



Rust

Introduction



Qu'est ce que Rust?

Rust est un langage de programmation moderne et performant, conçu pour offrir sécurité, rapidité, et contrôle précis de la mémoire.

Créé par Graydon Hoare et sponsorisé par Mozilla, Rust est désormais largement utilisé pour développer des systèmes, des logiciels embarqués, des applications web, et même des jeux vidéo.



Qu'est ce que Rust?

Les Points Forts de Rust

1. **Sécurité Mémoire** : Rust garantit une gestion sûre de la mémoire en évitant les erreurs courantes (comme les null pointer, les fuites de mémoire, les accès concurrents non sécurisés) grâce à un système de vérification de la mémoire à la compilation, sans nécessiter de ramasse-miettes (garbage collector).
2. **Performance et Concurrency** : Aussi rapide que le C et le C++, Il est idéal pour les applications nécessitant un haut niveau de performance, comme les moteurs de jeux ou les systèmes d'exploitation. Sa gestion de la concurrence et de l'asynchrone est puissante et intuitive, permettant de créer des applications performantes et réactives.
3. **Syntaxe Moderne et Expressive** : Rust possède une syntaxe influencée par des langages modernes comme Python et JavaScript, tout en maintenant la rigueur du C/C++. Ce compromis permet d'écrire du code performant et compréhensible, même pour des systèmes complexes.
4. **Gestion stricte de la propriété (Ownership)** : Le modèle de propriété de Rust (ou ownership) introduit un paradigme unique dans la gestion de la mémoire, où chaque valeur a un propriétaire unique. Les règles d'ownership permettent de garantir à la compilation que les accès mémoires sont sûrs et minimisent le risque de bugs de concurrence ou de segmentation.
5. **Écosystème de développement robuste** : Avec son gestionnaire de paquets et d'environnement (cargo), Rust facilite le développement, le test et le déploiement des projets. La bibliothèque standard est riche, et des milliers de bibliothèques (crates) sont disponibles pour accélérer le développement.



Qu'est ce que Rust?

Cas d'Utilisation de Rust

- **Développement de Systèmes** : Rust est idéal pour les projets de bas niveau, comme les OS (systèmes d'exploitation). Par exemple, le noyau de l'OS Redox est écrit en Rust.
- **Applications Web et Réseau** : Grâce aux frameworks comme Actix et Rocket, Rust est également utilisé pour créer des serveurs web rapides et sécurisés.
- **Traitement de Données et WebAssembly** : Rust peut être compilé en WebAssembly, le rendant utilisable côté client dans des applications web ou pour des traitements intensifs côté serveur.
- **Programmation Embarquée** : Sa performance et sa gestion mémoire fine en font un excellent choix pour des applications embarquées, où les ressources sont souvent limitées.



Qu'est ce que Rust?

Rust est souvent comparé à C et C++ pour sa performance, mais il est plus sûr grâce à ses vérifications strictes à la compilation.

Par rapport aux langages gérés par un ramasse-miettes (comme Go, Java, ou Python), Rust est plus performant dans la gestion de la mémoire et offre plus de contrôle, ce qui est crucial pour les applications systèmes ou les applications critiques en performance.



Rust?

Voici à quoi ressemble un programme
basique en Rust :

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



Qu'est ce que Rust?

L'Écosystème Rust : Cargo, Crates.io et Documentation

Rust bénéficie de cargo, un gestionnaire de paquets et d'outils intégré qui facilite la compilation, la gestion des dépendances, et la publication de projets.

Le site Crates.io héberge des milliers de bibliothèques open-source, permettant de rapidement intégrer des fonctionnalités variées (réseau, cryptographie, etc.) dans un projet Rust.



Installation de Rust

L'installation de Rust est simple et rapide grâce à son outil de gestion intégré, rustup.

Voici les étapes détaillées pour installer et configurer Rust, quel que soit le système d'exploitation.

***Vous pouvez aussi suivre directement l'installation sur:
<https://www.rust-lang.org/tools/install>***



Installation de Rust

1. Ouvrez un terminal (ou PowerShell sur Windows).
2. Exécutez la commande suivante pour installer rustup : **curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh**
3. Cette commande va télécharger et exécuter le script d'installation de Rust. Elle installe le compilateur Rust, ainsi que l'outil cargo, qui est essentiel pour la gestion des paquets et la compilation des projets Rust.
4. Suivez les instructions dans le terminal. Vous pouvez généralement appuyer sur 1 pour choisir l'installation par défaut.
5. Ajoutez Rust à votre PATH (si nécessaire). Le script devrait normalement le faire automatiquement, mais vous pouvez vérifier avec : **source \$HOME/.cargo/env**

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh  
  
source $HOME/.cargo/env
```

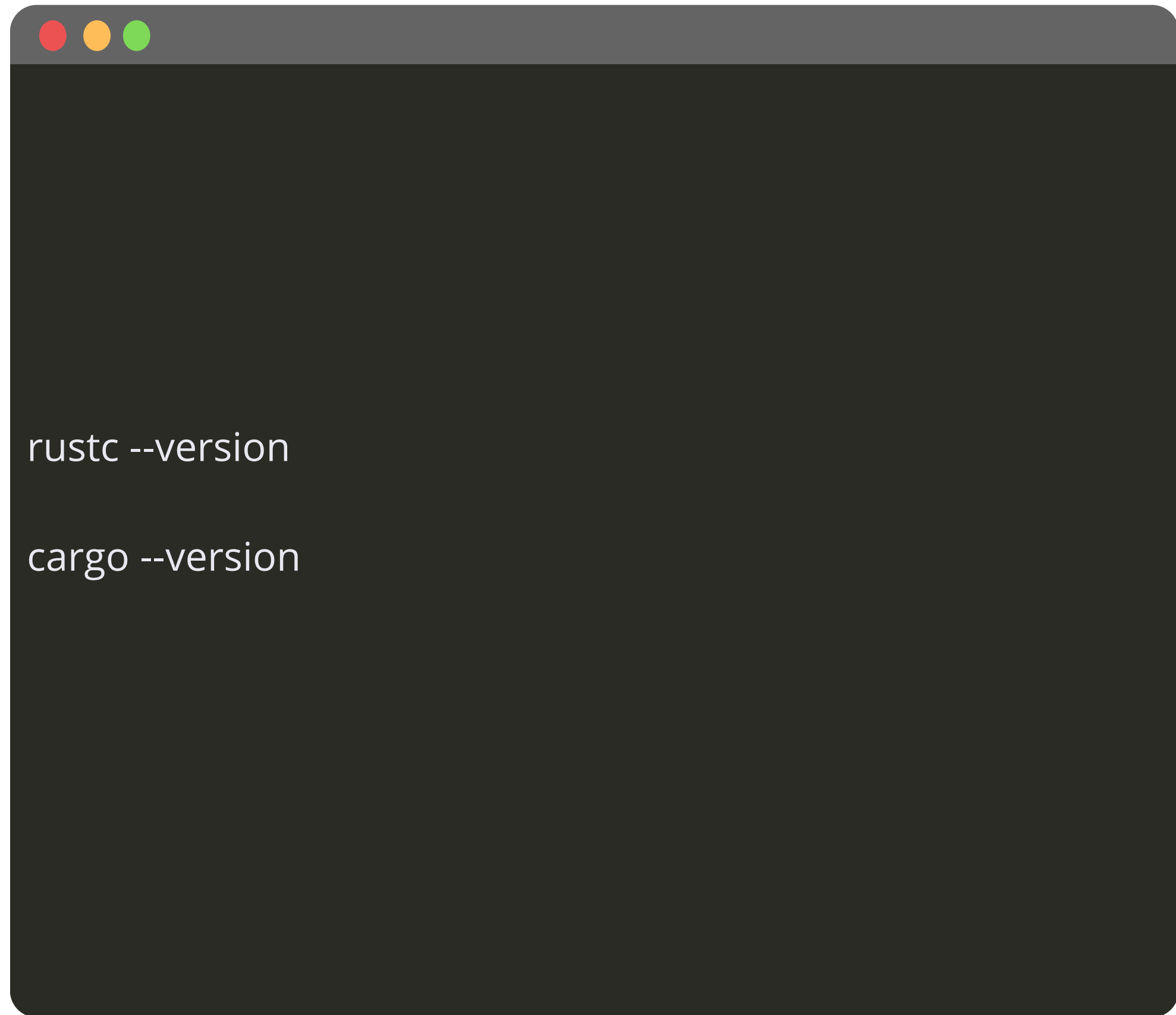


Vérifier l'installation

Après l'installation, vous pouvez vérifier que Rust est bien installé en exécutant les commandes suivantes dans le terminal :

1. `rustc --version`
2. `cargo --version`

Les deux commandes devraient renvoyer des informations sur les versions installées.



Configuration de Rust

Rust dispose de trois toolchains principaux :

- **Stable** : version stable recommandée pour la majorité des projets.
- **Beta** : version de test avant la prochaine stable.
- **Nightly** : version en développement avec les toutes dernières fonctionnalités.

Vous pouvez installer et utiliser différentes versions avec rustup.

Pour installer et utiliser la version nightly, par ex :
rustup install nightly
rustup default nightly

Pour revenir à la version stable, utilisez :
rustup default stable

```
//Pour installer et utiliser la version nightly, par ex :  
rustup install nightly  
rustup default nightly
```

```
//Pour revenir à la version stable, utilisez :  
rustup default stable
```



Mise à jour de Rust

Pour mettre à jour Rust vers la dernière version stable, exécutez :

