemple, `for number in numbers.iter() { println!("number = {}", number); }` exécute le bloc de code `println!("number = {}", number);` pour chaque élément de la collection `numbers`.

- let numbers = [1, 2, 3, 4, 5]; : déclare un tableau `numbers` avec les valeurs `1`, `2`, `3`, `4` et `5`.
- for number in numbers.iter() { ... } : exécute le bloc de code pour chaque élément de la collection `numbers`.
- println!("number = {}", number); : affiche la valeur de `number`.
- numbers.iter() : itère sur les éléments de la collection `numbers`.
- number : représente chaque élément de la collection `numbers`.

`match`

L'instruction `match` est utilisée pour effectuer des correspondances de motifs sur une valeur et exécuter un bloc de code en fonction du motif correspondant.

Syntaxe de base de l'instruction `match` en Rust :

```
fn main() {
    match value {
        pattern1 => {
            // Bloc de code à exécuter si value correspond à pattern1
        }
        pattern2 => {
            // Bloc de code à exécuter si value correspond à pattern2
        }
        _ => {
            // Bloc de code à exécuter si aucune des conditions n'est vraie
        }
    }
}
```

Exemple 1 de code Rust :

```
fn main() {
    let x = 5;
    match x {
        1 => println!("x est égal à 1"),
        2 => println!("x est égal à 2"),
        _ => println!("x n'est ni égal à 1 ni à 2"),
    }
}
```

Dans cet exemple, `match x { 1 => println!("x est égal à 1"), 2 => println!("x est égal à 2"), _ => println!("x n'est ni égal à 1 ni à 2"), }` effectue des correspondances de motifs sur la valeur `x` et exécute le bloc de code correspondant en fonction du motif correspondant.

- let x = 5; : déclare une variable `x` et l'initialise avec la valeur `5`.
- match x { ... } : effectue des correspondances de motifs sur la valeur `x`.
- 1 => println!("x est égal à 1") : exécute le bloc de code si `x` est égal à `1`.
- 2 => println!("x est égal à 2") : exécute le bloc de code si `x` est égal à `2`.
- _ => println!("x n'est ni égal à 1 ni à 2") : exécute le bloc de code par défaut si aucune des conditions n'est vraie.

Exemple 2 de code Rust :

```
fn main() {
    let x = 5;
    match x {
        1 | 2 => println!("x est égal à 1 ou 2"), // Correspondance
multiple
        3..=5 => println!("x est compris entre 3 et 5"), // Correspondance
```

```
de plage
        _ => println!("x n'est ni égal à 1 ni à 2, ni compris entre 3 et
5"), // Correspondance par défaut
    }
}
```

Dans cet exemple, `match x { 1 | 2 => println!("x est égal à 1 ou 2"), 3..=5 => println!("x est compris entre 3 et 5"), _ => println!("x n'est ni égal à 1 ni à 2, ni compris entre 3 et 5"), }` effectue des correspondances de motifs plus complexes sur la valeur `x` et exécute le bloc de code correspondant en fonction du motif correspondant.

- let x = 5; : déclare une variable `x` et l'initialise avec la valeur `5`.
- match x { ... } : effectue des correspondances de motifs sur la valeur `x`.
- 1 | 2 => println!("x est égal à 1 ou 2") : exécute le bloc de code si `x` est égal à `1` ou `2`.
- 3..=5 => println!("x est compris entre 3 et 5") : exécute le bloc de code si `x` est compris entre `3` et `5`.
- _ => println!("x n'est ni égal à 1 ni à 2, ni compris entre 3 et 5") : exécute le bloc de code par défaut si aucune des conditions n'est vraie.

`break`

L'instruction `break` est utilisée pour sortir d'une boucle.

Syntaxe de base de l'instruction `break` en Rust :

```
fn main() {
    loop {
        // ...
        if condition {
            break;
        }
        // ...
    }
}
```

Exemple :

```
fn main() {
    let mut x = 0;
    loop {
        println!("x = {}", x);
        x += 1;
        if x == 5 {
            break;
        }
    }
}
```

Dans cet exemple, `loop { println!("x = {}", x); x += 1; if x == 5 { break; } }` exécute le bloc de code `println!("x = {}", x); x += 1;` en boucle jusqu'à ce que la condition `x == 5` soit vraie, puis il sort de la boucle à l'aide de l'instruction `break`.

- let mut x = 0; : déclare une variable `x` et l'initialise avec la valeur `0`.
- loop { ... } : exécute le bloc de code en boucle indéfiniment.
- println!("x = {}", x); : affiche la valeur de `x`.
- x += 1; : incrémente la valeur de `x` de `1`.
- if x == 5 { break; } : vérifie si la valeur de `x` est égale à `5`, puis sort de la boucle à l'aide de l'instruction `break`.
- break; : sort de la boucle.

`continue`

L'instruction `continue` est utilisée pour passer à l'itération suivante d'une boucle.

Syntaxe de base de l'instruction `continue` en Rust :

```
fn main() {
    loop {
        // ...
        if condition {
            continue;
        }
        // ...
    }
}
```

Exemple :

```
fn main() {
    for x in 0..5 {
        if x % 2 == 0 {
            continue;
        }
        println!("x = {}", x); // Affichera les valeurs impaires de x : 1,
2, 3, 4
    }
}
```

Dans cet exemple, `for x in 0..5 { if x % 2 == 0 { continue; } println!("x = {}", x); }` exécute le bloc de code `println!("x = {}", x);` pour chaque élément de la plage `0..5`, mais il passe à l'itération suivante si la condition `x % 2 == 0` est vraie.

- for x in 0..5 { ... } : exécute le bloc de code pour chaque élément de la plage `0..5`.
- if x % 2 == 0 { continue; } : passe à l'itération suivante si la condition `x % 2 == 0` est vraie.
- println!("x = {}", x); : affiche la valeur de `x`.

- x % 2 : calcule le reste de la division de `x` par `2`.
- continue; : passe à l'itération suivante.
- 0..5 : représente une plage de `0` à `5` (excluant `5`).
- x : représente chaque élément de la plage `0..5`.
- 0, 1, 2, 3, 4 : sont les éléments de la plage `0..5`.

`return`

L'instruction `return` est utilisée pour renvoyer une valeur à partir d'une fonction.

Syntaxe de base de l'instruction `return` en Rust :

```
fn nom_de_la_fonction(paramètre1: Type, paramètre2: Type) -> Type_de_retour
{
    // ...
    if condition {
        return valeur;
    }
    // ...
}
```

Exemple :

```
fn add(x: i32, y: i32) -> i32 {
    return x + y;
}
fn main() {
    let result = add(5, 10);
    println!("Le résultat est : {}", result); // Le résultat est : 15
}
```

Dans cet exemple, `return x + y;` renvoie la valeur de `x + y` à partir de la fonction `add`.

- fn add(x: i32, y: i32) -> i32 { ... } : déclare une fonction `add` qui prend deux paramètres `x` et `y` de type `i32` et renvoie un résultat de type `i32`.
- return x + y; : renvoie la valeur de `x + y` à partir de la fonction `add`.
- let result = add(5, 10); : appelle la fonction `add` avec les paramètres `5` et `10` et stocke le résultat dans la variable `result`.
- println!("Le résultat est : {}", result); : affiche le résultat.
- 5, 10 : sont les paramètres de la fonction `add`.
- 15 : est le résultat de l'addition de `5` et `10`.
- result : est la variable qui stocke le résultat.
- Le résultat est : 15 : est le résultat affiché.

Concepts avancés

Les concepts avancés en Rust comprennent les types de données avancés, les traits, les génériques, les macros, la gestion des erreurs, etc. Ces concepts permettent de créer des programmes plus complexes et plus flexibles en Rust.

Pattern Matching et Contrôle de Flux

Le pattern matching est une technique utilisée pour effectuer des correspondances de motifs sur une valeur et exécuter un bloc de code en fonction du motif correspondant. En Rust, le pattern matching est réalisé à l'aide de l'instruction `match`, qui permet de comparer une valeur à une série de motifs et d'exécuter un bloc de code en fonction du motif correspondant.

`match`

L'instruction `match` est utilisée pour effectuer des correspondances de motifs sur une valeur et exécuter un bloc de code en fonction du motif correspondant.

Syntaxe de base de l'instruction `match` en Rust :

```
fn main() {
    match value {
        pattern1 => {
            // Bloc de code à exécuter si value correspond à pattern1
        }
        pattern2 => {
            // Bloc de code à exécuter si value correspond à pattern2
        }
        _ => {
            // Bloc de code à exécuter si aucune des conditions n'est vraie
        }
    }
}
```

Exemple 1 de code Rust :

```
fn main() {
    let x = 5;
    match x {
        1 => println!("x est égal à 1"),
        2 => println!("x est égal à 2"),
        _ => println!("x n'est ni égal à 1 ni à 2"),
    }
}
```

Dans cet exemple, `match x { 1 => println!("x est égal à 1"), 2 => println!("x est égal à 2"), _ => println!("x n'est ni égal à 1 ni à 2"), }` effectue des correspondances de motifs sur la valeur `x` et exécute le bloc de code correspondant en fonction du motif correspondant.

- let x = 5; : déclare une variable `x` et l'initialise avec la valeur `5`.
- match x { ... } : effectue des correspondances de motifs sur la valeur `x`.
- 1 => println!("x est égal à 1") : exécute le bloc de code si `x` est égal à `1`.
- 2 => println!("x est égal à 2") : exécute le bloc de code si `x` est égal à `2`.
- _ => println!("x n'est ni égal à 1 ni à 2") : exécute le bloc de code par défaut si aucune des conditions n'est vraie.

Exemple 2 de code Rust :

```
fn main() {
    let x = 5;
    match x {
        1 | 2 => println!("x est égal à 1 ou 2"), // Correspondance
multiple
        3..=5 => println!("x est compris entre 3 et 5"), // Correspondance
de plage
        _ => println!("x n'est ni égal à 1 ni à 2, ni compris entre 3 et
5"), // Correspondance par défaut
    }
}
```

Dans cet exemple, `match x { 1 | 2 => println!("x est égal à 1 ou 2"), 3..=5 => println!("x est compris entre 3 et 5"), _ => println!("x n'est ni égal à 1 ni à 2, ni compris entre 3 et 5"), }` effectue des correspondances de motifs plus complexes sur la valeur `x` et exécute le bloc de code correspondant en fonction du motif correspondant.

- let x = 5; : déclare une variable `x` et l'initialise avec la valeur `5`.
- match x { ... } : effectue des correspondances de motifs sur la valeur `x`.
- 1 | 2 => println!("x est égal à 1 ou 2") : exécute le bloc de code si `x` est égal à `1` ou `2`.
- 3..=5 => println!("x est compris entre 3 et 5") : exécute le bloc de code si `x` est compris entre `3` et `5`.
- _ => println!("x n'est ni égal à 1 ni à 2, ni compris entre 3 et 5") : exécute le bloc de code par défaut si aucune des conditions n'est vraie.

Comparaison avec `if let` et `while let`

`if let`

L'instruction `if let` est utilisée pour effectuer des correspondances de motifs sur une valeur et exécuter un bloc de code en fonction du motif correspondant.

Syntaxe de base de l'instruction `if let` en Rust :

```
fn main() {
    if let pattern = value {
        // Bloc de code à exécuter si value correspond à pattern
    }
}
```

Exemple :

```
fn main() {
    let x = Some(5); // Une option contenant la valeur 5
    if let Some(value) = x { // Correspondance de motif sur l'option
        println!("La valeur est : {}", value); // La valeur est : 5
    }
}
```

Dans cet exemple, `if let Some(value) = x { println!("La valeur est : {}", value); }` effectue des correspondances de motifs sur l'option `x` et exécute le bloc de code `println!("La valeur est : {}", value);` si l'option contient une valeur.

- let x = Some(5); : déclare une option `x` contenant la valeur `5`.
- if let Some(value) = x { ... } : effectue des correspondances de motifs sur l'option `x`.
- println!("La valeur est : {}", value); : affiche la valeur de `value`.
- Some(5) : représente une option contenant la valeur `5`.

`while let`

L'instruction `while let` est utilisée pour effectuer des correspondances de motifs sur une valeur et exécuter un bloc de code en boucle tant que le motif correspondant est vrai.

Syntaxe de base de l'instruction `while let` en Rust :

```
fn main() {
    while let pattern = value {
        // Bloc de code à exécuter en boucle tant que value correspond à
pattern
    }
}
```

Exemple :

```
fn main() {
    let mut x = Some(5); // Une option contenant la valeur 5
    while let Some(value) = x { // Correspondance de motif sur l'option
        println!("La valeur est : {}", value); // La valeur est : 5
        x = None; // Affecte à x la valeur None pour sortir de la boucle
    }
}
```

Dans cet exemple, `while let Some(value) = x { println!("La valeur est : {}", value); x = None; }` effectue des correspondances de motifs sur l'option `x` et exécute le bloc de code `println!("La valeur est : {}", value); x = None;` en boucle tant que l'option contient une valeur.

- let mut x = Some(5); : déclare une option `x` contenant la valeur `5`.
- while let Some(value) = x { ... } : effectue des correspondances de motifs sur l'option `x` en boucle tant que l'option contient une valeur.
- println!("La valeur est : {}", value); : affiche la valeur de `value`.
- x = None; : affecte à `x` la valeur `None` pour sortir de la boucle.
- Some(5) : représente une option contenant la valeur `5`.

`if let` et `while let` avec des énumérations

L'instruction `if let` et `while let` peut être utilisée avec des énumérations pour effectuer des correspondances de motifs sur les variantes de l'énumération et exécuter un bloc de code en fonction de la variante correspondante.

Exemple 1 de code Rust :

```
fn main() {
    enum Couleur {
        Rouge,
        Vert,
        Bleu,
    }
    let couleur = Couleur::Rouge;
    if let Couleur::Rouge = couleur {
        println!("La couleur est rouge");
    }
}
```

Dans cet exemple, `if let Couleur::Rouge = couleur { println!("La couleur est rouge"); }` effectue des correspondances de motifs sur la variante `Rouge` de l'énumération `Couleur` et exécute le bloc de code `println!("La couleur est rouge");` si la variante correspondante est trouvée.

Exemple 2 de code Rust :

```
fn main() {
    enum OptionEntier {
        Valeur(i32),
        Aucune,
    }
    let x = OptionEntier::Valeur(5);
    while let OptionEntier::Valeur(valeur) = x {
        println!("La valeur est : {}", valeur);
        x = OptionEntier::Aucune;
    }
}
```

Dans cet exemple, `while let OptionEntier::Valeur(valeur) = x { println!("La valeur est : {}", valeur); x = OptionEntier::Aucune; }` effectue des correspondances de motifs sur la variante `Valeur` de l'énumération `OptionEntier` et exécute le bloc de code `println!("La valeur est : {}", valeur); x = OptionEntier::Aucune;` en boucle tant que la variante correspondante est trouvée.

Patterns exhaustifs et le joker _

En Rust, les correspondances de motifs doivent être exhaustives, ce qui signifie que toutes les variantes d'une énumération doivent être prises en compte. Pour gérer les variantes non prises en compte, le joker _ peut être utilisé pour capturer toutes les autres variantes.

Patterns exhaustifs

Les patterns exhaustifs sont des correspondances de motifs qui prennent en compte toutes les variantes d'une énumération.

Exemple :

```
fn main() {
    enum Couleur {
        Rouge,
        Vert,
        Bleu,
    }
    let couleur = Couleur::Rouge;
    match couleur {
        Couleur::Rouge => println!("La couleur est rouge"),
        Couleur::Vert => println!("La couleur est verte"),
        Couleur::Bleu => println!("La couleur est bleue"),
    }
}
```

Dans cet exemple, `match couleur { Couleur::Rouge => println!("La couleur est rouge"), Couleur::Vert => println!("La couleur est verte"), Couleur::Bleu => println!("La couleur est bleue"), }` effectue des correspondances de motifs exhaustives sur la valeur `couleur` et exécute le bloc de code correspondant en fonction de la variante correspondante.

Le joker _

Le joker _ est utilisé pour capturer toutes les autres variantes non prises en compte dans les correspondances de motifs.

Exemple :

```
fn main() {
    enum Couleur {
        Rouge,
```

```
        Vert,
        Bleu,
    }
    let couleur = Couleur::Rouge;
    match couleur {
        Couleur::Rouge => println!("La couleur est rouge"),
        _ => println!("La couleur est inconnue"), // Capturer toutes les
autres variantes
    }
}
```

Dans cet exemple, `match couleur { Couleur::Rouge => println!("La couleur est rouge"), _ => println!("La couleur est inconnue"), }` effectue des correspondances de motifs sur la valeur `couleur` et exécute le bloc de code `println!("La couleur est inconnue");` pour toutes les autres variantes non prises en compte.

Matching sur les énumérations imbriquées

Les énumérations imbriquées sont des énumérations qui contiennent d'autres énumérations comme variantes. En Rust, les correspondances de motifs peuvent être utilisées pour effectuer des correspondances sur les variantes des énumérations imbriquées.

Exemple :

```
fn main() {
    enum Couleur {
        Rouge,
        Vert,
        Bleu,
    }
    enum Forme {
        Cercle(Couleur),
        Rectangle(Couleur),
    }
    let forme = Forme::Cercle(Couleur::Rouge);
    match forme {
        Forme::Cercle(Couleur::Rouge) => println!("La forme est un cercle
rouge"),
        Forme::Cercle(Couleur::Vert) => println!("La forme est un cercle
vert"),
        Forme::Rectangle(Couleur::Rouge) => println!("La forme est un
rectangle rouge"),
        Forme::Rectangle(Couleur::Vert) => println!("La forme est un
rectangle vert"),
        _ => println!("La forme est inconnue"),
    }
}
```

Dans cet exemple, `match forme { Forme::Cercle(Couleur::Rouge) => println!("La forme est un cercle rouge"), Forme::Cercle(Couleur::Vert) => println!("La forme est un cercle vert"), Forme::Rectangle(Couleur::Rouge) => println!("La forme est un rectangle rouge"), Forme::Rectangle(Couleur::Vert) => println!("La forme est un rectangle vert"), _ => println!("La forme est inconnue"), }` effectue des correspondances de motifs sur la valeur `forme` et exécute le bloc de code correspondant en fonction de la variante correspondante.

Matching sur les valeurs de référence

Les correspondances de motifs peuvent être utilisées pour effectuer des correspondances sur les valeurs de référence en Rust.

Exemple :

```
fn main() {
    let x = &5; // &5 est une référence à la valeur 5
    match x {
        &val => println!("La valeur est : {}", val), // &val correspond à la valeur de référence
    }
}
```

Dans cet exemple, `match x { &val => println!("La valeur est : {}", val), }` effectue des correspondances de motifs sur la valeur de référence `x` et exécute le bloc de code `println!("La valeur est : {}", val);` pour la valeur de référence `5`.

Matching sur les tuples

Les correspondances de motifs peuvent être utilisées pour effectuer des correspondances sur les tuples en Rust.

Exemple :

```
fn main() {
    let point = (5, 10);
    match point {
        (x, y) => println!("x = {}, y = {}", x, y), // (x, y) correspond aux éléments du tuple
    }
}
```

Dans cet exemple, `match point { (x, y) => println!("x = {}, y = {}", x, y), }` effectue des correspondances de motifs sur le tuple `point` et exécute le bloc de code `println!("x = {}, y = {}", x, y);` pour les éléments du tuple `5` et `10`.

Matching sur les pointeurs intelligents

Les correspondances de motifs peuvent être utilisées pour effectuer des correspondances sur les pointeurs intelligents en Rust.

Exemple :

```
fn main() {
    let x = Box::new(5); // Box::new(5) crée un pointeur intelligent
contenant la valeur 5
    match x {
        val => println!("La valeur est : {}", val), // val correspond au
pointeur intelligent
    }
}
```

Dans cet exemple, `match x { val => println!("La valeur est : {}", val), }` effectue des correspondances de motifs sur le pointeur intelligent `x` et exécute le bloc de code `println!("La valeur est : {}", val);` pour la valeur `5` du pointeur intelligent.

Matching sur les chaînes de caractères

Les correspondances de motifs peuvent être utilisées pour effectuer des correspondances sur les chaînes de caractères en Rust.

Exemple :

```
fn main() {
    let texte = "Hello, World!";
    match texte {
        "Hello, World!" => println!("Le texte est : Hello, World!"),
        _ => println!("Le texte est inconnu"),
    }
}
```

Dans cet exemple, `match texte { "Hello, World!" => println!("Le texte est : Hello, World!"), _ => println!("Le texte est inconnu"), }` effectue des correspondances de motifs sur la chaîne de caractères `texte` et exécute le bloc de code `println!("Le texte est : Hello, World!");` si la chaîne de caractères correspondante est trouvée.

Guards dans les patterns

Les guards dans les patterns sont des conditions supplémentaires qui peuvent être utilisées pour filtrer les correspondances de motifs en Rust.

Exemple :

```
fn main() {
    let x = 5;
```

```
    match x {
        val if val < 0 => println!("La valeur est négative"),
        val if val > 0 => println!("La valeur est positive"),
        _ => println!("La valeur est nulle"),
    }
}
```

Détaillons ce code :

- let x = 5; : déclare une variable `x` et l'initialise avec la valeur `5`.
- match x { ... } : effectue des correspondances de motifs sur la valeur `x`.
- val if val < 0 : est un garde qui filtre les correspondances de motifs si `val` est inférieur à `0`.
- val if val < 0 => println!("La valeur est négative") : exécute le bloc de code si `val` est inférieur à `0`.
- val if val > 0 => println!("La valeur est positive") : exécute le bloc de code si `val` est supérieur à `0`.
- _ => println!("La valeur est nulle") : exécute le bloc de code par défaut si aucune des conditions n'est vraie.

Génériques

En Rust, les types génériques permettent de définir des fonctions, des structures, des énumérations ou des méthodes qui peuvent opérer sur plusieurs types de données différents, plutôt que sur un seul. L'utilisation de types génériques améliore la capacité de réutilisation du code, réduit la duplication et augmente la clarté en traitant explicitement les intentions de types. Les génériques sont une fonctionnalité de programmation qui permet de définir des types de données et des fonctions qui peuvent être utilisés avec différents types de données sans duplication de code. En Rust, les génériques sont définis à l'aide du mot-clé `fn` suivi du nom de la fonction, des paramètres génériques entre crochets angulaires `<...>`, et des types de données génériques `T`, `U`, `V`, `W`.

Syntaxe de base des génériques en Rust :

```
fn nom_de_la_fonction<T, U, V>(paramètre1: T, paramètre2: U) -> V {
    // ...
}
```

Exemple :

```
fn maximum<T: PartialOrd>(liste: &[T]) -> &T { // T est un type générique
qui implémente PartialOrd. liste est une référence à un tableau de T. &T
est une référence à un T.
    let mut max = &liste[0]; // max est une référence à la première valeur
de liste.
    for element in liste {  // Pour chaque élément de liste...
        if element > max { // Si l'élément est supérieur à max...
            max = element; // Affecter l'élément à max.
        }
    }
    max // Retourner max.
```

```
}

// Appel de la fonction maximum avec un tableau de nombres.
fn main() {
    let nombres = vec![10, 5, 7, 3, 8];
    let max = maximum(&nombres); // Appel de la fonction maximum avec un
tableau de nombres.
    println!("Le maximum est : {}", max); // Le maximum est : 10
}
```

Voyons en détail ce code :

- fn maximum<T: PartialOrd>(liste: &[T]) -> &T { ... } : déclare une fonction `maximum` avec un type générique `T` qui implémente `PartialOrd`, un paramètre `liste` qui est une référence à un tableau de `T`, et un type de retour `&T` qui est une référence à un `T`.
- let mut max = &liste[0]; : déclare une variable `max` qui est une référence à la première valeur de `liste`.
- for element in liste { ... } : itère sur chaque élément de `liste`.
- if element > max { ... } : vérifie si l'élément est supérieur à `max`.
- max = element; : affecte l'élément à `max`.
- max : est la valeur maximale de `liste`.
- &nombres : est une référence à un tableau de nombres.
- Le maximum est : 10 : est le résultat affiché.

Les traits

Les traits sont une fonctionnalité de programmation qui permet de définir des comportements communs pour différents types de données en Rust. Les traits sont similaires aux interfaces dans d'autres langages de programmation et permettent de définir des méthodes et des comportements qui peuvent être implémentés par différents types de données. Les traits sont utilisés pour définir des comportements communs pour différents types de données en Rust. Ils permettent de définir des méthodes et des comportements qui peuvent être implémentés par différents types de données. Les traits sont similaires aux interfaces dans d'autres langages de programmation et sont utilisés pour définir des comportements communs pour différents types de données.