```
rustup update
```



Installation de l'éditeur

Visual Studio Code (VSCode): VSCode est un éditeur très populaire pour Rust grâce à son intégration avec l'extension Rust Analyzer.

1. Ouvrez VSCode et allez dans le menu des extensions.
2. Recherchez et installez l'extension Rust Analyzer.
3. Configurez Rust Analyzer : une fois l'extension installée, elle devrait automatiquement détecter votre installation de Rust.

Rust Analyzer offre :

- Autocomplétion
- Documentation contextuelle
- Diagnostic des erreurs en temps réel

Autres Éditeurs: Rust est également compatible avec d'autres éditeurs populaires :

- IntelliJ IDEA : via l'extension Rust pour JetBrains.
- Vim/Neovim : en installant Rust Analyzer avec des plugins comme Coc.nvim.
- Atom : avec l'extension ide-rust.



Création d'un projet

Cargo est l'outil de gestion de projets et de dépendances de Rust. Pour créer un nouveau projet Rust, exécutez dans le terminal, au sein du dossier de votre choix:

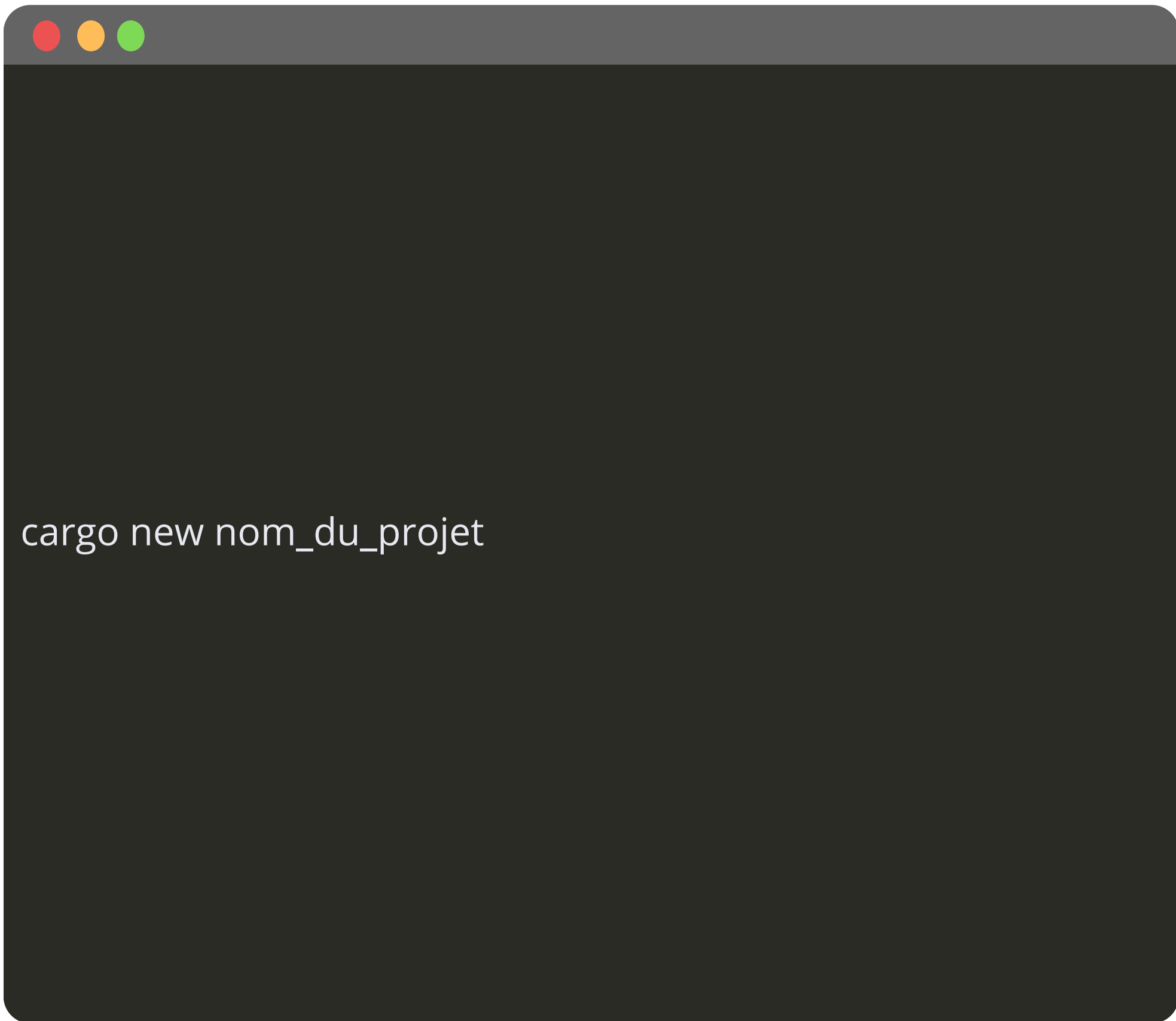
cargo new nom_du_projet

Cette commande crée un dossier avec la structure de base d'un projet Rust, incluant :

- Un fichier Cargo.toml pour gérer les dépendances et les métadonnées du projet.
- Un dossier src contenant le fichier main.rs.

Vous pouvez ensuite compiler et exécuter le projet avec :

cargo run



Les bases de Rust



Structure d'un projet

La structure d'un projet Rust est généralement standardisée et organisée pour faciliter le développement, la maintenance, et la gestion des dépendances.

Rust utilise l'outil de gestion de projets Cargo pour initialiser et gérer la structure de base. D'ailleurs, vous l'avez vu, nous avons utilisé Cargo pour créer un projet.

Voici à quoi ressemble la structure typique d'un projet Rust: Lorsque vous créez un nouveau projet Rust avec `cargo new`, vous obtenez une structure de base comme celle-ci :

mon_projet/

- *Cargo.toml*
- *Cargo.lock*
- *src/*
 - *main.rs*



Structure d'un projet

Cargo.toml :

Le fichier de configuration principal du projet. Il contient les métadonnées du projet, les dépendances et les informations de compilation.

Cargo.lock :

Ce fichier est généré automatiquement pour verrouiller les versions précises des dépendances installées.

src/ :

Ce répertoire contient le code source principal de votre projet.

main.rs :

Le fichier d'entrée principal pour un binaire Rust. C'est ici que se trouve la fonction main() si le projet est une application.



Structure d'un projet

Utiliser cargo pour Gérer le Projet

- **Construire le projet** : *cargo build*
- **Exécuter le projet** : *cargo run*
- **Tester le projet** : *cargo test*
- **Documenter le projet** : *cargo doc --open*



fn main()

La fonction `fn main()` est le point d'entrée d'un programme Rust. C'est ici que commence l'exécution du programme.

Décomposons les éléments de cette structure :

1. Déclaration de la fonction main :

- `fn` : Déclare une fonction en Rust.
- `main` : Nom de la fonction par laquelle le programme commence.
- `()` : Les parenthèses marquent le début et la fin de la liste des paramètres.

2. Corps de la fonction `{}` :

- Tout le code à exécuter est placé à l'intérieur des accolades `{}`.
- Dans cet exemple, `println!` est une macro qui affiche du texte sur la console.

3. **Point-virgule ;** : Chaque ligne de code est suivie d'un point-virgule pour indiquer la fin de l'instruction.

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



Print et ses variantes

Définition: la fonction `print` et ses variantes (`println!`, `eprint!`, et `eprintln!`) sont utilisées pour afficher du texte à l'écran ou pour écrire dans des flux de sortie. Vous remarquerez qu'ils contiennent tous un `!`. Ce qui signifie qu'il s'agit d'une fonction macro. Nous en parlerons juste après.

1. `println!` : La macro `println!` est utilisée pour afficher du texte avec un retour à la ligne à la fin. C'est la plus utilisée.

- `println!("Hello, world!");` // Affiche "Hello, world!" suivi d'un retour à la ligne

2. `print!` : La macro `print!` fonctionne comme `println!`, mais sans ajouter de retour à la ligne à la fin.

- `print!("Hello, ");` // Affiche "Hello, " sans retour à la ligne
- `print!("world!");` // Affiche "world!" immédiatement après
- Ce code affichera : Hello, world! sur la même ligne.

3. `eprintln!` et `eprint!` : Les macros `eprintln!` et `eprint!` fonctionnent comme `println!` et `print!`, mais elles écrivent dans le flux d'erreur standard (`stderr`) au lieu du flux de sortie standard (`stdout`). Elles sont particulièrement utiles pour afficher des messages d'erreur.

- `eprintln!` : Affiche un message d'erreur avec un retour à la ligne.
- `eprint!` : Affiche un message d'erreur sans retour à la ligne.



print et interpolation

Vous pouvez insérer des valeurs variables dans le texte avec des accolades {} :

```
fn main() {  
  let name = "Alice";  
  let age = 25;  
  println!("Name: {}, Age: {}", name, age);  
}
```



print et formatage

Rust offre un puissant système de formatage pour personnaliser l'affichage des valeurs.

Spécifier la Largeur et l'Alignement

Vous pouvez définir la largeur et l'alignement des valeurs affichées :

```
fn main() {  
    println!("{:>10}", "hello"); // Aligne "hello" à droite sur 10  
    caractères  
    println!("{:<10}", "world"); // Aligne "world" à gauche sur 10  
    caractères  
}
```



print et nombres

Précision et Format Numérique

Pour les nombres, vous pouvez spécifier des options de précision et de format :