Syntaxe de base des traits en Rust :

```
trait NomDuTrait {
    // Définition des méthodes et des comportements du trait
}
```

Exemple :

```
trait Affichable {
    fn afficher(&self); // Définition d'une méthode afficher pour le trait
Affichable.
}
```

```rust
// Définition d'une structure Point avec les champs x et y.
struct Point {
    x: i32,
    y: i32,
}

// Implémentation du trait Affichable pour la structure Point.
impl Affichable for Point {
    fn afficher(&self) {
        println!("x = {}, y = {}", self.x, self.y);
    }
}

fn main() {
    let point = Point { x: 5, y: 10 }; // Création d'une instance de la
structure Point.
    point.afficher(); // Appel de la méthode afficher pour le point.
}
```

Voyons en détail ce code :

- trait Affichable { fn afficher(&self); } : définit un trait `Affichable` avec une méthode `afficher` qui prend une référence `&self`.
- struct Point { x: i32, y: i32, } : définit une structure `Point` avec les champs `x` et `y`.
- impl Affichable for Point { fn afficher(&self) { println!("x = {}, y = {}", self.x, self.y); } } : implémente le trait `Affichable` pour la structure `Point` en définissant la méthode `afficher`.
- let point = Point { x: 5, y: 10 }; : crée une instance de la structure `Point`.
- point.afficher(); : appelle la méthode `afficher` pour le point.
- x = 5, y = 10 : est le résultat affiché.

Les macros

Les macros sont une fonctionnalité de programmation qui permet de définir des fragments de code réutilisables en Rust. Les macros sont utilisées pour générer du code à la compilation et permettent d'écrire du code plus générique et plus flexible. Les macros sont définies à l'aide du mot-clé `macro_rules!` suivi du nom de la macro et de ses règles de correspondance.

Syntaxe de base des macros en Rust :

```rust
macro_rules! nom_de_la_macro {
    // Définition des règles de correspondance de la macro
}
```

Exemple :

```rust
macro_rules! dire_bonjour {
    () => {
```

```
            println!("Bonjour, monde!");
        };
    }

    fn main() {
        dire_bonjour!(); // Appel de la macro dire_bonjour.
    }
```

Voyons en détail ce code :

- macro_rules! dire_bonjour { () => { println!("Bonjour, monde!"); }; } : définit une macro `dire_bonjour` qui génère le code `println!("Bonjour, monde!");`.
- dire_bonjour!(); : appelle la macro `dire_bonjour` pour générer le code `println!("Bonjour, monde!");`.
- le `!` après le nom de la macro indique qu'il s'agit d'une macro et non d'une fonction.

La différence entre une macro et une fonction est que la macro est évaluée à la compilation, tandis que la fonction est évaluée à l'exécution. Les macros sont utilisées pour générer du code à la compilation, ce qui permet d'écrire du code plus générique et plus flexible.

Ownership (Propriété)

L'ownership (propriété) est un concept clé en Rust qui permet de gérer la mémoire de manière sûre et efficace en suivant des règles strictes de propriété, d'emprunt et de retour. L'ownership est un concept clé en Rust qui permet de gérer la mémoire de manière sûre et efficace en suivant des règles strictes de propriété, d'emprunt et de retour. L'ownership garantit que les ressources mémoire sont libérées de manière sûre et évite les problèmes de fuites de mémoire et de corruption de mémoire.

Règles de l'ownership

Les règles de l'ownership en Rust sont les suivantes :

- Chaque valeur en Rust a une variable qui est sa propriétaire.
- Une valeur ne peut avoir qu'un seul propriétaire à la fois.
- Lorsqu'un propriétaire sort du champ d'application, la valeur est libérée.
- La propriété peut être transférée à une autre variable ou empruntée temporairement.
- Les emprunts sont des références à une valeur qui n'ont pas la propriété de la valeur.
- Les emprunts peuvent être mutables ou immuables.
- Les emprunts sont vérifiés à la compilation pour éviter les problèmes de sécurité et de concurrence.
- Les emprunts sont libérés lorsque la référence sort du champ d'application.
- Les emprunts peuvent être partagés entre plusieurs propriétaires.

Emprunts et références

Les emprunts et les références sont des concepts clés en Rust qui permettent de partager des valeurs entre plusieurs propriétaires sans transférer la propriété de la valeur. Les emprunts et les références sont utilisés pour accéder aux valeurs sans posséder la valeur elle-même, ce qui évite les problèmes de transfert de propriété et de duplication de données.

- **Emprunt** : En Rust, l'emprunt est un concept central qui permet d'accéder à des données sans en prendre possession, assurant ainsi la sécurité de la mémoire et prévenant les erreurs courantes comme les violations d'accès concurrent ou les fuites de mémoire. Voici la syntaxe de base et quelques règles concernant les emprunts en Rust :

Emprunts Borrowing (Emprunt immuable) :

Règles des Emprunts Borrowing (Emprunt immuable) :

1. **Durée de vie des emprunts** : Un emprunt ne peut pas outrepasser la durée de vie de la ressource empruntée. Cela garantit qu'un emprunt ne pointe jamais vers une donnée qui a été libérée.

2. **Emprunts mutables exclusifs** : Vous ne pouvez pas mélanger des emprunts mutables et immuables d'une même ressource dans une même portée. Rust empêche cela pour éviter des modifications inattendues et des accès concurrents à la même donnée.

3. **Non-dangling pointers** : Les règles d'emprunt de Rust assurent que vous ne créez jamais de pointeurs suspendus (dangling pointers) — des pointeurs qui référencent une zone de mémoire qui a été libérée.

4. **Pattern Matching & Destructuring** : Vous pouvez utiliser les emprunts dans le pattern matching et le destructuring pour accéder à des parties d'une structure sans en prendre possession.

```
fn main() {
    let x = 5; // Déclaration d'une variable x avec la valeur 5.
    let emprunt = &x; // Création d'un emprunt immuable à la valeur de x.
    println!("La valeur de x est : {}", x); // Affichage de la valeur de x.
    println!("La valeur de l'emprunt est : {}", emprunt); // Affichage de
la valeur de l'emprunt.
}
```

Dans ce exemple, un emprunt `immuable` est créé à la valeur de `x` en utilisant l'opérateur `&`. Cela signifie que l'emprunt ne possède pas la valeur de x, mais peut y accéder de manière immuable. Cela garantit que la valeur de `x` ne peut pas être modifiée pendant la durée de vie de l'emprunt.

Emprunt mutables : Les emprunts mutables permettent de modifier la valeur de la variable à laquelle ils sont empruntés. En Rust, les emprunts mutables sont vérifiés à la compilation pour éviter les problèmes de sécurité et de concurrence.

```
fn main() {
    let mut x = 5; // Déclaration d'une variable x avec la valeur 5.
    let emprunt = &mut x; // Création d'un emprunt mutable à la valeur de
x.
    *emprunt += 1; // Modification de la valeur de x à travers l'emprunt.
    println!("La valeur de x est : {}", x); // Affichage de la valeur de x.
    println!("La valeur de l'emprunt est : {}", emprunt); // Affichage de
la valeur de l'emprunt.
}
```

Voyons en détail ce code :

- `let mut x = 5;` : déclare une variable `x` et l'initialise avec la valeur `5`.
- `let emprunt = &mut x;` : crée un emprunt mutable à la valeur de `x`.
- `*emprunt += 1;` : modifie la valeur de `x` à travers l'emprunt.
- `*` : est l'opérateur de déréférencement qui permet d'accéder à la valeur de l'emprunt.
- `println!("La valeur de x est : {}", x);` : affiche la valeur de `x`.

Les règles suivantes s'appliquent aux emprunts mutables :

1. Vous ne pouvez avoir qu'un seul emprunt mutable d'une ressource dans une portée donnée. Cela empêche les conditions de course.
2. Les emprunts mutables sont exclusifs : si vous avez un emprunt mutable, vous ne pouvez pas avoir d'emprunts immuables en même temps.

- **Références** : Les références sont des pointeurs intelligents qui permettent d'accéder à une valeur sans posséder la valeur elle-même. Les références sont utilisées pour partager des valeurs entre plusieurs propriétaires sans transférer la propriété de la valeur.

Syntaxe de base des références en Rust :

Références immuables : Pour créer une référence immuable à une valeur, utilisez le symbole &. Les références immuables permettent de lire une valeur sans la modifier :

```
fn main() {
    let x = 5; // Déclaration d'une variable x avec la valeur 5.
    let reference = &x; // Création d'une référence à la valeur de x.
    println!("La valeur de x est : {}", x); // Affichage de la valeur de x.
    println!("La valeur de la référence est : {}", reference); // Affichage
de la valeur de la référence.
}
```

Référence mutables :Pour créer une référence mutable à une valeur, utilisez &mut. Les références mutables permettent de modifier une valeur :

```
fn main() {
    let mut x = 5; // Déclaration d'une variable x avec la valeur 5.
    let reference = &mut x; // Création d'une référence mutable à la valeur
de x.
    *reference += 1; // Modification de la valeur de x à travers la
référence.
    println!("La valeur de x est : {}", x); // Affichage de la valeur de x.
    println!("La valeur de la référence est : {}", reference); // Affichage
de la valeur de la référence.
}
```

Règles importantes

1. Non-Dualité : Vous ne pouvez pas avoir une référence mutable lorsque vous avez une ou plusieurs références immuables. De même, si vous avez une référence mutable, vous ne pouvez pas créer de références immuables. Cette règle aide à prévenir les conditions de course.

2. Portée des Références : Une référence mutable ne peut exister que dans une portée où aucune autre référence (ni mutable ni immuable) à la même donnée n'existe.

3. Dangling References : Rust garantit qu'une référence ne peut jamais être une référence suspendue (dangling). Cela signifie que les données à laquelle une référence pointe ne seront jamais nettoyées tant que la référence existe.

4. Déréférencement : Utilisez l'opérateur de déréférencement * pour accéder ou modifier la valeur pointée par une référence.

Déréférencement et Modification : Lorsque vous avez une référence mutable à une valeur, vous pouvez modifier cette valeur en utilisant l'opérateur (*) de déréférencement. Exemple :

```
fn main() {
    let mut x = 5; // Déclaration d'une variable x avec la valeur 5.
    let reference = &mut x; // Création d'une référence mutable à la valeur
de x.
    *reference += 10; // Modification de la valeur de x à travers la
référence.
    println!("La valeur de x est : {}", x); // Affichage de la valeur de x.
    println!("La valeur de la référence est : {}", reference); // Affichage
de la valeur de la référence.
}
```

Les références, à la fois immuables et mutables, sont fondamentales pour travailler avec le système d'emprunt en Rust, offrant une manière sûre d'accéder et de modifier les données sans prendre possession complète de ces dernières.

Slices

Les slices sont des vues sur des parties d'un tableau en Rust. Ils permettent d'accéder à des parties d'un tableau sans posséder le tableau lui-même. Les slices sont utilisés pour partager des parties d'un tableau entre plusieurs propriétaires sans transférer la propriété du tableau.

Syntaxe de base des slices en Rust :

```
fn main() {
    let tableau = [1, 2, 3, 4, 5]; // Déclaration d'un tableau avec des
valeurs.
    let slice = &tableau[1..4]; // Création d'un slice à partir du tableau
qui contiendra les éléments 1, 2 et 3.

    for element in slice { // Pour chaque élément du slice...
        println!("Le slice est : {}", element); // Affichage de la valeur
de l'élément.
```

```
    }
}
```

Règles Importantes

1. **Sécurité** : Lorsque vous créez un slice, Rust vérifie les indices à la compilation si possible, ou à l'exécution, pour s'assurer qu'ils sont valides et ne débordent pas de la collection. Ceci aide à prévenir les erreurs de segmentation.
2. **Immutabilité** : Si la collection d'origine est immuable, le slice est également immuable. Si vous avez une référence mutable vers la collection d'origine et qu'aucun emprunt mutable ou immuable actif n'existe, vous pouvez créer un slice mutable.
3. **Durée de vie** : Les slices empruntent les données qu'ils référencent. Par conséquent, la durée de vie d'un slice ne peut pas dépasser celle de la collection à laquelle il se réfère.

Utilisations Courantes

1. **Accès en Lecture** : Les slices sont utilisés pour accéder en lecture à une portion d'une collection sans en prendre possession.
2. **Passage de Paramètres** : Ils permettent de passer une séquence d'éléments à une fonction sans lui passer la collection entière.
3. **Modèles de Conception** : Les slices facilitent certains modèles de conception, comme le partage de données entre différentes parties d'un programme sans copie ni duplication.

Les slices représentent un outil puissant dans Rust pour la manipulation efficace et sécurisée des séquences de données. Ils illustrent bien l'engagement de Rust envers la sécurité mémoire, la prévention des erreurs d'accès et la performance.

Les chaînes de caractères

Les chaînes de caractères sont des types de données qui représentent des séquences de caractères en Rust. Les chaînes de caractères sont utilisées pour stocker et manipuler du texte dans les programmes Rust. Les chaînes de caractères peuvent être créées à l'aide de littéraux de chaînes, de chaînes formatées, de chaînes mutables, etc.

- Exemple : Gestion d'une chaîne de caractères

```rust
fn main() {
    let mut chaine = String::from("Hello, "); // Création d'une chaîne mutable.
    chaine.push_str("World!"); // Ajout d'une chaîne à la chaîne existante.
    println!("{}", chaine); // Affichage de la chaîne.
}
```

Étudions en détail ce code :

- `let mut chaine = String::from("Hello, ");` : crée une chaîne de caractères mutable `chaine` avec la valeur `"Hello, "`.
- `chaine.push_str("World!");` : ajoute la chaîne `"World!"` à la chaîne existante `chaine`.
- push_str : est une méthode qui ajoute une chaîne à une chaîne existante.
- `println!("{}", chaine);` : affiche la chaîne `chaine`.

Les chaînes de caractères sont des types de données couramment utilisés en programmation pour stocker et manipuler du texte. En Rust, les chaînes de caractères sont gérées de manière sûre et efficace à l'aide de types de données spéciaux tels que `String` et `&str`.

Structures et Énumérations : Les types de données avancés

Les types de données avancés en Rust comprennent les énumérations, les structures, les tuples, les pointeurs intelligents, les chaînes de caractères, etc. Ces types de données permettent de représenter des données plus complexes et de créer des structures de données plus avancées en Rust.

Enum et pattern matching

Les énumérations sont des types de données qui permettent de définir un type avec un ensemble fini de valeurs possibles. En Rust, les énumérations sont déclarées à l'aide du mot-clé `enum` suivi du nom de l'énumération et de ses variantes. Voici un exemple de déclaration d'énumération en Rust :

Syntaxe de base de l'énumération en Rust :

```
fn main() {
    enum NomEnumeration {
        Variante1,
        Variante2,
        Variante3,
        // ...
    }

    let nom_variable = NomEnumeration::Variante1;

    match nom_variable {
        NomEnumeration::Variante1 => println!("La variante est Variante1"),
        NomEnumeration::Variante2 => println!("La variante est Variante2"),
        NomEnumeration::Variante3 => println!("La variante est Variante3"),
        _ => println!("La variante est inconnue"),
    }
}
```

Exemple :

```
fn main() {
    enum Couleur {
        Rouge,
        Vert,
        Bleu,
```

```
    }
    let couleur = Couleur::Rouge;

    match couleur {
        Couleur::Rouge => println!("La couleur est rouge"),
        Couleur::Vert => println!("La couleur est verte"),
        Couleur::Bleu => println!("La couleur est bleue"),
        _ => println!("La couleur est inconnue"),
    }
}
```

Dans cet exemple, `enum Couleur { Rouge, Vert, Bleu, }` déclare une énumération `Couleur` avec les variantes `Rouge`, `Vert` et `Bleu`. Ensuite, `let couleur = Couleur::Rouge;` crée une instance de l'énumération `Couleur` avec la variante `Rouge`.

Définir et utiliser une structure

Les structures sont des types de données qui permettent de regrouper plusieurs valeurs sous un seul nom. En Rust, les structures sont déclarées à l'aide du mot-clé `struct` suivi du nom de la structure et de ses champs. Voici un exemple de déclaration de structure en Rust :

Syntaxe de base de la structure en Rust :

```
fn main() {
    struct NomStructure {
        champ1: Type1,
        champ2: Type2,
        champ3: Type3,
        // ...
    }
}
```

Exemple :

```
fn main() {
    struct Point {
        x: i32,
        y: i32,
    }
    let point = Point { x: 5, y: 10 };

    println!("x = {}", point.x); // x = 5
}
```

Dans cet exemple, `struct Point { x: i32, y: i32, }` déclare une structure `Point` avec les champs `x` et `y` de type `i32`. Ensuite, `let point = Point { x: 5, y: 10 };` crée une instance de la structure `Point` avec les valeurs `5` et `10` pour les champs `x` et `y`.

Tuples

Les tuples sont des types de données qui permettent de regrouper plusieurs valeurs sous un seul nom sans avoir à définir une structure. En Rust, les tuples sont déclarés à l'aide de parenthèses et de virgules pour séparer les valeurs. Voici un exemple de déclaration de tuple en Rust :

Syntaxe de base du tuple en Rust :

```
fn main() {
    let nom_tuple = (valeur1, valeur2, valeur3);
}
```

Exemple :

```
fn main() {
    let point = (5, 10);

    let (x, y) = point; // Décomposition de tuple
    println!("x = {}", x); // x = 5
    println!("y = {}", y); // y = 10
}
```

Dans cet exemple, `let point = (5, 10);` crée un tuple `point` avec les valeurs `5` et `10`. Ensuite, `let (x, y) = point;` décompose le tuple `point` en deux variables `x` et `y` avec les valeurs `5` et `10`.

Les énumérations, les structures et les tuples sont des types de données avancés en Rust qui permettent de représenter des données plus complexes et de créer des structures de données plus avancées. Ces types de données sont largement utilisés en Rust pour représenter des données structurées et complexes.

Méthode et champ associé

Les méthodes et les champs associés sont des fonctionnalités avancées en Rust qui permettent de définir des méthodes et des champs qui sont associés à une énumération, une structure ou un trait. Les méthodes et les champs associés sont utilisés pour encapsuler la logique et les données associées à une énumération, une structure ou un trait.

Syntaxe de base des méthodes et des champs associés en Rust :

```
fn main() {
    impl NomStructure {
        fn nom_methode(&self) {
            // ...
        }
    }
}
```

Exemple :

```
fn main() {
    struct Point {
        x: i32,
        y: i32,
    }
    impl Point {
        fn new(x: i32, y: i32) -> Point {
            Point { x, y }
        }
        fn afficher(&self) {
            println!("x = {}, y = {}", self.x, self.y);
        }
    }
    let point = Point::new(5, 10);
    point.afficher();
}
```

Étudions en détail ce code :

- `struct Point { x: i32, y: i32, }` : déclare une structure `Point` avec les champs `x` et `y` de type `i32`.
- `impl Point { fn new(x: i32, y: i32) -> Point { Point { x, y } } fn afficher(&self) { println!("x = {}, y = {}", self.x, self.y); } }` : implémente la structure `Point` avec une méthode `new` pour créer une nouvelle instance de `Point` et une méthode `afficher` pour afficher les valeurs de `x` et `y`.
- `let point = Point::new(5, 10);` : crée une instance de la structure `Point` en utilisant la méthode `new`.
- `point.afficher();` : appelle la méthode `afficher` pour afficher les valeurs de `x` et `y`.
- `x = 5, y = 10` : est le résultat affiché.

Exemple : Gestion d'états avec enum

```
fn main() {
    enum Etat {
        Inactif,
        Actif,
        EnPanne,
    }
    impl Etat {
        fn afficher(&self) {
            match self {
                Etat::Inactif => println!("L'appareil est inactif"),
                Etat::Actif => println!("L'appareil est actif"),
                Etat::EnPanne => println!("L'appareil est en panne"),
            }
        }
    }
```

```
    let etat = Etat::Actif;
    etat.afficher();
}
```

Gestion des erreurs

La gestion des erreurs est une technique utilisée pour gérer les erreurs et les exceptions qui peuvent survenir pendant l'exécution d'un programme. En Rust, la gestion des erreurs est réalisée à l'aide de types de données spéciaux tels que `Result` et `Option`, ainsi que des instructions telles que `match`, `if let`, `unwrap`, `expect`, `?`, etc. Ces types de données et instructions permettent de gérer les erreurs de manière sûre et explicite, en évitant les exceptions non contrôlées.

Result et Option

Les types de données pour la gestion des erreurs sont des types spéciaux qui permettent de représenter les résultats réussis ou les erreurs qui peuvent survenir pendant l'exécution d'un programme. En Rust, les types de données les plus couramment utilisés pour la gestion des erreurs sont `Result` et `Option`.

`Result<T, E>`

Le type `Result` est utilisé pour représenter un résultat réussi (`Ok`) ou une erreur (`Err`). `<>` : Les crochets angulaires sont utilisés pour spécifier les types de données génériques `T` et `E`. `T` est le type de la valeur réussie et `E` est le type de l'erreur. `Result` est souvent utilisé pour les opérations qui peuvent échouer, telles que l'ouverture de fichiers, la lecture de données, etc.

Syntaxe de base du type `Result` en Rust :

```
fn main() {
    let result: Result<i32, &str> = Ok(5); // Un résultat réussi de type
i32
    let error: Result<i32, &str> = Err("Erreur"); // Une erreur de type
&str
}
```

Exemple :

```
fn divide(x: i32, y: i32) -> Result<i32, &str> {
    if y == 0 {
        Err("Division par zéro")
    } else {
        Ok(x / y)
    }
}
fn main() {
    let result = divide(10, 2);
    match result {
```

```
        Ok(value) => println!("Le résultat est : {}", value),
        Err(error) => println!("Erreur : {}", error),
    }
}
```

Dans cet exemple, `divide(10, 2)` renvoie un résultat réussi `Ok(5)` si la division est réussie, sinon il renvoie une erreur `Err("Division par zéro")`. Ensuite, `match result { Ok(value) => println!("Le résultat est : {}", value), Err(error) => println!("Erreur : {}", error), }` affiche le résultat réussi ou l'erreur en fonction du résultat de la division.

- fn divide(x: i32, y: i32) -> Result<i32, &str> { ... } : déclare une fonction `divide` qui prend deux paramètres `x` et `y` de type `i32` et renvoie un résultat réussi de type `i32` ou une erreur de type `&str`.
- if y == 0 { Err("Division par zéro") } else { Ok(x / y) } : renvoie une erreur si `y` est égal à `0`, sinon renvoie le résultat de la division.
- let result = divide(10, 2); : appelle la fonction `divide` avec les paramètres `10` et `2` et stocke le résultat dans la variable `result`.
- match result { Ok(value) => println!("Le résultat est : {}", value), Err(error) => println!("Erreur : {}", error), } : affiche le résultat réussi ou l'erreur en fonction du résultat de la division.

`Option<T>`

Le type `Option` est utilisé pour représenter une valeur réussie (`Some`) ou l'absence de valeur (`None`).

Syntaxe de base du type `Option` en Rust :

```
fn main() {
    let some_value: Option<i32> = Some(5); // Une valeur réussie de type
i32
    let none_value: Option<i32> = None; // L'absence de valeur
}
```

Exemple :

```
/**
 * Dans ce bloc de code get_first_element() est une fonction qui prend un
tableau de nombres et renvoie le premier élément du tableau s'il existe,
sinon elle renvoie l'absence de valeur.
 * get_first_element(&numbers) : appelle la fonction get_first_element()
avec un tableau de nombres et stocke le résultat dans la variable
first_element.
 * match first_element { Some(value) => println!("Le premier élément est :
{}", value), None => println!("Aucun élément trouvé"), } : affiche le
premier élément si une valeur est réussie, sinon affiche l'absence de
valeur.
 */
fn get_first_element(numbers: &[i32]) -> Option<i32> {
    if numbers.len() > 0 {
        Some(numbers[0]) // Une valeur réussie
```

```
    } else {
        None // L'absence de valeur
    }
}
fn main() {
    let numbers = [1, 2, 3, 4, 5];
    let first_element = get_first_element(&numbers);
    match first_element {
        Some(value) => println!("Le premier élément est : {}", value),
        None => println!("Aucun élément trouvé"),
    }
}
```

Étudions en détail ce code :

- `fn get_first_element(numbers: &[i32]) -> Option<i32> { ... }` : déclare une fonction `get_first_element` qui prend un tableau de nombres et renvoie une valeur réussie de type `i32` ou l'absence de valeur.
- `if numbers.len() > 0 { Some(numbers[0]) } else { None }` : renvoie une valeur réussie si le tableau de nombres n'est pas vide, sinon renvoie l'absence de valeur.
- `let first_element = get_first_element(&numbers);` : appelle la fonction `get_first_element` avec un tableau de nombres et stocke le résultat dans la variable `first_element`.
- `match first_element { Some(value) => println!("Le premier élément est : {}", value), None => println!("Aucun élément trouvé"), }` : affiche le premier élément si une valeur est réussie, sinon affiche l'absence de valeur.
- `Le premier élément est : 1` : est le résultat affiché.
- `Aucun élément trouvé` : est le résultat affiché.
- `Some(value)` : est utilisé pour représenter une valeur réussie.
- `None` : est utilisé pour représenter l'absence de valeur.