```
fn main() {  
    let number = 42.56789;  
    println!("{:.2}", number); // Affiche avec 2 décimales : "42.57"  
    println!("{:X}", 255);     // Affiche en hexadécimal : "FF"  
}
```



Macro !

Le point d'exclamation ! après println! indique que c'est une macro et non une fonction classique en Rust. Les macros se distinguent des fonctions par cet usage du point d'exclamation.

Les macros offrent plusieurs avantages par rapport aux fonctions ordinaires :

- 1. **Interpolation de texte**: Les macros permettent une interpolation de texte dynamique avec des accolades {}, ce qui rend println! flexible et capable de prendre un nombre variable d'arguments. Rust substitue automatiquement les valeurs fournies dans les emplacements {} à l'intérieur du texte.*
- 2. **Évaluation différée** : Avec une macro, Rust peut optimiser le code en évitant de calculer certaines expressions si elles ne sont pas nécessaires. Par exemple, println! ne fait de calculs de formatage que si l'affichage est effectivement requis, ce qui améliore les performances dans certains cas.*
- 3. **Contrôle sur la syntaxe** : Les macros peuvent manipuler le code source de façon flexible, permettant d'ajouter ou de supprimer du code au moment de la compilation. Par exemple, vec! permet de créer facilement des vecteurs avec des éléments initiaux sans devoir spécifier le type exact.*



Macro !

Différence entre une Macro et une Fonction

Contrairement à une fonction, une macro en Rust :

- *Peut accepter un nombre variable d'arguments, sans nécessiter une définition spécifique de paramètres.*
- *S'exécute avant la compilation, car elle génère du code à partir du code source fourni.*
- *Peut inclure des contrôles supplémentaires qui ne sont pas possibles dans une fonction ordinaire, comme l'utilisation de la syntaxe {} pour le formatage dynamique.*

Si on a bien compris:

println! est une macro, car elle a besoin de fonctionnalités de génération de code flexibles qui ne sont pas réalisables avec une fonction ordinaire. Le point d'exclamation ! permet ainsi de reconnaître facilement cette distinction.



Les commentaires

les commentaires sont des notes ou des explications écrites dans le code, mais ignorées lors de la compilation. Ils servent à documenter le code pour faciliter sa compréhension.

Commentaires Simples

Un commentaire simple commence par `//` et s'étend jusqu'à la fin de la ligne. Ces commentaires sont utilisés pour annoter le code sans générer de documentation.

```
fn main() {  
    // Ceci est un commentaire simple  
    let x = 5; // Initialisation de la variable x  
    println!("La valeur de x est : {}", x); // Affiche x  
}
```



Les commentaires

Commentaires sur Plusieurs Lignes

Les commentaires multilignes commencent par `/*` et se terminent par `*/`. Ils peuvent être utilisés pour commenter de larges blocs de code ou des explications détaillées.

Vous pouvez également imbriquer des commentaires `/* ... */`, ce qui est utile si vous avez besoin de désactiver un bloc de code contenant lui-même des commentaires.

```
fn main() {  
    /*  
        Ceci est un commentaire  
        sur plusieurs lignes.  
        Utilisé pour des explications détaillées.  
    */  
    let y = 10;  
    println!("La valeur de y est : {}", y);  
}
```



Les commentaires

Commentaires de Documentation

Rust utilise des commentaires de documentation pour générer une documentation lisible et accessible via cargo doc. Ces commentaires commencent par `///` pour documenter une fonction, une structure ou un module.

- Les commentaires de documentation sont utilisés par cargo doc pour générer des fichiers HTML détaillant les modules, les fonctions et leurs descriptions.
- Les sections spéciales comme `# Arguments`, `# Retour`, etc., ajoutent des détails utiles pour le lecteur.

```
/// Cette fonction additionne deux nombres
///
/// # Arguments
///
/// * `a` - Le premier nombre
/// * `b` - Le second nombre
///
/// # Retour
///
/// La somme de `a` et `b`.
fn addition(a: i32, b: i32) -> i32 {
    a + b
}
```



Les commentaires

Commentaires de Documentation au Niveau du Module

Les commentaires de documentation peuvent également être utilisés pour documenter des modules entiers. Pour cela, on utilise `//!` en haut du fichier source.

Ces commentaires apparaissent en haut de la documentation du module lorsque vous générez la documentation avec cargo doc.

```
//! Ce module contient les fonctions arithmétiques de base  
//! comme l'addition et la soustraction.
```

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



Variables et mutabilité

En Rust, la gestion des variables fonctionne différemment de nombreux langages de programmation. La compréhension de la mutabilité permet de mieux comprendre comment ce langage favorise la sécurité et l'efficacité dans la gestion de la mémoire et des données.

Par défaut, et donc contrairement à d'autres langages, les variables en Rust sont immutables, ce qui signifie qu'elles ne peuvent pas être modifiées une fois assignées, à moins qu'elles ne soient explicitement déclarées comme mutables.



Variable immuable

Lorsque vous déclarez une variable en Rust avec le mot-clé `let`, elle est, par défaut, immuable.

Cela signifie que sa valeur ne peut pas être modifiée après l'initialisation.

Pourquoi l'immuabilité par défaut ?

- Elle aide à éviter les erreurs liées aux modifications inattendues de données.
- L'immuabilité renforce la sécurité en empêchant les changements accidentels de variables.
- Elle permet au compilateur d'optimiser le code plus efficacement.

```
fn main() {  
    let x = 10; // `x` est immuable  
    println!("La valeur de x est : {}", x);  
  
    // Tentative de modification  
    // x = 20; // Erreur : x est immuable  
}
```



Rendre une variable mutable

Pour permettre la modification d'une variable après sa déclaration, on utilise le mot-clé `mut` lors de la déclaration.

L'ajout de `mut` indique au compilateur que la variable peut être modifiée après l'initialisation.

Cependant, l'utilisation de `mut` doit être intentionnelle, car elle implique un potentiel changement d'état, ce qui peut affecter la fiabilité du code.

```
fn main() {  
    let mut y = 5; // `y` est mutable  
    println!("La valeur initiale de y est : {}", y);  
  
    y = 15; // Modification de la valeur  
    println!("La valeur modifiée de y est : {}", y);  
}
```



Portée et le Masquage des Variables (Shadowing)

Rust propose une autre façon de "changer" une valeur sans utiliser la mutabilité : le masquage de variables (ou shadowing).

Le masquage permet de déclarer une nouvelle variable avec le même nom que la précédente, masquant temporairement l'ancienne version. Cela permet de réassigner des valeurs sans rendre la variable mutable.

Le masquage est souvent utilisé pour :

- Modifier une valeur sans rendre la variable mutable.
- Changer de type : par exemple, si vous avez une chaîne de caractères que vous voulez transformer en nombre, vous pouvez utiliser `let` pour "masquer" la variable avec un nouveau type.

```
fn main() {  
    let x = 5;  
    println!("x initial est : {}", x);  
  
    let x = x + 10; // Masquage de x avec une nouvelle valeur  
    println!("x après masquage est : {}", x);  
  
    {  
        let x = x * 2; // Nouveau masquage dans une portée interne  
        println!("x dans la portée interne est : {}", x);  
    }  
  
    println!("x dans la portée externe est toujours : {}", x);  
}
```



Types et Mutabilité

En Rust, les types eux-mêmes peuvent influencer la mutabilité.

Par exemple, un String peut être immuable en tant que type, mais des méthodes associées permettent de la modifier si la variable est mutable.

```
fn main() {  
    let mut s = String::from("Hello");  
    s.push_str(", world!"); // Ajout à la chaîne de caractères  
    println!("{}", s); // Affiche : Hello, world!  
}
```



Variables et mutabilité

Conventions de Codage pour la Mutabilité

- Utilisez la mutabilité (mut) uniquement quand cela est nécessaire.
- Préférez l'immuabilité pour les variables qui ne nécessitent pas de modifications.
- Utilisez le shadowing pour "changer" une valeur sans rendre la variable mutable, en particulier pour effectuer des transformations de types ou des calculs temporaires.

Différence entre Mutabilité et Masquage:

Aspect	Mutabilité	Masquage
Syntaxe	let mut variable = valeur;	let variable = nouvelle_valeur;
But	Permet de modifier la valeur	Permet de réassigner une nouvelle valeur
Changement de type	Non autorisé	Autorisé (on peut passer de i32 à String)
Portée	Affecte la même variable	Crée une nouvelle instance de la variable



Variables et mutabilité

En Résumé

- **Immutabilité par défaut** : Les variables sont immuables, favorisant la sécurité et l'optimisation du code.
- **Mutabilité intentionnelle** : Utilisez `mut` pour permettre les modifications si nécessaire.
- **Shadowing** : Permet de réutiliser un nom de variable sans la rendre mutable, et même de changer son type.

En combinant ces techniques, Rust permet un équilibre entre sécurité, flexibilité et performance, tout en encourageant de bonnes pratiques de codage.



Les types de données

Rust est un langage fortement typé, ce qui signifie que chaque valeur a un type clairement défini et que le compilateur vérifie ces types à la compilation. Cela permet de savoir comment pouvoir interagir avec ce type de données. Les types sont divisés en deux grandes catégories :

- les types scalaires (ne représentent qu'une seule valeur)
- les types composés. (sont composés de plusieurs valeurs)

A noter que Rust est un langage à typage statique, ce qui signifie que l'on doit connaître le type de toutes les variables lors de la compilation.

Le compilateur peut généralement inférer (deviner) de quel type il s'agit si vous ne l'avez pas spécifié explicitement.



Les numériques entiers

Rust prend en charge les nombres entiers signés ou non-signés selon leurs taille:

Nombre signé: Peut -être négatif

- **i8: 8 bits**
 - de -128 à 127
- **i16: 16 bits**
 - -32,768 à 32,767
- **i32: 32 bits**
 - de 2,147,483,648 à 2,147,483,647
- **i64: 64 bits**
 - de -9,223,372,036,854,775,808 à 9,223,372,036,854,775,807
- **i128: 128 bits**
 - de -170,141,183,460,469,231,731,687,303,715,884,105,728 à 170,141,183,460,469,231,731,687,303,715,884,105,727

Nombre non signé: Ne peut pas être négatif

- **u8: 8 bits**
 - de 0 à 255
- **u16: 16 bits**
 - de 0 à 65,535
- **u32: 32 bits**
 - de 0 à 4,294,967,295
- **u64: 64 bits**
 - de 0 à 18,446,744,073,709,551,615
- **u128: 128 bits**
 - de 0 à 340,282,366,920,938,463,463,374,607,431,768,211,455



Les numériques flottants

Les nombres flottants (f32 et f64) sont représentés en utilisant la norme IEEE 754.

Contrairement aux entiers, ils présentent des limitations en termes de précision et de représentation exacte de certaines valeurs.

- f32 (32 bits): Précision approximative de 7 chiffres
- f64 (64 bits): Précision approximative de 15 chiffres

Limite de Précision : Les flottants ne peuvent pas représenter toutes les valeurs exactement, en particulier celles qui ne sont pas des puissances de deux. Par exemple, des valeurs comme 0.1 et 0.3 ne peuvent pas être représentées avec précision.

NaN et Infinis : Les types f32 et f64 gèrent aussi des valeurs spéciales, comme :

- NaN (Not a Number), représentant des résultats d'opérations indéfinies (comme $0.0 / 0.0$).
- INFINITY et -INFINITY, pour les résultats hors limite (comme $1.0 / 0.0$).



Booléen

En Rust, le type booléen, ou bool, est un type de données simple qui ne peut avoir que deux valeurs : true ou false. Il est souvent utilisé pour représenter des valeurs de vérité ou pour contrôler des structures conditionnelles (comme if ou while).

Pour déclarer une variable de type bool, il suffit de lui assigner les valeurs true ou false.

```
fn main() {  
    let est_vrai: bool = true; // Déclaration explicite  
    let est_faux = false;      // Inférence de type  
    println!("est_vrai : {}, est_faux : {}", est_vrai, est_faux);  
}
```



Le Char

Le type char représente un caractère Unicode et occupe 4 octets (32 bits) en mémoire. Ce type est utilisé pour stocker non seulement les caractères ASCII (comme les lettres et les chiffres) mais également tout caractère Unicode, y compris les accents, les symboles, et même les emojis.

Caractéristiques du Type char

1. **Taille de 4 octets** : Contrairement à d'autres langages où les caractères sont souvent représentés par un seul octet (u8), un char en Rust utilise 4 octets pour garantir qu'il puisse stocker n'importe quel caractère Unicode.
2. **Support d'Unicode** : Le type char peut représenter tout caractère Unicode, ce qui inclut une large variété de langues, de symboles, et d'icônes.

Les caractères sont définis avec des guillemets simples (')

```
fn main() {  
    let lettre = 'A';  
    let chiffre = '7';  
    let symbole = '#';  
    let emoji = '😄';  
  
    println!("Lettre: {}, Chiffre: {}, Symbole: {}, Emoji: {}",  
        lettre, chiffre, symbole, emoji);  
}
```



Exemples d'Utilisation du char

Le type char est utile pour manipuler et tester des caractères individuels dans une chaîne ou pour utiliser des symboles particuliers dans le code.

Chaque caractère char en Rust est un point de code Unicode, ce qui signifie que vous pouvez spécifier des caractères avec leurs valeurs Unicode.

Dans cet exemple, `\u{2665}` représente le caractère cœur (♥), défini par sa valeur Unicode.

```
fn main() {  
    let cœur = '\u{2665}'; // Caractère Unicode pour le  
    symbole de cœur  
    println!("Symbole du cœur : {}", cœur);  
}
```



Exemples d'Utilisation du char

Rust propose diverses méthodes pour travailler avec des char. Voici quelques exemples pratiques :

- Tester si un caractère est numérique : `is_numeric()`
- Tester si un caractère est alphabétique : `is_alphabetic()`
- Transformer en majuscule ou en minuscule : `to_uppercase()` ou `to_lowercase()`

```
fn main() {  
    let c = 'a';  
    println!("{}", c, c.is_alphabetic()); // true  
    println!("{}", c, c.is_numeric());    //  
false  
  
    let majuscule = c.to_uppercase();  
    println!("{}", c, majuscule); // 'A'  
}
```



&str

&str (littéral de chaîne ou string slice) est un type qui représente une séquence de caractères en lecture seule, ou une référence immuable à une chaîne de caractères.

Il pointe vers une chaîne stockée quelque part en mémoire, généralement une chaîne littérale incluse directement dans le code ou une portion d'une String.

Caractéristiques de &str

1. **Référence Immuable** : ce qui signifie que les données ne peuvent pas être modifiées.
2. **Durée de Vie Fixe** : Les littéraux de chaîne sont créés à la compilation, et leur durée de vie est gérée par le compilateur, ce qui garantit la sécurité et prévient les erreurs de gestion mémoire.
3. **Taille Fixe** : Un &str a une taille fixe en mémoire puisqu'il pointe vers une séquence de caractères de longueur définie.

Déclaration et Utilisation de &str

Les chaînes de caractères en &str sont créées avec des guillemets doubles ("):



```
fn main() {  
    let c = 'a';  
    println!("{}", c, c.is_alphabetic()); // true  
    println!("{}", c, c.is_numeric());    //  
false  
  
    let majuscule = c.to_uppercase();  
    println!("En majuscule : {:?}", majuscule); // 'A'  
}
```

&str

Utilisation dans les Fonctions

En Rust, il est courant de passer des chaînes en &str en tant que paramètres, car cela permet de transmettre des chaînes sans transférer la propriété (ownership), rendant le code plus efficace en mémoire.

```
fn saluer(nom: &str) {  
    println!("Bonjour, {} !", nom);  
}  
  
fn main() {  
    let nom = "Alice";  
    saluer(nom); // Pas besoin de transférer l'ownership  
}
```



&str

Littéraux de Chaîne Multi-lignes

En Rust, il est courant de passer des chaînes en &str en tant que paramètres, car cela permet de transmettre des chaînes sans transférer la propriété (ownership), rendant le code plus efficace en mémoire.

```
fn main() {  
    let texte = "Voici une chaîne \\  
                multi-ligne en Rust.";   
    println!("{}", texte);  
}
```



&str

Manipulation de &str

Les littéraux &str sont souvent utilisés pour traiter le texte de manière efficace :

- Longueur : `.len()` retourne la longueur en octets (et non en caractères).
- Accès par Index : Rust ne permet pas l'indexation directe (`str[0]`), car les caractères Unicode peuvent occuper plus d'un octet.
- Parcours et Troncature : `.chars()` pour parcourir chaque caractère, et `.split()` pour diviser la chaîne en plusieurs parties.
- Recherche : `.contains()` pour vérifier si un sous-texte est présent.

```
fn main() {  
    let texte = "Salut, les codeurs !";  
  
    println!("Longueur en octets : {}", texte.len()); // Longueur  
    en octets  
    println!("Contient 'codeurs' ? : {}",  
    texte.contains("codeurs")); // Recherche  
    println!("Parcours des caractères :");  
  
    for c in texte.chars() {  
        print!("{}", c);  
    }  
}
```



String

Le type String représente une chaîne de caractères dynamique stockée dans la mémoire du tas (heap). Contrairement au type &str, qui est une référence immuable à une séquence de caractères de longueur fixe, String est mutable et permet de manipuler des chaînes de manière flexible et performante.

Caractéristiques de String

1. **Mutable** : Une valeur de type String peut être modifiée (on peut lui ajouter ou lui retirer des caractères).
2. **Taille Dynamique** : La taille d'une String peut être ajustée à tout moment.
3. **Allouée sur le Tas** : Les String sont stockées dans la mémoire du tas (heap), ce qui permet de gérer des chaînes de longueur variable.

Création d'une String

Il existe plusieurs manières de créer une String en Rust :

- À partir d'un littéral &str
- Avec to_string()
- Chaîne vide avec String::new()

```
fn main() {  
    let ma_chaine = String::from("Bonjour, Rust !");  
    println!("{}", ma_chaine);  
  
    let ma_chaine2 = "Bonjour".to_string();  
    println!("{}", ma_chaine2);  
  
    let mut ma_chaine3 = String::new();  
    ma_chaine3.push_str("Bonjour, ");  
    ma_chaine3.push_str("Rust !");  
    println!("{}", ma_chaine3);  
}
```



String

Manipulation des String

Rust propose de nombreuses méthodes pour travailler avec String, notamment pour ajouter, supprimer, et rechercher des sous-chaînes.

Ajouter des Caractères ou du Texte

- `push(char)` : Ajoute un caractère à la fin de la String.
- `push_str(&str)` : Ajoute un `&str` (littéral de chaîne) à la fin de la String.