Les types de données `Result` et `Option` sont largement utilisés en Rust pour représenter les résultats réussis ou les erreurs, ainsi que les valeurs réussies ou l'absence de valeur. Ces types de données permettent de gérer les erreurs de manière sûre et explicite, en évitant les exceptions non contrôlées.

Propagation des erreurs

La propagation des erreurs est une technique utilisée pour propager les erreurs d'une fonction à une autre fonction qui l'appelle. En Rust, la propagation des erreurs est réalisée à l'aide de l'opérateur `?` qui permet de propager les erreurs d'une fonction à une autre fonction.

Syntaxe de base de la propagation des erreurs en Rust :

```
fn main() {
    fn fonction1() -> Result<i32, &str> {
        Ok(5)
    }
    fn fonction2() -> Result<i32, &str> {
        let resultat = fonction1()?;
```

```
        Ok(resultat)
    }
}
```

Exemple :

```
fn divide(x: i32, y: i32) -> Result<i32, &str> {
    if y == 0 {
        Err("Division par zéro")
    } else {
        Ok(x / y)
    }
}
fn multiply(x: i32, y: i32) -> Result<i32, &str> {
    let result = divide(x, y)?;
    Ok(result * 2)
}

fn main() {
    let result = multiply(10, 2);
    match result {
        Ok(value) => println!("Le résultat est : {}", value),
        Err(error) => println!("Erreur : {}", error),
    }
}
```

Dans cet exemple, `multiply(10, 2)` appelle la fonction `divide` pour diviser `10` par `2` et stocke le résultat dans la variable `result`. Ensuite, `match result { Ok(value) => println!("Le résultat est : {}", value), Err(error) => println!("Erreur : {}", error), }` affiche le résultat réussi ou l'erreur en fonction du résultat de la multiplication.

- `fn divide(x: i32, y: i32) -> Result<i32, &str> { ... }` : déclare une fonction `divide` qui prend deux paramètres `x` et `y` de type `i32` et renvoie un résultat réussi de type `i32` ou une erreur de type `&str`.
- `if y == 0 { Err("Division par zéro") } else { Ok(x / y) }` : renvoie une erreur si `y` est égal à `0`, sinon renvoie le résultat de la division.
- `fn multiply(x: i32, y: i32) -> Result<i32, &str> { let result = divide(x, y)?; Ok(result * 2) }` : appelle la fonction `divide` pour diviser `x` par `y` et stocke le résultat dans la variable `result`, puis multiplie le résultat par `2` et renvoie le résultat réussi ou l'erreur.
- `let result = multiply(10, 2);` : appelle la fonction `multiply` avec les paramètres `10` et `2` et stocke le résultat dans la variable `result`.
- `match result { Ok(value) => println!("Le résultat est : {}", value), Err(error) => println!("Erreur : {}", error), }` : affiche le résultat réussi ou l'erreur en fonction du résultat de la multiplication.

Exemple : Lecture d'un fichier et gestion des erreurs

```rust
use std::fs::File;
use std::io::Read;

fn lire_fichier(nom_fichier: &str) -> Result<String, std::io::Error> {
    let mut fichier = File::open(nom_fichier)?;
    let mut contenu = String::new();
    fichier.read_to_string(&mut contenu)?;
    Ok(contenu)
}

fn main() {
    let contenu = lire_fichier("fichier.txt");
    match contenu {
        Ok(valeur) => println!("Le contenu du fichier est : {}", valeur),
        Err(erreur) => println!("Erreur : {}", erreur),
    }
}
```

Que fait ce script :

- `use std::fs::File;` : importe le module `File` du module standard `fs` pour ouvrir et lire un fichier.
- `use std::io::Read;` : importe le trait `Read` du module standard `io` pour lire le contenu du fichier.
- `fn lire_fichier(nom_fichier: &str) -> Result<String, std::io::Error> { ... }` : déclare une fonction `lire_fichier` qui prend le nom du fichier à lire et renvoie le contenu du fichier ou une erreur.
- `let mut fichier = File::open(nom_fichier)?;` : ouvre le fichier spécifié et stocke le résultat dans la variable `fichier`, en propageant l'erreur si le fichier n'existe pas.
- `fichier.read_to_string(&mut contenu)?;` : lit le contenu du fichier et stocke le résultat dans la variable `contenu`, en propageant l'erreur si la lecture échoue.
- `Ok(contenu)` : renvoie le contenu du fichier ou une erreur.
- `let contenu = lire_fichier("fichier.txt");` : appelle la fonction `lire_fichier` avec le nom du fichier à lire et stocke le résultat dans la variable `contenu`.
- `match contenu { Ok(valeur) => println!("Le contenu du fichier est : {}", valeur), Err(erreur) => println!("Erreur : {}", erreur), }` : affiche le contenu du fichier si la lecture réussit, sinon affiche l'erreur.
- `Le contenu du fichier est : ...` : est le résultat affiché si la lecture réussit.
- `Erreur : ...` : est le résultat affiché si la lecture échoue.

Ce script lit le contenu d'un fichier spécifié et affiche le contenu du fichier si la lecture réussit, sinon affiche l'erreur.

Gestion des erreurs avec `?` et `unwrap`

L'opérateur `?`

est utilisé pour propager les erreurs d'une fonction à une autre fonction, tandis que la méthode `unwrap` est utilisée pour extraire la valeur réussie d'un résultat ou provoquer une panique si une erreur se produit.

Syntaxe de base de l'opérateur `?` et de la méthode `unwrap` en Rust :

```
fn main() {
    let result = fonction()?; // Propagation des erreurs avec l'opérateur ?
    let value = result.unwrap(); // Extraction de la valeur réussie avec la
méthode unwrap
}
```

`unwrap` : est une méthode qui extrait la valeur réussie d'un résultat ou provoque une panique si une erreur se produit

Exemple :

```
fn divide(x: i32, y: i32) -> Result<i32, &str> {
    if y == 0 {
        Err("Division par zéro")
    } else {
        Ok(x / y)
    }
}

fn main() {
    let result = divide(10, 2)?;
    let value = result.unwrap();
    println!("Le résultat est : {}", value);
}
```

Rust dans la Pratique

Collections

Les collections sont des types de données qui permettent de stocker et de manipuler des ensembles de valeurs dans un programme. En Rust, les collections les plus couramment utilisées sont les vecteurs, les chaînes de caractères et les hachages. Ces collections offrent des fonctionnalités avancées pour stocker, accéder, modifier et parcourir des ensembles de valeurs.

Vecteurs, String et HashMap

Les vecteurs, les chaînes de caractères et les hachages sont des types de données couramment utilisés en Rust pour stocker et manipuler des ensembles de valeurs.

Vecteurs

Les vecteurs en Rust sont des collections homogènes qui stockent leurs éléments dans un espace contigu de la mémoire. Ils peuvent croître ou décroître dynamiquement, ce qui les rend très flexibles pour stocker une collection d'éléments qui peut varier en taille. Un vecteur est représenté par le type `Vec<T>`, où `T` est

le type des éléments qu'il contient. Voici une introduction à l'utilisation et à la manipulation des vecteurs en Rust.

Création d'un Vecteur

Pour créer un nouveau vecteur vide, vous pouvez utiliser Vec::new() ou la macro vec! pour créer un vecteur avec des éléments initiaux :

```
fn main() {
    let mut vecteur_vide: Vec<i32> = Vec::new(); // Création d'un vecteur
vide.
    let vecteur_initial = vec![1, 2, 3, 4, 5]; // Création d'un vecteur
avec des valeurs initiales.
}
```

Voyons les détails de ce code :

- `Vec<i32>` : est le type de données du vecteur qui stocke des entiers 32 bits.
- `let mut vecteur_vide: Vec<i32> = Vec::new();` : crée un nouveau vecteur vide de type `i32`.
- `let vecteur_initial = vec![1, 2, 3, 4, 5];` : crée un vecteur avec des valeurs initiales `1`, `2`, `3`, `4` et `5`.
- `vec![]` : est une macro qui crée un vecteur avec des valeurs initiales.
- `!` : est utilisé pour appeler une macro.

Modification et Accès aux Éléments

Modification

Vous pouvez ajouter des éléments à un vecteur en utilisant la méthode **push** :

- **push** : ajoute un élément à la fin du vecteur.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.push(6); // Ajout d'une valeur au vecteur.

    for element in &vecteur { // Pour chaque élément du vecteur...
        println!("Le vecteur est : {}", element); // Affichage de la valeur
de l'élément.
    }
}
```

- **pop** : Supprime et retourne le dernier élément du vecteur, s'il existe..

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
```

```
des valeurs.
    let dernier_element = vecteur.pop(); // Suppression du dernier élément
du vecteur.

    match dernier_element {
        Some(valeur) => println!("Le dernier élément est : {}", valeur),
        None => println!("Aucun élément trouvé"),
    }
}
```

- **insert(index, element)** : Insère un élément à une position donnée dans le vecteur..

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.insert(2, 6); // Insertion de la valeur 6 à l'index 2.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

- **remove(index)** : Supprime et retourne l'élément à la position donnée.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    let element = vecteur.remove(2); // Suppression de l'élément à l'index
2.

    println!("L'élément supprimé est : {}", element); // Affichage de la
valeur de l'élément supprimé.
}
```

- **reverse** : inverse les éléments du vecteur.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.reverse(); // Inversion des éléments du vecteur.

    println!("{:?}", vecteur); // Affichage du vecteur inversé.
}
```

- **append** : ajoute les éléments d'un autre vecteur à la fin du vecteur.

```

```

```rust
fn main() {
    let mut vecteur1 = vec![1, 2, 3]; // Création d'un vecteur avec des
valeurs.
    let vecteur2 = vec![4, 5, 6]; // Création d'un autre vecteur avec des
valeurs.
    vecteur1.append(&mut vecteur2); // Ajout des éléments du vecteur2 au
vecteur1.

    println!("{:?}", vecteur1); // Affichage du vecteur1.
}
```

- **extend** : ajoute les éléments d'un itérable à la fin du vecteur. Cela peut être un autre vecteur, une tranche, une chaîne de caractères, etc.

```rust
fn main() {
    let mut vecteur = vec![1, 2, 3]; // Création d'un vecteur avec des
valeurs.
    let tableau = [4, 5, 6]; // Création d'un tableau avec des valeurs.
    vecteur.extend(&tableau); // Ajout des éléments du tableau au vecteur.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

- **dedup** : supprime les éléments en double du vecteur.

```rust
fn main() {
    let mut vecteur = vec![1, 2, 2, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5, 5, 5]; //
Création d'un vecteur avec des valeurs.
    vecteur.dedup(); // Suppression des doublons.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

- **resize** : redimensionne le vecteur pour contenir le nombre spécifié d'éléments.

```rust
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.resize(10, 0); // Redimensionnement du vecteur pour contenir 10
éléments avec la valeur 0.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

- **truncate** : tronque le vecteur pour contenir le nombre spécifié d'éléments. Cela supprime les éléments supplémentaires.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.truncate(3); // Troncature du vecteur pour contenir 3 éléments.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

Accès aux Éléments

- **get(index)** : Retourne une option contenant une référence à l'élément à une position donnée.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let element = vecteur.get(2); // Récupération de l'élément à l'index 2.

    match element {
        Some(valeur) => println!("L'élément est : {}", valeur), //
Affichage de la valeur de l'élément.
        None => println!("Aucun élément trouvé"), // Affichage si aucun
élément n'est trouvé.
    }
}
```

- **first() / last()** : Retourne une option contenant une référence au premier / dernier élément.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let premier = vecteur.first(); // Récupération du premier élément.
    let dernier = vecteur.last(); // Récupération du dernier élément.

    match premier {
        Some(valeur) => println!("Le premier élément est : {}", valeur), //
Affichage de la valeur du premier élément.
        None => println!("Aucun élément trouvé"), // Affichage si aucun
élément n'est trouvé.
    }

    match dernier {
        Some(valeur) => println!("Le dernier élément est : {}", valeur), //
Affichage de la valeur du dernier élément.
        None => println!("Aucun élément trouvé"), // Affichage si aucun
élément n'est trouvé.
```

```
    }
}
```

Itération sur les éléments

- **iter** : renvoie un itérateur sur les éléments du vecteur.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.

    for element in vecteur.iter() { // Pour chaque élément du vecteur...
        println!("Le vecteur est : {}", element); // Affichage de la valeur
de l'élément.
    }
}
```

- **iter_mut** : renvoie un itérateur mutable sur les éléments du vecteur. Cela permet de modifier les éléments du vecteur pendant l'itération.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.

    for element in vecteur.iter_mut() { // Pour chaque élément mutable du
vecteur...
        *element += 1; // Incrémentation de la valeur de l'élément.
    }

    for element in &vecteur { // Pour chaque élément du vecteur...
        println!("Le vecteur est : {}", element); // Affichage de la valeur
de l'élément.
    }
}
```

- **into_iter** : consomme le vecteur et renvoie un itérateur sur ses éléments. Cela signifie que le vecteur n'est plus utilisable après l'appel de cette méthode.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.

    for element in vecteur.into_iter() { // Pour chaque élément du
vecteur...
        println!("Le vecteur est : {}", element); // Affichage de la valeur
de l'élément.
```

```
    }
}
```

Recherche et Sélection d'Éléments

- **contains(element)** : Vérifie si le vecteur contient un élément donné.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let contient = vecteur.contains(&3); // Vérification si le vecteur
contient la valeur 3.

    println!("Le vecteur contient la valeur 3 : {}", contient); //
Affichage si le vecteur contient la valeur 3 ou non.
}
```

- **sort() / sort_by(key)** : Trie les éléments du vecteur en place, éventuellement avec une fonction de tri personnalisée.