```
fn main() {  
    let mut texte = String::from("Bonjour");  
    texte.push(' ');           // Ajoute un espace  
    texte.push_str("tout le monde !"); // Ajoute une sous-  
    chaîne  
    println!("{}", texte);  
}
```



String

Concaténer des String et &str

On peut aussi utiliser l'opérateur + pour concaténer des String, bien qu'il attende un String comme premier opérande et un &str comme second :

```
fn main() {  
    let bonjour = String::from("Bonjour, ");  
    let marseille = "Marseille !";  
    let salutation = bonjour + marseille;  
    println!("{}", salutation);  
}
```



String

Formatage de Chaînes

Pour créer des chaînes formatées, on utilise la macro format! :

```
fn main() {  
    let prenom = "Alice";  
    let age = 30;  
    let message = format!("{}", prenom, age);  
    println!("{}", message);  
}
```



String

Indexation et Accès aux Caractères

Rust n'autorise pas l'accès direct aux caractères d'une String via des indices (`str[0]`) car chaque caractère UTF-8 peut occuper plusieurs octets. Pour accéder aux caractères d'une String, on utilise :

- `chars()` pour un itérateur sur chaque caractère
- `bytes()` pour un itérateur sur chaque octet

```
fn main() {  
    let prenom = "Alice";  
    let age = 30;  
    let message = format!("{}", prenom, age);  
    println!("{}", message);  
}
```



String

Méthodes Utiles pour les String

Voici quelques méthodes courantes pour manipuler des String :

- `len()` : Donne la longueur en octets.
- `is_empty()` : Vérifie si la chaîne est vide.
- `replace(&str, &str)` : Remplace toutes les occurrences d'un sous-texte par un autre.
- `contains(&str)` : Vérifie si une sous-chaîne est présente.
- `split(&str)` : Divise la chaîne en plusieurs parties selon un séparateur.

```
fn main() {  
    let texte = String::from("Bonjour le monde !");  
    println!("Longueur : {}", texte.len());  
    println!("Contient 'monde' ? {}",  
        texte.contains("monde"));  
    let remplace = texte.replace("monde", "Rust");  
    println!("Après remplacement : {}", remplace);  
}
```



Conversion entre String et &str

De String à &str

La conversion d'une String en &str se fait avec `.as_str()` ou par simple référence (&).

De &str à String

On peut utiliser `.to_string()` ou `String::from()` pour créer une String à partir d'un &str :

```
fn main() {  
    let ma_chaine = String::from("Bonjour, Rust !");  
    let mon_slice: &str = &ma_chaine;  
    println!("{}", mon_slice);  
}  
  
// Ou  
  
fn main() {  
    let mon_slice = "Salut !";  
    let ma_chaine = mon_slice.to_string();  
    println!("{}", ma_chaine);  
}
```



Tuples

Les tuples sont des types de données qui permettent de regrouper plusieurs valeurs de différents types en une seule unité. Ils sont utiles pour regrouper des données liées sans avoir à créer de struct dédiée.

Caractéristiques des Tuples

1. **Hétérogénéité** : Les tuples peuvent contenir des éléments de types différents. Par exemple, un tuple peut contenir un entier, un flottant, et une chaîne.
2. **Taille Fixe** : La taille d'un tuple est fixée lors de sa création. Contrairement aux tableaux et aux vecteurs, les tuples ne peuvent pas être redimensionnés.
3. **Accès aux Éléments par Index** : Les éléments d'un tuple peuvent être accédés à l'aide de leur position, commençant à l'index 0.

Un tuple est défini en entourant ses éléments par des parenthèses et en séparant chaque élément par une virgule

```
fn main() {  
    let mon_tuple = (42, 3.14, "Rust");  
    println!("Tuple : {:?}", mon_tuple);  
}
```



Tuples

Accéder aux Éléments d'un Tuple

On accède aux éléments d'un tuple en utilisant un point (.) suivi de l'index de l'élément :

```
fn main() {  
    let mon_tuple = (42, 3.14, "Rust");  
  
    let entier = mon_tuple.0;  
    let flottant = mon_tuple.1;  
    let chaine = mon_tuple.2;  
  
    println!("Entier : {}, Flottant : {}, Chaîne : {}", entier,  
            flottant, chaine);  
}
```



Tuples

Destructuration de Tuples

Rust permet de déstructurer un tuple, c'est-à-dire d'extraire ses éléments en une seule ligne, ce qui peut rendre le code plus lisible :

```
fn main() {  
    let mon_tuple = (42, 3.14, "Rust");  
  
    let (entier, flottant, chaine) = mon_tuple; //  
Déstructuration  
    println!("Entier : {}, Flottant : {}, Chaîne : {}", entier,  
flottant, chaine);  
}
```



Tuples

Utilisation des Tuples dans les Fonctions:

Les tuples peuvent être utilisés comme valeurs de retour pour transmettre plusieurs valeurs depuis une fonction. Nous n'avons pas encore vu les fonctions mais nous y arriverons très vite.

```
fn calculs(a: i32, b: i32) -> (i32, i32) {  
    (a + b, a * b) // Retourne un tuple avec la somme et le  
    produit  
}  
  
fn main() {  
    let (somme, produit) = calculs(3, 4);  
    println!("Somme : {}, Produit : {}", somme, produit);  
}
```



Tuples

Les Tuples Vides : ()

Le tuple vide () est appelé le type unité en Rust, et représente l'absence de valeur. Il est souvent utilisé comme valeur de retour par défaut des fonctions qui n'ont pas de return explicite :

```
fn saluer() {  
    println!("Bonjour, Rust !");  
} // `saluer` retourne `()`, le type unité
```



Tuples

Tuples Imbriqués

Les tuples peuvent également être imbriqués dans d'autres tuples, permettant de créer des structures plus complexes :

```
fn main() {  
    let tuple_complexe = ((1, 2), (3, "quatre"), 5.0);  
    println!("Tuple imbriqué : {:?}", tuple_complexe);  
}
```



Les tuples

Exemples d'Utilisation Courante des Tuples

Les tuples sont utiles dans de nombreuses situations, comme pour :

- Retourner plusieurs valeurs à partir d'une fonction.
- Grouper des données de types différents pour une meilleure organisation.
- Fournir des valeurs temporaires ou intermédiaires dans le code sans nécessiter une structure plus formelle.

Conclusion

Les tuples en Rust sont une manière rapide et efficace de regrouper des valeurs de types différents. Leur utilisation pour organiser les données et pour les valeurs de retour de fonctions rend le code plus lisible et mieux structuré.



Array

En Rust, un array (ou tableau) est une collection de valeurs de taille fixe et de type unique, stockées en mémoire de manière contiguë.

Les arrays sont principalement utilisés lorsque la taille de la collection est connue à la compilation et ne change pas durant l'exécution.

Caractéristiques des Arrays en Rust

1. **Taille Fixe** : La taille d'un array est fixée lors de sa création et ne peut pas être modifiée.
2. **Type Unique** : Tous les éléments d'un array doivent être du même type.
3. **Stockage sur la Pile** : Les arrays sont stockés sur la pile (stack), sauf s'ils sont intégrés dans une structure qui nécessite une allocation dynamique.



Array

Déclaration et Initialisation

Un array en Rust est défini en plaçant les éléments entre crochets [], séparés par des virgules.

La syntaxe de déclaration d'un array est [Type; taille], où Type est le type de données des éléments et taille est le nombre d'éléments.

Ici, mon_array est un array de 5 éléments de type i32.

```
fn main() {  
    let mon_array: [i32; 5] = [1, 2, 3, 4, 5];  
    println!("{:?}", mon_array);  
}
```



Array

Initialiser tous les éléments à la même valeur

Pour initialiser tous les éléments d'un array avec la même valeur, utilisez la syntaxe [valeur; taille] :

```
fn main() {  
    let mon_array: [i32; 5] = [1, 2, 3, 4, 5];  
    println!("{:?}", mon_array);  
}
```



Array

Accéder aux Éléments d'un Array

Les éléments d'un array peuvent être accédés en utilisant leur index, commençant à 0. Cependant, Rust vérifie les index et renvoie une erreur si un index invalide est utilisé, évitant ainsi les erreurs de segmentation courantes dans d'autres langages.

```
fn main() {  
    let nombres = [10, 20, 30, 40, 50];  
    let premier = nombres[0];  
    let dernier = nombres[4];  
  
    println!("Premier : {}, Dernier : {}", premier, dernier);  
}
```



Array

Boucler sur les Éléments d'un Array

La boucle for permet d'itérer facilement sur chaque élément d'un array :

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



Array

Taille d'un Array

La méthode `.len()` permet d'obtenir la taille d'un array :

```
fn main() {  
    let nombres = [1, 2, 3, 4, 5];  
    println!("Taille : {}", nombres.len());  
}
```



Array

Arrays et Slices

Un slice est une vue sur une portion d'un array. Un slice ne possède pas de taille fixe, il est donc souvent utilisé pour manipuler une partie d'un array sans transférer la propriété.

Création d'un Slice

On peut obtenir un slice en utilisant l'opérateur de découpage [début..fin] :

```
fn main() {  
    let nombres = [1, 2, 3, 4, 5];  
    let slice = &nombres[1..4]; // Inclut nombres[1] jusqu'à  
    nombres[3]  
    println!("Slice : {:?}" , slice);  
}
```



Array

Arrays Multi-dimensionnels

Les arrays peuvent être imbriqués pour créer des tableaux multi-dimensionnels, utiles pour des matrices ou des grilles.

```
fn main() {  
    let matrice = [  
        [1, 2, 3],  
        [4, 5, 6],  
        [7, 8, 9]  
    ];  
    println!("{:?}", matrice);  
}
```



Les fonctions



Définition

Une fonction en programmation est un bloc de code autonome qui effectue une tâche spécifique et peut être réutilisé plusieurs fois dans un programme. Elle prend en entrée zéro ou plusieurs paramètres (ou arguments), exécute une série d'instructions, et peut renvoyer un résultat ou une valeur.

Caractéristiques d'une Fonction

1. **Modularité** : Elle permet de diviser le code en segments plus petits et spécifiques, ce qui améliore la lisibilité et la maintenance.
2. **Réutilisabilité** : Une fois définie, une fonction peut être appelée (ou exécutée) plusieurs fois avec différents paramètres.
3. **Encapsulation** : Elle cache les détails de son implémentation et ne montre que son interface (nom et paramètres), ce qui simplifie son utilisation.

Exemple Général d'une Fonction

Une fonction peut être imaginée comme une boîte noire qui :

- Reçoit des valeurs d'entrée,
- Effectue un traitement sur ces valeurs,
- Produit et retourne un résultat.

Elles sont fondamentales en programmation, car elles permettent de structurer, simplifier et optimiser le code.



Fonction

Déclaration d'une Fonction

La syntaxe de base pour déclarer une fonction est la suivante :

- `fn` : Le mot-clé pour déclarer une fonction.
- `nom_de_fonction` : Le nom de la fonction, en utilisant la convention `snake_case`.
- `param1: Type` : Les paramètres sont spécifiés avec leur nom et leur type. Rust requiert de toujours indiquer le type des paramètres.
- `-> TypeRetour` : Indique le type de retour de la fonction. Si une fonction ne retourne rien, le type de retour est `()`, le type unité (par défaut si omis).

```
fn nom_de_fonction(param1: Type, param2: Type) -> TypeRetour {  
    // Code de la fonction  
}
```



Fonction

Exemple de Fonction Simple

Ici, ajouter prend deux paramètres x et y de type i32 et retourne leur somme.

Dans Rust, la dernière expression sans point-virgule ; est implicitement retournée.

```
fn ajouter(x: i32, y: i32) -> i32 {  
    x + y // Retourne la somme de x et y  
}  
  
fn main() {  
    let resultat = ajouter(5, 10);  
    println!("Le résultat est : {}", resultat);  
}
```



Fonction de base

Une fonction peut ne pas avoir de valeur de retour, ni de paramètre.

Auquel cas elle renvoie (), le type unité.

Les fonctions qui n'ont pas de type de retour sont souvent appelées « procédures » :

```
fn saluer() {  
    println!("Bonjour !");  
}  
  
fn main() {  
    saluer(); // Appelle la fonction qui n'a pas de retour  
}
```



Fonction avec paramètre

Les paramètres de fonction en programmation sont des valeurs fournies à une fonction lors de son appel pour personnaliser ou adapter son exécution.

En Rust, les paramètres permettent de rendre une fonction plus flexible en lui permettant d'opérer sur différentes valeurs sans que le code de la fonction doive être modifié.

Caractéristiques des Paramètres en Rust

1. Type statique : Rust exige que chaque paramètre ait un type explicite lors de la définition de la fonction. Cela permet au compilateur de vérifier la cohérence des types et d'éviter des erreurs d'exécution.
2. Immutabilité par défaut : Par défaut, les paramètres sont immuables (non modifiables) dans une fonction en Rust. Cela garantit qu'une fonction ne peut pas altérer les valeurs d'entrée, sauf indication contraire avec le mot-clé `mut`.

```
fn nom_de_fonction(param1: Type1, param2: Type2) -> TypeRetour
{
    // Corps de la fonction
}
```



Fonction avec paramètre

Explications

- saluer : Le nom de la fonction.
- nom: &str : Un paramètre nom de type &str, qui est une référence immuable vers une chaîne de caractères.
- Pas de type de retour : La fonction ne renvoie aucune valeur. Elle réalise une action (affiche un message) sans fournir de résultat.

```
fn saluer(nom: &str) {  
    println!("Bonjour, {} !", nom);  
}  
  
fn main() {  
    saluer("Alice");  
}
```



Fonction avec paramètre

Paramètres dans les Closures

En Rust, les closures sont des fonctions anonymes qui peuvent capturer des valeurs de leur environnement. Elles fonctionnent aussi avec des paramètres :

```
fn main() {  
    let multiplier = |x: i32| x * 2;  
    println!("5 * 2 = {}", multiplier(5));  
}
```



Paramètres et Immutabilité

Par défaut, les paramètres sont immuables dans Rust. Si vous souhaitez modifier un paramètre à l'intérieur d'une fonction, vous devez utiliser le mot-clé `mut`.

Toutefois, une pratique courante consiste à modifier une copie locale des données :

```
fn doubler(mut nombre: i32) -> i32 {  
    nombre *= 2;  
    nombre  
}  
  
fn main() {  
    let nombre = 5;  
    let resultat = doubler(nombre);  
    println!("Le double de {} est {}", nombre, resultat);  
}
```



Fonctions avec retour

Les fonctions peuvent retourner des valeurs explicites avec l'instruction `return`, bien que Rust permette également d'omettre `return` en plaçant la valeur de retour comme dernière expression (sans point-virgule) :

```
fn maximum(a: i32, b: i32) -> i32 {  
    if a > b {  
        return a; // Utilisation de `return`  
    }  
    b // Dernière expression, retournée implicitement  
}  
  
fn main() {  
    let max = maximum(3, 7);  
    println!("Le maximum est : {}", max);  
}
```



Fonctions avec des Références et Gestion de la Propriété

En Rust, pour éviter de transférer la propriété (ownership) des données, on utilise des références (&) comme paramètres, ce qui rend la fonction plus flexible :

```
fn longueur(texte: &str) -> usize {
    texte.len()
}

fn main() {
    let phrase = String::from("Rust est rapide !");
    let longueur_phrase = longueur(&phrase); // Passage par
référence
    println!("La longueur de la phrase est : {}", longueur_phrase);
}
```



Fonctions avec des Fonctions Anonymes (Closures)

Rust prend en charge les fonctions anonymes ou closures, qui sont des fonctions sans nom créées directement dans le code. Elles permettent de capturer leur environnement (les variables locales).

```
fn main() {  
    let multiplier = |x: i32, y: i32| x * y;  
    println!("3 * 4 = {}", multiplier(3, 4));  
}
```



Exemples de Fonction avec Différents Types de Retours

Fonction avec Option<T>

Rust utilise souvent le type Option<T> pour signaler l'absence de valeur, ce qui rend le code plus sûr. Par exemple :

```
fn division(a: i32, b: i32) -> Option<i32> {  
    if b == 0 {  
        None  
    } else {  
        Some(a / b)  
    }  
}  
  
fn main() {  
    match division(10, 2) {  
        Some(resultat) => println!("Résultat : {}", resultat),  
        None => println!("Erreur : division par zéro"),  
    }  
}
```



Exemples de Fonction avec Différents Types de Retours

Fonction avec Result<T, E>

De même, Result<T, E> est utilisé pour gérer les erreurs :

```
fn ouvrir_fichier(nom: &str) -> Result<(), String> {  
    if nom.is_empty() {  
        Err(String::from("Nom de fichier vide"))  
    } else {  
        println!("Ouverture du fichier {}", nom);  
        Ok(())  
    }  
}  
  
fn main() {  
    match ouvrir_fichier("data.txt") {  
        Ok(_) => println!("Fichier ouvert avec succès"),  
        Err(e) => println!("Erreur : {}", e),  
    }  
}
```



Les types génériques



Définition

En Rust, les types génériques Option et Result sont utilisés pour gérer les situations où une valeur peut être absente ou une opération peut échouer.

Ces types permettent de gérer de manière sécurisée les erreurs et les valeurs nulles, en éliminant les risques associés aux null et aux exceptions non gérées.



Option<T>

Le type Option<T> représente une valeur qui peut être présente (Some) ou absente (None).

Utilisation de Option<T>:

Option est utile lorsque l'on travaille avec des valeurs qui peuvent être nulles ou non définies. Par exemple, lorsque l'on cherche un élément dans une collection ou une base de données, le résultat peut être soit l'élément recherché (Some), soit l'absence d'élément (None).

Exemple de Fonction Utilisant Option

Voici une fonction qui retourne la racine carrée d'un nombre s'il est positif et retourne None si le nombre est négatif.

rust

Dans cet exemple :

- Some(x.sqrt()) est retourné si la racine carrée existe.
- None est retourné si le nombre est négatif.

```
fn racine_carre(x: f64) -> Option<f64> {  
    if x >= 0.0 {  
        Some(x.sqrt()) // Retourne la racine carrée  
    } else {  
        None // Retourne None pour les nombres négatifs  
    }  
}  
  
fn main() {  
    let nombre = -4.0;  
    match racine_carre(nombre) {  
        Some(racine) => println!("La racine carrée de {} est {}",  
nombre, racine),  
        None => println!("Pas de racine carrée pour les nombres  
négatifs."),  
    }  
}
```



Result<T, E>

Le type Result<T, E> est utilisé pour les opérations pouvant échouer. Il est défini de manière à fournir une valeur (Ok) ou une erreur (Err), et permet ainsi de gérer les erreurs de manière explicite:

- Ok(T) contient la valeur de retour de l'opération réussie.
- Err(E) contient des informations sur l'erreur.

Exemple de Fonction Utilisant Result

Voici une fonction qui effectue une division. Elle retourne Ok avec le résultat si la division est valide, ou Err avec un message d'erreur en cas de division par zéro.

Dans cet exemple :

- Ok(a / b) est retourné si la division est possible.
- Err(String::from("Erreur : division par zéro")) est retourné si le diviseur est zéro, ce qui permet de gérer cette erreur explicitement.