```
fn main() {
    let mut vecteur = vec![5, 3, 1, 2, 4]; // Création d'un vecteur avec
des valeurs.
    vecteur.sort(); // Tri des éléments du vecteur.

    println!("{:?}", vecteur); // Affichage du vecteur trié.
}
```

```
fn main() {
    let mut vecteur = vec![5, 3, 1, 2, 4]; // Création d'un vecteur avec
des valeurs.
    vecteur.sort_by(|a, b| b.cmp(a)); // Tri des éléments du vecteur dans
l'ordre inverse.

    println!("{:?}", vecteur); // Affichage du vecteur trié dans l'ordre
inverse.
}
```

Voyons ce code en détail :

- `|a, b| b.cmp(a)` : est une fonction de tri personnalisée qui trie les éléments dans l'ordre inverse.
- `b.cmp(a)` : compare les éléments `a` et `b` dans l'ordre inverse.
- `cmp` : est une méthode de comparaison qui renvoie un ordre de tri pour deux éléments.

- `vecteur.sort_by(|a, b| b.cmp(a))` : trie les éléments du vecteur dans l'ordre inverse.
- **binary_search_by** : recherche un élément dans le vecteur en utilisant un prédicat spécifié et renvoie son index.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let index = vecteur.binary_search_by(|element| element.cmp(&3)); //
Recherche de la valeur 3 dans le vecteur.

    match index {
        Ok(i) => println!("La valeur 3 est à l'index : {}", i), //
Affichage de l'index de la valeur 3.
        Err(_) => println!("La valeur 3 n'est pas trouvée"), // Affichage
si la valeur 3 n'est pas trouvée.
    }
}
```

- **binary_search_by_key** : recherche un élément dans le vecteur en utilisant une clé spécifiée et renvoie son index.

```
fn main() {
    let vecteur = vec!["un", "deux", "trois", "quatre", "cinq"]; //
Création d'un vecteur avec des valeurs.
    let index = vecteur.binary_search_by_key(&"quatre", |element|
element.len()); // Recherche de la valeur "quatre" dans le vecteur.

    match index {
        Ok(i) => println!("La valeur 'quatre' est à l'index : {}", i), //
Affichage de l'index de la valeur "quatre".
        Err(_) => println!("La valeur 'quatre' n'est pas trouvée"), //
Affichage si la valeur "quatre" n'est pas trouvée.
    }
}
```

- **binary_search(element)** : Recherche un élément donné dans le vecteur et renvoie l'index de l'élément s'il est trouvé.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let index = vecteur.binary_search(&3); // Recherche de la valeur 3 dans
le vecteur.

    match index {
        Ok(i) => println!("La valeur 3 est à l'index : {}", i), //
Affichage de l'index de la valeur 3.
```

```
        Err(_) => println!("La valeur 3 n'est pas trouvée"), // Affichage
si la valeur 3 n'est pas trouvée.
    }
}
```

Gestion des Collections Clé-Valeur

- **contains_key** : renvoie vrai si le vecteur contient la clé spécifiée, sinon faux. (Note : Cette méthode est généralement utilisée avec les hashmaps, pas les vecteurs.)

En Rust, la méthode `contains_key` est spécifique aux types de collections qui gèrent des paires clé-valeur, comme les `HashMap` ou les `BTreeMap`, et non aux vecteurs (`Vec<T>`). Les vecteurs ne possèdent pas de méthode `contains_key` car ils sont des collections indexées par des positions numériques, pas par des clés.

```
use std::collections::HashMap;

fn main() {
    let mut hashmap = std::collections::HashMap::new(); // Création d'une
nouvelle hashmap.
    hashmap.insert("un", 1); // Ajout d'une paire clé-valeur à la hashmap.
    hashmap.insert("deux", 2); // Ajout d'une autre paire clé-valeur à la
hashmap.

    let contient = hashmap.contains_key("un"); // Vérification si la
hashmap contient la clé "un".

    println!("La hashmap contient la clé 'un' : {}", contient); //
Affichage si la hashmap contient la clé "un" ou non.
}
```

```
use std::collections::HashMap;

fn main() {
  let mut scores = HashMap::new();

  scores.insert(String::from("Bleu"), 10);
  scores.insert(String::from("Rouge"), 50);

  // Vérifie si la clé "Bleu" existe dans le HashMap
  if scores.contains_key("Bleu") {
    println!("La score pour l'équipe Bleu est présent.");
  } else {
    println!("Aucun score trouvé pour l'équipe Bleu.");
  }

  // Vérifie si la clé "Vert" existe
  if scores.contains_key("Vert") {
    println!("La score pour l'équipe Vert est présent.");
```

```
    } else {
        println!("Aucun score trouvé pour l'équipe Vert.");
    }
}
```

Dans cet exemple :

- Un `HashMap`nommé `scores` est créé et rempli avec des scores pour deux équipes, `"Bleu"` et `"Rouge"`.
- La méthode `contains_key` est utilisée pour vérifier si une clé spécifique ("Bleu" ou "Vert") existe dans le `HashMap`.
- `contains_key` renvoie `true` si la clé est présente dans le `HashMap`, et `false` dans le cas contraire.
- L'utilisation de `contains_key` est donc pertinente dans des contextes où vous devez vérifier l'existence d'une clé spécifique dans un `HashMap` ou un `BTreeMap`, ce qui est utile pour éviter les doublons ou pour décider si une opération spécifique doit être effectuée en fonction de la présence ou non d'une clé.

Ce que sont les HashMap et les BTreeMap, et comment les utiliser, sera abordé dans un autre chapitre.

Taille et Capacité

- **len** : renvoie le nombre d'éléments dans le vecteur.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let taille = vecteur.len(); // Récupération de la taille du vecteur.

    println!("La taille du vecteur est : {}", taille); // Affichage de la
taille du vecteur.
}
```

- **capacity** : renvoie la capacité du vecteur. La capacité est le nombre d'éléments que le vecteur peut contenir sans allouer de nouveau stockage.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let capacite = vecteur.capacity(); // Récupération de la capacité du
vecteur.

    println!("La capacité du vecteur est : {}", capacite); // Affichage de
la capacité du vecteur.
}
```

- **clear** : supprime tous les éléments du vecteur.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.clear(); // Suppression de tous les éléments du vecteur.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

Accès Bas-Niveau aux Éléments

- **as_slice** : renvoie une tranche contenant une vue sur les éléments du vecteur.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let tranche = vecteur.as_slice(); // Récupération d'une tranche
contenant une vue sur les éléments du vecteur.

    println!("{:?}", tranche); // Affichage de la tranche.
}
```

- **as_ptr** : renvoie un pointeur vers les éléments du vecteur.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let pointeur = vecteur.as_ptr(); // Récupération du pointeur vers les
éléments du vecteur.

    println!("Le pointeur du vecteur est : {:?}", pointeur); // Affichage
du pointeur du vecteur.
}
```

- **as_mut_ptr** : renvoie un pointeur mutable vers les éléments du vecteur.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    let pointeur_mutable = vecteur.as_mut_ptr(); // Récupération du
pointeur mutable vers les éléments du vecteur.

    println!("Le pointeur mutable du vecteur est : {:?}",
pointeur_mutable); // Affichage du pointeur mutable du vecteur.
}
```

Modification et Gestion des Éléments

- **swap** : échange les éléments à deux index spécifiés. Cette méthode est utile pour permuter les éléments d'un vecteur.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.swap(1, 3); // Échange des éléments aux index 1 et 3.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

- **retain** : conserve uniquement les éléments qui satisfont le prédicat spécifié. Cela permet de filtrer les éléments d'un vecteur en fonction d'un prédicat.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    vecteur.retain(|&x| x % 2 == 0); // Conservation des éléments pairs.

    println!("{:?}", vecteur); // Affichage du vecteur.
}
```

- **drain_filter** : supprime et renvoie un itérateur sur les éléments du vecteur qui satisfont le prédicat spécifié.

```
fn main() {
    let mut vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec
des valeurs.
    let mut iterateur = vecteur.drain_filter(|&x| x % 2 == 0); //
Suppression des éléments pairs.

    for element in iterateur { // Pour chaque élément de l'itérateur...
        println!("L'élément supprimé est : {}", element); // Affichage de
la valeur de l'élément supprimé.
    }
}
```

Division et Filtrage

- **split** : divise le vecteur en deux à l'index spécifié.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
```

```
    let (gauche, droite) = vecteur.split_at(3); // Division du vecteur en
deux à l'index 3.

    println!("{:?}", gauche); // Affichage de la partie gauche du vecteur.
    println!("{:?}", droite); // Affichage de la partie droite du vecteur.
}
```

- **split_at** : divise le vecteur en deux à l'index spécifié.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let (gauche, droite) = vecteur.split_at(3); // Division du vecteur en
deux à l'index 3.

    println!("{:?}", gauche); // Affichage de la partie gauche du vecteur.
    println!("{:?}", droite); // Affichage de la partie droite du vecteur.
}
```

- **partition** : divise le vecteur en deux en fonction d'un prédicat.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let (pairs, impairs): (Vec<i32>, Vec<i32>) =
vecteur.into_iter().partition(|&x| x % 2 == 0); // Division du vecteur en
deux en fonction de la parité.

    println!("{:?}", pairs); // Affichage des éléments pairs du vecteur.
    println!("{:?}", impairs); // Affichage des éléments impairs du
vecteur.
}
```

Voyons les détails de ce code :

- `let (pairs, impairs): (Vec<i32>, Vec<i32>)` : déclare deux vecteurs `pairs` et `impairs` pour stocker les éléments pairs et impairs.
- `vecteur.into_iter().partition(|&x| x % 2 == 0)` : divise le vecteur en deux en fonction de la parité des éléments.
- vecteur.into_iter() : convertit le vecteur en un itérateur.
- partition(|&x| x % 2 == 0) : divise le vecteur en deux en fonction de la parité des éléments.
- `println!("{:?}", pairs);` : affiche les éléments pairs du vecteur.
- `println!("{:?}", impairs);` : affiche les éléments impairs du vecteur.

Aggrégation et Transformation

- **join** : concatène les éléments du vecteur avec une chaîne séparatrice. (Note : Cette méthode n'est pas directement disponible sur Vec en Rust, mais vous pouvez l'utiliser sur des slices ou en utilisant des méthodes de l'itérateur avec des vecteurs de chaînes.)

```
fn main() {
    let vecteur = vec!["un", "deux", "trois", "quatre", "cinq"]; //
Création d'un vecteur avec des valeurs.
    let chaine = vecteur.join(", "); // Concaténation des éléments du
vecteur avec une chaîne séparatrice.

    println!("Le vecteur concaténé est : {}", chaine); // Affichage du
vecteur concaténé.
}
```

- **count** : compte le nombre d'éléments qui satisfont le prédicat spécifié.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let nombre_pairs = vecteur.iter().filter(|&x| x % 2 == 0).count(); //
Comptage des éléments pairs.

    println!("Le nombre d'éléments pairs est : {}", nombre_pairs); //
Affichage du nombre d'éléments pairs.
}
```

Inspection et État de la Collection

- **is_empty** : renvoie vrai si le vecteur est vide, sinon faux.