```
fn division(a: f64, b: f64) -> Result<f64, String> {
    if b == 0.0 {
        Err(String::from("Erreur : division par zéro"))
    } else {
        Ok(a / b)
    }
}

fn main() {
    let resultat = division(10.0, 0.0);
    match resultat {
        Ok(valeur) => println!("Résultat de la division : {}", valeur),
        Err(message) => println!("{}", message),
    }
}
```



Avantages

Avantages d'Option et Result

1. Sécurité : Ils éliminent les références nulles (null) et réduisent les erreurs de segmentation.
2. Contrôle explicite des erreurs : Result encourage à vérifier chaque cas d'erreur et à traiter les erreurs de manière explicite.
3. Meilleure lisibilité : Le code est plus clair et rend les intentions explicites.



Manipulation

Rust propose des méthodes pour simplifier le travail avec ces types.

Méthodes courantes pour Option

- `.unwrap()` : Extrait la valeur `Some` ou panique si c'est `None`.
- `.unwrap_or(default)` : Extrait la valeur `Some` ou utilise `default` si c'est `None`.
- `.map()` : Applique une fonction à la valeur `Some` sans affecter `None`.

```
fn main() {  
    let valeur: Option<i32> = Some(5);  
    let valeur_double = valeur.map(|x| x * 2); // Double la valeur  
    // si elle est présente  
    println!("{:?}", valeur_double.unwrap_or(0)); // Affiche 10 ou 0  
    // si None  
}
```



Méthodes

- `.unwrap()` : Extrait la valeur Ok ou panique si c'est Err.
- `.unwrap_or(default)` : Extrait la valeur Ok ou utilise default si c'est Err.
- `.map()` et `.map_err()` : Appliquent une fonction sur Ok ou Err respectivement.

```
fn main() {  
    let resultat: Result<i32, &str> = Ok(10);  
    let double = resultat.map(|x| x * 2); // Double la valeur si Ok  
    println!("{:?}", double.unwrap_or(0)); // Affiche 20 ou 0 si Err  
}
```



Les opérateurs



Arithmétiques

Les opérateurs arithmétiques sont utilisés pour les opérations mathématiques de base. Ils fonctionnent avec les types numériques (i32, f64, etc.).

- +
 - Addition
 - $a + b$
- -
 - Soustraction
 - $a - b$
- *
 - Multiplication
 - $a * b$
- /
 - Division
 - a / b
- %
 - Modulo (reste de division)
 - $a \% b$

```
fn main() {  
    let a = 10;  
    let b = 3;  
    println!("Addition : {}", a + b);    // 13  
    println!("Modulo : {}", a % b);    // 1  
}
```



Comparaison

Les opérateurs de comparaison retournent un booléen (true ou false) en fonction du résultat de la comparaison. Ils sont utiles dans les structures de contrôle (if, while, etc.).

- ==
 - Égalité
 - `a == b`
- !=
 - Inégalité
 - `a != b`
- >
 - Supérieur à
 - `a > b`
- <
 - Inférieur à
 - `a < b`
- >=
 - Supérieur ou égal à
 - `a >= b`
- <=
 - Inférieur ou égal à
 - `a <= b`



```
fn main() {  
    let x = 5;  
    let y = 10;  
    println!("x est-il inférieur à y ? {}", x < y); // true  
}
```

Logique

Les opérateurs logiques permettent de combiner plusieurs conditions booléennes.

- &&
 - ET logique
 - $(a < b) \ \&\& \ (b < c)$
- ||
 - OU logique
 - $(a < b) \ || \ (b < c)$
 -
- !
 - NON logique
 - !a

```
fn main() {  
    let x = true;  
    let y = false;  
    println!("x ET y : {}", x && y);    // false  
    println!("x OU y : {}", x || y);    // true  
    println!("NON x : {}", !x);         // false  
}
```



Assignment

Les opérateurs d'assignation permettent d'affecter des valeurs à des variables, avec des variantes combinant assignation et opération.

- =
 - Assignation
 - $a = b$
- +=
 - Addition et assignation
 - $a += b$
- -=
 - Soustraction et assignation
 - $a -= b$
- *=
 - Multiplication et assignation
 - $a *= b$
- /=
 - Division et assignation
 - $a /= b$
- %=
 - Modulo et assignation
 - $a \% = b$



```
fn main() {  
    let mut x = 10;  
    x += 5; // équivaut à x = x + 5  
    println!("x après addition : {}", x); // 15  
}
```

Manipulation de Bits

Les opérateurs de bits sont utilisés pour manipuler directement les bits des valeurs numériques, souvent dans les applications bas niveau ou pour des optimisations spécifiques.

- `&`
 - ET bit à bit
 - `a & b`
- `|`
 - OU
 - OU bit à bit
- `^`
 - OU exclusif (XOR) bit à bit
 - `a ^ b`
- `<<`
 - Décalage à gauche
 - `a << 1`
- `>>`
 - Décalage à droite
 - `a >> 1`

```
fn main() {  
    let mut x = 10;  
    x += 5; // équivaut à x = x + 5  
fn main() {  
    let a = 0b1010; // 10 en binaire  
    let b = 0b1100; // 12 en binaire  
    println!("a ET b : {:04b}", a & b); // 1000 (8)  
    println!("a OU b : {:04b}", a | b); // 1110 (14)  
    println!("a XOR b : {:04b}", a ^ b); // 0110 (6)  
}  
  
    println!("x après addition : {}", x); // 15  
}
```



Déréférencement et de Référence

En Rust, les opérateurs de référence `&` et de déréférencement `*` sont utilisés pour travailler avec des pointeurs et des références.

- `&` : Crée une référence vers une valeur.
- `*` : Déréférence une référence pour accéder à la valeur sous-jacente.

```
fn main() {  
    let x = 5;  
    let ref_x = &x; // Référence de x  
    println!("Valeur de x via ref_x : {}", *ref_x); // Déréférencement  
    // pour accéder à x  
}
```



Portée

L'opérateur de portée `::` est utilisé pour accéder aux éléments d'un module ou d'un type, comme les constantes ou les fonctions associées.

```
mod math {  
    pub fn ajouter(a: i32, b: i32) -> i32 {  
        a + b  
    }  
}  
  
fn main() {  
    let somme = math::ajouter(5, 10); // Utilisation de `::` pour  
    accéder à `ajouter`  
    println!("La somme est : {}", somme);  
}
```



Conversion

Rust permet des conversions de type explicites via les opérateurs `as`.

```
fn main() {  
    let nombre_f64 = 10.7_f64;  
    let nombre_entier = nombre_f64 as i32; // Conversion explicite  
    en entier  
    println!("Nombre converti : {}", nombre_entier); // 10  
}
```



Le contrôle de flux



Introduction au contrôle de flux

Le contrôle de flux en programmation désigne les structures et instructions qui orientent la séquence d'exécution d'un programme. Il permet de définir comment et quand les différentes parties d'un programme doivent être exécutées, de répondre aux conditions spécifiques, ou encore de répéter certaines opérations de manière contrôlée.

Rust propose un ensemble d'outils puissants pour le contrôle de flux, notamment:

- les conditions if et match,
- les boucles loop, while, et for
- les structures de gestion d'erreurs avec Option et Result.

Grâce à la syntaxe expressive de Rust et à sa gestion explicite des valeurs et des erreurs, le langage encourage un code plus sécurisé et lisible, en forçant le développeur à gérer les cas d'erreur potentiels dès la phase de développement.



Introduction au contrôle de flux

Par exemple:

- l'instruction conditionnelle `if` permet d'exécuter un bloc de code uniquement si une certaine condition est remplie, tandis que `match` offre une approche puissante pour gérer des situations où une valeur peut correspondre à plusieurs cas possibles.
- Les boucles permettent quant à elles de répéter des instructions jusqu'à ce qu'une condition soit atteinte, avec des contrôles précis grâce aux instructions `break` et `continue`.

Dans un contexte de gestion d'erreurs, Rust se distingue par l'utilisation explicite de `Option` et `Result`, des types qui encouragent un contrôle de flux robuste et sécurisé.

- `Option` gère les valeurs qui peuvent être absentes
- `Result` gère les erreurs potentielles dans une opération, offrant une alternative claire aux exceptions traditionnelles en programmation.

En combinant ces types avec les structures de contrôle de flux, Rust aide à construire des programmes résistants aux erreurs, où chaque situation d'exécution est traitée explicitement.

Le contrôle de flux en Rust permet ainsi de structurer le code de façon logique et sécurisée, garantissant une gestion claire et prévisible des conditions et des erreurs possibles, pour un programme plus fiable et maintenable.



La condition if

L'instruction if évalue une condition et exécute le bloc de code associé si la condition est vraie.

En Rust, la condition doit toujours être de type booléen (true ou false). Rust ne permet pas l'utilisation implicite d'un nombre ou d'une autre valeur comme condition, ce qui garantit une plus grande sécurité et lisibilité.

```
if condition {  
    // Code exécuté si la condition est vraie  
}
```

```
fn main() {  
    let age = 18;  
  
    if age >= 18 {  
        println!("Vous êtes majeur !");  
    }  
}
```



La condition if .. else if .. else

Les instructions else if et else permettent d'ajouter des conditions supplémentaires et des alternatives dans le cas où aucune des conditions précédentes n'est remplie.

- else if : Ajoute une condition supplémentaire qui est vérifiée si les conditions précédentes sont fausses.
- else : Permet de définir un bloc de code par défaut qui est exécuté si aucune des conditions précédentes n'est vraie.

```
if condition1 {  
    // Code si condition1 est vraie  
} else if condition2 {  
    // Code si condition1 est fausse et condition2 est vraie  
} else {  
    // Code si toutes les conditions précédentes sont fausses  
}
```

```
fn main() {  
    let temperature = 25;  
  
    if temperature > 30 {  
        println!("Il fait chaud !");  
    } else if temperature < 10 {  
        println!("Il fait froid !");  
    } else {  
        println!("Le temps est agréable.");  
    }  
}
```



if en expression

En Rust, les instructions if peuvent également être utilisées comme expressions, permettant d'assigner une valeur à une variable en fonction d'une condition. Cela évite l'utilisation de branches conditionnelles redondantes et améliore la lisibilité.

Exemple d'if comme Expression

```
fn main() {  
    let nombre = 7;  
    let est_pair = if nombre % 2 == 0 { "pair" } else { "impair" };  
;  
  
    println!("Le nombre est {}", est_pair);  
}
```

Ici, la variable `est_pair` prend la valeur "pair" si `nombre` est pair, sinon elle prend la valeur "impair".