```
fn main() {
    let vecteur = vec![1, 2, 3, 4, 5]; // Création d'un vecteur avec des
valeurs.
    let est_vide = vecteur.is_empty(); // Vérification si le vecteur est
vide.

    println!("Le vecteur est vide : {}", est_vide); // Affichage si le
vecteur est vide ou non.
}
```

Voyons les détails de ce code :

- `|&x|` : est une fermeture qui prend un élément du vecteur.
- `|&x| x % 2 == 0` : est une fermeture qui conserve uniquement les éléments pairs.
- `retain` : est une méthode qui conserve uniquement les éléments qui satisfont le prédicat spécifié.

- `println!("{:?}", vecteur);` : affiche le vecteur après avoir conservé uniquement les éléments pairs.

Vous pouvez découvrir d'autres méthodes de manipulation des vecteurs dans la documentation officielle de Rust à cette adresse : https://doc.rust-lang.org/std/vec/struct.Vec.html

Entrées utilisateur et Manipulation de Chaînes de Caractères

Entrées utilisateur

Lecture d'entrées utilisateur avec `std::io`

std::io est un module standard de Rust qui fournit des fonctionnalités pour la lecture et l'écriture de données à partir de la console, de fichiers et d'autres sources d'entrée/sortie. Il est couramment utilisé pour interagir avec l'utilisateur en lisant des entrées à partir de la console.

Lecture d'une chaîne de caractères

```
use std::io;

fn main() {
    println!("Entrez votre nom :");

    let mut nom = String::new(); // Création d'une nouvelle chaîne pour
stocker le nom.
    io::stdin().read_line(&mut nom).expect("Échec de la lecture de
l'entrée"); // Lecture du nom de l'utilisateur.

    println!("Bonjour, {}", nom); // Affichage du nom de l'utilisateur.
}
```

Voyons les détails de ce code :

- `let mut nom = String::new();` : crée une nouvelle chaîne vide pour stocker le nom de l'utilisateur.
- `io::stdin()` : renvoie un gestionnaire d'entrée pour la console.
- `read_line()` : lit une ligne d'entrée à partir de la console et stocke le résultat dans la chaîne `nom`.
- `expect();` : gère les erreurs de lecture de l'entrée et affiche un message d'erreur en cas d'échec.
- `io::stdin().read_line(&mut nom).expect("Échec de la lecture de l'entrée");` : lit une ligne d'entrée à partir de la console et stocke le résultat dans la chaîne `nom`.
- `println!("Bonjour, {}", nom);` : affiche le nom de l'utilisateur.

Exemple : Programme manipulant des données utilisateur

```rust
use std::io;

fn main() {
    let mut noms = Vec::new(); // Création d'un vecteur pour stocker les noms.
    let mut age = Vec::new(); // Création d'un vecteur pour stocker les âges.

    loop {
        println!("Entrez le nom de l'utilisateur (ou 'fin' pour terminer) :");
        let mut nom = String::new(); // Création d'une nouvelle chaîne pour stocker le nom.
        io::stdin().read_line(&mut nom).expect("Échec de la lecture de l'entrée"); // Lecture du nom de l'utilisateur.

        if nom.trim() == "fin" { // Vérification si l'utilisateur a entré "fin".
            break; // Sortie de la boucle si l'utilisateur a entré "fin".
        }

        noms.push(nom.trim().to_string()); // Ajout du nom à la liste des noms.

        println!("Entrez l'âge de l'utilisateur :");
        let mut age_utilisateur = String::new(); // Création d'une nouvelle chaîne pour stocker l'âge.
        io::stdin().read_line(&mut age_utilisateur).expect("Échec de la lecture de l'entrée"); // Lecture de l'âge de l'utilisateur.

        let age_utilisateur: u32 = match age_utilisateur.trim().parse() { // Conversion de l'âge en nombre entier non signé.
            Ok(valeur) => valeur, // Récupération de la valeur si la conversion réussit.
            Err(_) => {
                println!("Veuillez entrer un nombre valide."); // Affichage d'un message d'erreur si la conversion échoue.
                continue; // Retour au début de la boucle.
            }
        };

        age.push(age_utilisateur); // Ajout de l'âge à la liste des âges.
    }

    println!("Les noms des utilisateurs sont : {:?}", noms); // Affichage des noms des utilisateurs.
    println!("Les âges des utilisateurs sont : {:?}", age); // Affichage des âges des utilisateurs.
}
```

Gestion de projets avec Cargo

Cargo est l'outil de gestion de paquets et de construction de projets de Rust. Il est utilisé pour créer, construire, tester et publier des projets Rust, ainsi que pour gérer les dépendances et les bibliothèques externes. Cargo facilite le développement de projets Rust en automatisant de nombreuses tâches courantes, telles que la gestion des dépendances, la compilation du code, l'exécution des tests et la génération de la documentation.

Création d'un nouveau projet

Pour créer un nouveau projet Rust avec Cargo, vous pouvez utiliser la commande `cargo new` suivie du nom du projet. Par exemple, pour créer un projet nommé `mon_projet`, vous pouvez exécuter la commande suivante dans votre terminal :

```
cargo new mon_projet
```

Cette commande crée un nouveau répertoire nommé `mon_projet` contenant les fichiers et répertoires suivants :

```
mon_projet
├── Cargo.toml
└── src
    └── main.rs
```

Dépendances et Crates

En Rust, les bibliothèques et les paquets sont appelés des "crates". Les dépendances externes sont spécifiées dans le fichier `Cargo.toml` à la racine du projet. Voici un exemple de fichier `Cargo.toml` avec des dépendances spécifiées :

```toml
[package]
name = "mon_projet"
version = "0.1.0"
edition = "2018"

[dependencies]
rand = "0.8.4"
```

Dans cet exemple, la section `[dependencies]` spécifie les dépendances du projet. La dépendance `rand` est spécifiée avec la version `0.8.4`. Lorsque vous exécutez `cargo build` ou `cargo run`, Cargo télécharge et installe automatiquement les dépendances spécifiées à partir du registre de crates de Rust.

Tests

Les tests unitaires peuvent être écrits dans le fichier `src/main.rs` ou dans des fichiers séparés dans le répertoire `tests`. Voici un exemple de test unitaire dans le fichier `src/main.rs` :

```rust
fn somme(a: i32, b: i32) -> i32 {
    a + b
}

#[cfg(test)]

mod tests {
    use super::*;

    #[test]
    fn test_somme() {
        assert_eq!(somme(2, 2), 4);
        assert_eq!(somme(1, 3), 4);
        assert_eq!(somme(-1, 1), 0);
    }
}
```

Dans cet exemple, la fonction `somme` est testée avec plusieurs cas de test à l'aide de la macro `#[test]`. Lorsque vous exécutez `cargo test`, Cargo exécute les tests unitaires et affiche les résultats.

Documentation

La documentation du projet peut être générée à l'aide de la commande `cargo doc`. Cette commande génère la documentation du projet à partir des commentaires de documentation (doc comments) dans le code source. La documentation générée est stockée dans le répertoire `target/doc` et peut être consultée en ouvrant le fichier `index.html` dans un navigateur web.

Exemple : Création d'un projet avec dépendances externes

```rust
use rand::Rng;

fn main() {
    let nombre_secret = rand::thread_rng().gen_range(1..101);
    println!("Devinez le nombre secret entre 1 et 100.");

    loop {
        let mut supposition = String::new();
        std::io::stdin().read_line(&mut supposition).expect("Échec de la
lecture de l'entrée");
        let supposition: u32 = match supposition.trim().parse() {
            Ok(valeur) => valeur,
            Err(_) => {
                println!("Veuillez entrer un nombre valide.");
                continue;
            }
        };

        match supposition.cmp(&nombre_secret) {
            std::cmp::Ordering::Less => println!("Trop petit !"),
            std::cmp::Ordering::Greater => println!("Trop grand !"),
```

```
            std::cmp::Ordering::Equal => {
                println!("Bravo, vous avez deviné le nombre secret !");
                break;
            }
        }
    }
}
```

Analysons ce code en détail :

- `use rand::Rng;` : importe le trait `Rng` du crate `rand` pour générer un nombre aléatoire.
- `let nombre_secret = rand::thread_rng().gen_range(1..101);` : génère un nombre aléatoire entre 1 et 100 à l'aide de la méthode `gen_range` du générateur de nombres aléatoires `thread_rng`.
- `std::io::stdin().read_line(&mut supposition).expect("Échec de la lecture de l'entrée");` : lit une ligne d'entrée à partir de la console et stocke le résultat dans la chaîne `supposition`.
- `match supposition.cmp(&nombre_secret)` : compare la supposition de l'utilisateur avec le nombre secret et affiche un message en fonction du résultat de la comparaison.
- `std::cmp::Ordering::Less` : indique que la supposition est inférieure au nombre secret.
- `std::cmp::Ordering::Greater` : indique que la supposition est supérieure au nombre secret.
- `std::cmp::Ordering::Equal` : indique que la supposition est égale au nombre secret.
- `continue;` : passe à l'itération suivante de la boucle.
- `break;` : sort de la boucle.

Dans cet exemple, le projet utilise la dépendance externe `rand` pour générer un nombre aléatoire. La dépendance `rand` est spécifiée dans le fichier `Cargo.toml` comme suit :

```
[dependencies]
rand = "0.8.4"
```

Construction et Exécution du Projet

Pour construire et exécuter un projet Rust avec Cargo, vous pouvez utiliser les commandes suivantes :

- `cargo build` : compile le projet et génère un exécutable dans le répertoire `target/debug`.
- `cargo run` : compile et exécute le projet.
- `cargo test` : exécute les tests unitaires du projet.
- `cargo doc` : génère la documentation du projet.
- `cargo clean` : supprime les fichiers générés par Cargo.
- `cargo update` : met à jour les dépendances du projet.
- `cargo check` : vérifie la validité du code sans le compiler.
- `cargo fmt` : formate le code selon les conventions de style de Rust.
- `cargo clippy` : exécute l'analyseur statique de code Clippy pour détecter les erreurs et les problèmes de style.
- `cargo install` : installe un exécutable Rust à partir d'un crate.
- `cargo publish` : publie un crate sur le registre de crates de Rust.

- `cargo search` : recherche un crate sur le registre de crates de Rust.
- `cargo new` : crée un nouveau projet Rust.
- `cargo init` : initialise un nouveau projet Rust dans un répertoire existant.

Programmation concurrente et Parallélisme

La programmation concurrente et le parallélisme sont des concepts fondamentaux pour améliorer l'efficacité et la performance des applications modernes. Rust offre un excellent soutien pour ces paradigmes à travers plusieurs abstractions. Voici un aperçu des concepts et un exemple de leur utilisation.

Threads

Les threads permettent à votre programme d'exécuter plusieurs tâches simultanément. Rust fournit une abstraction pour travailler avec des threads de manière à éviter de nombreux problèmes courants, comme les conditions de course ou les deadlocks.

Création de Threads

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| { // Création d'un nouveau thread.
        // Code exécuté dans un nouveau thread.
        for i in 1..=5 { // Pour chaque nombre de 1 à 5...
            println!("Nouveau thread : {}", i);
            thread::sleep(std::time::Duration::from_millis(500)); // Mise
en sommeil du thread.
        }
    });

    for i in 1..=3 {
        println!("Thread principal : {}", i);
        thread::sleep(std::time::Duration::from_millis(1000)); // Mise en
sommeil du thread principal.
    }

    handle.join().unwrap(); // Attente de la fin du thread.
}
```

Analysons ce code en détail :

- `let handle = thread::spawn(|| { ... });` : crée un nouveau thread et exécute le code spécifié dans la fermeture.
- `thread::sleep(std::time::Duration::from_millis(500));` : met le thread en sommeil pendant 500 millisecondes.
- `handle.join().unwrap();` : attend la fin du thread et récupère le résultat.
- `thread::sleep(std::time::Duration::from_millis(1000));` : met le thread principal en sommeil pendant 1000 millisecondes.

- `println!("Nouveau thread : {}", i);` : affiche le numéro de l'itération dans le nouveau thread.
- `println!("Thread principal : {}", i);` : affiche le numéro de l'itération dans le thread principal.
- `for i in 1..=5 { ... }` : exécute une boucle dans le nouveau thread. de 1 à 5.

Canaux de communication

Les canaux de communication permettent à différents threads d'échanger des données de manière asynchrone. Rust fournit une abstraction pour travailler avec des canaux de manière à éviter de nombreux problèmes courants, comme les conditions de course ou les deadlocks.

Rust propose des canaux (channels) pour la communication entre threads. Un canal a deux extrémités : un émetteur (sender) et un récepteur (receiver).

Création d'un canal

```
use std::sync::mpsc;
use std::thread;

fn main() {
    let (emetteur, recepteur) = mpsc::channel(); // Création d'un canal
pour la communication entre threads.

    thread::spawn(move || { // Création d'un nouveau thread.
        let message = "Bonjour depuis le thread émetteur !".to_string();
        emetteur.send(message).unwrap(); // Envoi d'un message à travers le
canal.
    });

    let message_recu = recepteur.recv().unwrap(); // Réception du message à
travers le canal.
    println!("Message reçu : {}", message_recu); // Affichage du message
reçu.
}
```

Analysons ce code en détail :

- `let (emetteur, recepteur) = mpsc::channel();` : crée un canal pour la communication entre threads.
- `thread::spawn(move || { ... });` : crée un nouveau thread et exécute le code spécifié dans la fermeture.
- `emetteur.send(message).unwrap();` : envoie un message à travers le canal.
- `let message_recu = recepteur.recv().unwrap();` : reçoit un message à travers le canal.
- `println!("Message reçu : {}", message_recu);` : affiche le message reçu.
- `let message = "Bonjour depuis le thread émetteur !".to_string();` : crée un message à envoyer à travers le canal.

Modèle de mémoire partagée

Le modèle de mémoire partagée permet à plusieurs threads d'accéder et de modifier des données partagées de manière synchronisée. Rust fournit une abstraction pour travailler avec la mémoire partagée de manière à éviter de nombreux problèmes courants, comme les conditions de course ou les deadlocks.

Rust utilise le concept de propriété et d'emprunt pour éviter les problèmes dans un environnement concurrent. Les types `Arc` et `Mutex` sont souvent utilisés ensemble pour partager un état entre des threads de manière sécurisée.

Exemple : Mémoire partagée avec Mutex

```
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let donnees_partagees = Arc::new(Mutex::new(0)); // Création d'une
mémoire partagée avec Mutex.

    let mut threads = vec![]; // Création d'un vecteur pour stocker les
threads.

    for _ in 0..5 {
        let donnees_partagees = Arc::clone(&donnees_partagees); // Clonage
de la mémoire partagée pour chaque thread.
        let thread = thread::spawn(move || { // Création d'un nouveau
thread.
            let mut donnees = donnees_partagees.lock().unwrap(); //
Verrouillage de la mémoire partagée.
            *donnees += 1; // Incrémentation des données partagées.
        });
        threads.push(thread); // Ajout du thread au vecteur.
    }

    for thread in threads {
        thread.join().unwrap(); // Attente de la fin de chaque thread.
    }

    println!("Données partagées : {}", *donnees_partagees.lock().unwrap());
// Affichage des données partagées.
}
```

Analysons ce code en détail :

- `let donnees_partagees = Arc::new(Mutex::new(0));` : crée une mémoire partagée avec Mutex.
- `let donnees_partagees = Arc::clone(&donnees_partagees);` : clone la mémoire partagée pour chaque thread.
- `let mut donnees = donnees_partagees.lock().unwrap();` : verrouille la mémoire partagée.
- `*donnees += 1;` : incrémente les données partagées.

- `thread.join().unwrap();` : attend la fin de chaque thread.
- `println!("Données partagées : {}", *donnees_partagees.lock().unwrap());` : affiche les données partagées.
- `let mut threads = vec![];` : crée un vecteur pour stocker les threads.
- `for _ in 0..5 { ... }` : crée cinq threads pour incrémenter les données partagées.
- `let thread = thread::spawn(move || { ... });` : crée un nouveau thread et exécute le code spécifié dans la fermeture.
- `for thread in threads { ... }` : attend la fin de chaque thread.
- `let donnees_partagees = Arc::clone(&donnees_partagees);` : clone la mémoire partagée pour chaque thread.

Exemple : Programme téléchargeant des données en parallèle

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

fn telecharger_donnees(url: &str, donnees: Arc<Mutex<Vec<u8>>>) {
    // Simulation du téléchargement de données à partir de l'URL.
    thread::sleep(Duration::from_secs(2));
    let donnees_telechargees = b"Exemple de données téléchargées";
    let mut donnees = donnees.lock().unwrap();
    donnees.extend_from_slice(donnees_telechargees);
}

fn main() {
    let donnees = Arc::new(Mutex::new(Vec::new())); // Création d'une
mémoire partagée avec Mutex pour stocker les données téléchargées.

    let urls = vec!["http://exemple.com/fichier1",
"http://exemple.com/fichier2", "http://exemple.com/fichier3"];

    let mut threads = vec![]; // Création d'un vecteur pour stocker les
threads.

    for url in urls {
        let donnees = Arc::clone(&donnees); // Clonage de la mémoire
partagée pour chaque thread.
        let thread = thread::spawn(move || { // Création d'un nouveau
thread.
            telecharger_donnees(url, donnees); // Téléchargement des
données à partir de l'URL.
        });
        threads.push(thread); // Ajout du thread au vecteur.
    }

    for thread in threads {
        thread.join().unwrap(); // Attente de la fin de chaque thread.
    }

    let donnees = donnees.lock().unwrap(); // Verrouillage de la mémoire
```

```
partagée.
    println!("Données téléchargées : {:?}", *donnees); // Affichage des
données téléchargées.
}
```

Projets et Ressources Complémentaires

Rust est un langage puissant et flexible, idéal pour une multitude de projets, allant des outils en ligne de commande aux serveurs web et aux systèmes embarqués. Voici des idées pour mettre en pratique vos connaissances en Rust, suivies de ressources pour approfondir votre apprentissage et vous connecter à la communauté.

Créer un projet en Rust

Idées de projets pour appliquer les connaissances acquises

1. **Interpréteur de Langage de Programmation**: Développez un simple interpréteur pour un langage de programmation inventé ou existant. Cela peut vous aider à comprendre les principes de la compilation et de l'interprétation.
2. **Serveur Web**: Utilisez `hyper` ou `actix-web` pour créer un serveur web performant. Commencez par servir des pages statiques avant de passer à des applications web dynamiques.
3. **Outils en Ligne de Commande**: Créez des outils pour simplifier les tâches courantes sur votre système. Vous pourriez développer un programme pour gérer vos notes ou automatiser des tâches répétitives.
4. **Jeu en 2D**: Avec `ggez` ou `piston`, vous pouvez créer des jeux simples pour explorer la programmation graphique et de jeu.
5. **Client Torrent**: Implémentez un client torrent simple pour comprendre les réseaux et la communication peer-to-peer.

Étapes de développement d'un projet

1. **Conception et Planification**: Définissez clairement le but de votre projet. Planifiez les fonctionnalités principales et l'interface utilisateur si nécessaire.
2. **Environnement de Développement**: Configurez votre environnement avec Rust et les outils nécessaires (Cargo, Clippy, Rustfmt).
3. **Prototypage**: Commencez par coder les fonctionnalités de base. Un prototype fonctionnel vous permet de tester rapidement des idées.
4. **Itération**: Développez votre projet en ajoutant des fonctionnalités, en corrigeant des bugs et en améliorant les performances.
5. **Tests**: Écrivez des tests unitaires et d'intégration pour assurer la fiabilité de votre code.

6. **Documentation**: Documentez votre code et créez un README utile pour expliquer comment utiliser votre projet.

7. **Publication**: Publiez votre code sur GitHub et partagez-le avec la communauté. Considérez la publication de votre projet sur crates.io si vous avez créé une bibliothèque.

Ressources complémentaires et communauté

Livres et documentation en ligne

- **The Rust Programming Language**: Le livre officiel de Rust, disponible gratuitement en ligne à l'adresse https://doc.rust-lang.org/book/.
- **Rust by Example**: Un autre livre officiel de Rust, disponible gratuitement en ligne à l'adresse https://doc.rust-lang.org/rust-by-example/.
- **Rust Reference**: La documentation de référence officielle de Rust, disponible gratuitement en ligne à l'adresse https://doc.rust-lang.org/reference/.
- **Rustonomicon**: Un livre sur les aspects sombres de Rust, disponible gratuitement en ligne à l'adresse https://doc.rust-lang.org/nomicon/.

Forums, groupes et conférences

- **Rust Users Forum**: Un forum de discussion pour les utilisateurs de Rust, disponible à l'adresse https://users.rust-lang.org/.
- **Rust Subreddit**: Un subreddit dédié à Rust, disponible à l'adresse https://www.reddit.com/r/rust/.
- **Rust Discord**: Un serveur Discord pour discuter de Rust, disponible à l'adresse https://discord.gg/rust-lang.

Outils et bibliothèques

- **Crates.io**: Le registre de crates de Rust, disponible à l'adresse https://crates.io/.
- **Rustup**: Un outil pour gérer les versions de Rust et les outils associés, disponible à l'adresse https://rustup.rs/.
- **Clippy**: Un linter pour Rust, disponible à l'adresse https://doc.rust-lang.org/clippy/.
- **Rustfmt**: Un formateur de code pour Rust, disponible à l'adresse https://doc.rust-lang.org/beta/nightly-rustc/rustfmt/index.html.
- **Rust Analyzer**: Un analyseur de code pour Rust, disponible à l'adresse https://rust-analyzer.github.io/.

Conclusion

Ce guide a couvert les bases de la programmation en Rust, y compris les types de données, les structures de contrôle, les fonctions, les modules, les collections, les entrées utilisateur, la manipulation de chaînes de caractères, la gestion de projets avec Cargo, la programmation concurrente et le parallélisme. Nous avons également exploré des idées de projets et des ressources pour approfondir votre apprentissage et vous connecter à la communauté.

Révision et prochaines étapes

Récapitulatif des concepts clés

- **Types de données et variables**: Rust offre une variété de types de données, y compris les types primitifs, les types composés et les types personnalisés.
- **Structures de contrôle**: Les structures de contrôle, telles que les boucles et les instructions conditionnelles, permettent de contrôler le flux d'exécution du programme.
- **Fonctions et modules**: Les fonctions et les modules permettent d'organiser et de réutiliser le code de manière efficace.
- **Collections**: Les collections, telles que les vecteurs et les chaînes de caractères, offrent des moyens flexibles de stocker et de manipuler des données.
- **Entrées utilisateur et manipulation de chaînes de caractères**: La bibliothèque standard de Rust offre des fonctionnalités pour lire des entrées utilisateur et manipuler des chaînes de caractères.
- **Gestion de projets avec Cargo**: Cargo est l'outil de gestion de paquets et de construction de projets de Rust, qui facilite le développement de projets Rust en automatisant de nombreuses tâches courantes.
- **Programmation concurrente et parallélisme**: Rust offre un excellent soutien pour la programmation concurrente et le parallélisme à travers plusieurs abstractions, telles que les threads, les canaux de communication et la mémoire partagée.
- **Projets et ressources complémentaires**: Vous pouvez appliquer vos connaissances en Rust en créant des projets et en explorant les ressources disponibles pour approfondir votre apprentissage et vous connecter à la communauté.

Prochaines étapes

- **Créez un projet en Rust**: Utilisez les idées de projets pour créer un projet en Rust et appliquer vos connaissances.
- **Explorez les ressources complémentaires**: Utilisez les livres, la documentation en ligne, les forums, les groupes et les conférences pour approfondir votre apprentissage et vous connecter à la communauté.
- **Partagez votre travail**: Publiez votre projet sur GitHub, partagez-le avec la communauté et contribuez à l'écosystème de Rust.
- **Continuez à apprendre et à pratiquer**: La programmation en Rust est un voyage continu. Continuez à explorer de nouveaux concepts, à résoudre des problèmes et à développer des projets pour renforcer vos compétences.

Conseils pour continuer à apprendre et pratiquer

- **Explorez des projets open source**: Contribuez à des projets open source en Rust pour acquérir de l'expérience pratique et apprendre des meilleures pratiques.
- **Participez à des événements et des hackathons**: Rejoignez des événements et des hackathons pour rencontrer d'autres développeurs, apprendre de nouvelles techniques et développer des projets en équipe.
- **Créez des projets personnels**: Trouvez des problèmes intéressants à résoudre et créez des projets personnels pour explorer de nouveaux concepts et technologies.
- **Enseignez et partagez vos connaissances**: Enseignez aux autres ce que vous avez appris en écrivant des articles, en donnant des présentations ou en aidant les autres développeurs.
- **Restez curieux et ouvert d'esprit**: La programmation en Rust est un domaine en constante évolution. Restez curieux, ouvert d'esprit et prêt à apprendre de nouvelles choses.

- Vous pouver utiliser le playground de Rust pour tester vos codes en ligne : https://play.rust-lang.org/

Remerciements et ressources

Merci d'avoir suivi ce guide sur la programmation en Rust. Pour en savoir plus sur Rust, consultez les ressources suivantes :

- The Rust Programming Language
- Rust by Example****