```
fn main() {  
    let nombre = 7;  
    let est_pair = if nombre % 2 == 0 { "pair" } else { "impair" };  
  
    println!("Le nombre est {}", est_pair);  
}
```



L'instruction if let

if let est une variante de if qui simplifie la vérification et l'extraction de valeurs dans certaines structures de données, notamment les énumérations comme Option et Result. Elle est particulièrement utile pour tester une seule variante d'une énumération tout en ignorant les autres.

Exemple avec Option

Dans cet exemple :

- Si valeur contient Some(10), alors x prend la valeur 10 et est affiché.
- Si valeur est None, le programme affiche "Pas de valeur."

ex qui montrerait None:

```
fn main() {  
    let valeur: Option<i32> = None;  
    if let Some(x) = valeur {  
        println!("La valeur est : {}", x);  
    } else {  
        println!("Pas de valeur.");  
    }  
}
```



```
fn main() {  
    let valeur = Some(10);  
  
    if let Some(x) = valeur {  
        println!("La valeur est : {}", x);  
    } else {  
        println!("Pas de valeur.");  
    }  
}
```

La boucle loop

La boucle loop est une boucle infinie qui continue à exécuter son bloc de code jusqu'à ce qu'une instruction break soit rencontrée. Elle est utile lorsqu'on ne connaît pas à l'avance le nombre d'itérations nécessaires.

Syntaxe

```
loop {  
  // Code à répéter indéfinimentif condition_de_sortie {  
    break; // Sort de la boucle si la condition est vraie  
}
```

```
fn main() {  
  let mut compteur = 0;  
  
  loop {  
    println!("Compteur : {}", compteur);  
    compteur += 1;  
  
    if compteur == 5 {  
      break; // Sortie de la boucle quand compteur atteint 5  
    }  
  }  
}
```



La boucle while

La boucle while exécute un bloc de code tant qu'une condition est vraie. Elle est utile quand on veut répéter une opération, mais seulement sous une certaine condition.

Syntaxe

```
while condition {  
    // Code exécuté tant que la condition est vraie  
}
```

```
fn main() {  
    let mut compteur = 0;  
  
    loop {  
        println!("Compteur : {}", compteur);  
        compteur += 1;  
  
        if compteur == 5 {  
            break; // Sortie de la boucle quand compteur atteint 5  
        }  
    }  
}
```



La boucle for

La boucle for est souvent utilisée pour itérer sur des collections (comme les tableaux ou vecteurs) ou sur des plages de valeurs. Elle est concise et permet d'éviter les erreurs courantes comme les débordements d'indices.

Syntaxe

```
for element in collection {  
    // Code exécuté pour chaque élément de la collection  
}
```

```
fn main() {  
    let nombres = [10, 20, 30, 40, 50];  
  
    for nombre in nombres.iter() {  
        println!("Nombre : {}", nombre);  
    }  
  
    // Boucle sur une plage de valeurs  
    for i in 0..5 {  
        println!("i : {}", i);  
    }  
}
```



break et continue

Rust propose deux instructions pour contrôler le flux au sein des boucles :

- `break` : Interrompt la boucle en cours, sortant immédiatement de celle-ci.
- `continue` : Ignore le reste du code de l'itération en cours et passe à l'itération suivante.

Ici, le programme affiche uniquement les nombres impairs de 1 à 7, mais sort de la boucle lorsque `i` atteint 7.

```
fn main() {  
    for i in 1..10 {  
        if i % 2 == 0 {  
            continue; // Ignore les nombres pairs  
        }  
        if i == 7 {  
            break; // Sort de la boucle quand i atteint 7  
        }  
        println!("i : {}", i);  
    }  
}
```



boucle avec valeur de retour

En Rust, une boucle loop peut renvoyer une valeur. Pour cela, on peut utiliser l'instruction break en passant une valeur avec break.

Ici, la boucle loop se termine lorsque compteur atteint 10, et la valeur de compteur * 2 est renvoyée pour être stockée dans resultat.

```
fn main() {  
    let mut compteur = 0;  
  
    let resultat = loop {  
        compteur += 1;  
  
        if compteur == 10 {  
            break compteur * 2; // La boucle renvoie la valeur de  
            `compteur * 2`  
        }  
    };  
  
    println!("Résultat : {}", resultat); // Affiche : "Résultat : 20"  
}
```



match

Match est une structure de contrôle qui permet de prendre des décisions en fonction de la valeur d'une expression. Elle est similaire aux instructions switch d'autres langages. La structure match est souvent utilisée pour travailler avec des énumérations, comme Option et Result, et pour effectuer un contrôle de flux précis et lisible.

match expression {

```
pattern1 => { /* code pour pattern1 */ },  
pattern2 => { /* code pour pattern2 */ },  
_ => { /* code pour tous les autres cas */ },  
}
```

Chaque branche de match contient un pattern (motif) qui, si la valeur de expression correspond à ce motif, exécute le bloc de code associé.

Le motif _ agit comme un joker pour capturer tous les cas non explicitement traités, garantissant qu'il n'y a pas de cas non couverts.

```
fn main() {  
    let nombre = 3;  
  
    match nombre {  
        1 => println!("Un"),  
        2 => println!("Deux"),  
        3 => println!("Trois"),  
        _ => println!("Autre nombre"), // Le caractère '_' attrape tous  
        les autres cas  
    }  
}
```



Gestion de mémoire et sécurité



Introduction

En Rust, les concepts de propriété (ownership), emprunt (borrowing), et durée de vie (lifetimes) sont essentiels pour garantir la sécurité de la mémoire sans utiliser de ramasse-miettes (garbage collector). I

Is permettent à Rust de gérer efficacement la mémoire et d'éviter les erreurs courantes comme les références invalides et les fuites de mémoire.



Ownership

Le système de propriété (ownership) de Rust est l'un des concepts les plus fondamentaux du langage, conçu pour gérer la mémoire de manière sûre et efficace.

Grâce à ce système, Rust garantit qu'il n'y aura pas de fuites de mémoire ni de problèmes de sécurité liés aux références non valides. Cela signifie aussi que Rust peut offrir des performances proches du C et du C++ sans nécessiter de ramasse-miettes (garbage collector).

Les Trois Règles de la Propriété

1. **Chaque valeur a un propriétaire** : En Rust, chaque valeur (comme une chaîne de caractères ou un entier) est possédée par une variable spécifique qui en devient l'unique propriétaire.
2. **Une valeur ne peut avoir qu'un seul propriétaire à la fois** : Si la propriété d'une valeur est transférée d'une variable à une autre, la variable d'origine perd son droit d'utiliser cette valeur.
3. **Lorsque le propriétaire sort du champ de validité (scope), la valeur est libérée** : Lorsque la variable propriétaire d'une valeur sort de son champ de validité, Rust libère automatiquement la mémoire associée à cette valeur.

Ces règles permettent à Rust de contrôler qui a accès à quelle donnée, empêchant ainsi les accès non sécurisés et optimisant la gestion de la mémoire.



Base de propriété

Dans cet exemple :

- La variable s1 est le propriétaire de la chaîne "Bonjour".
- Lorsqu'on assigne s1 à s2, la propriété de la chaîne "Bonjour" est transférée à s2.
- s1 devient invalide après le transfert, et Rust empêche toute utilisation de s1 après ce point pour éviter des erreurs.

```
fn main() {  
    let s1 = String::from("Bonjour"); // s1 possède la chaîne  
    "Bonjour"  
    let s2 = s1;                       // La propriété de "Bonjour" est  
    transférée à s2  
  
    // println!("{}", s1); // Erreur : s1 n'est plus valide  
    println!("{}", s2); // Fonctionne, car s2 est maintenant le  
    propriétaire  
}
```



Clonage des Donnée

Si vous avez besoin de copier une valeur sans transférer la propriété, Rust propose la méthode `clone()`.

Cela crée une copie complète de la donnée, ce qui signifie que chaque variable peut posséder sa propre version de la valeur sans dépendre d'une autre.

Dans ce cas, `s1` et `s2` possèdent chacun leur propre copie de la chaîne "Bonjour" et peuvent être utilisés indépendamment.

```
fn main() {  
    let s1 = String::from("Bonjour");  
    let s2 = s1.clone(); // Crée une copie profonde de la chaîne  
    "Bonjour"  
  
    println!("s1 : {}, s2 : {}", s1, s2); // Les deux valeurs sont valides  
}
```



Types de Données avec Propriété et Copie

Rust distingue deux catégories de types de données pour la propriété :

1. Types qui se déplacent (move) : Comme String, Vec, et autres types de données alloués sur le tas (heap). La propriété est transférée, à moins qu'on utilise clone().
2. Types qui se copient (copy) : Les types simples (entiers, flottants, booléens, caractères, etc.) sont copiés par défaut, car ils sont stockés sur la pile (stack) et sont plus rapides à dupliquer.
- 3.

Les types qui implémentent le trait Copy ne transfèrent pas la propriété ; ils sont simplement copiés.

```
fn main() {  
    let x = 5; // x est un entier (implémente Copy)  
    let y = x; // Copie la valeur de x dans y  
  
    println!("x : {}, y : {}", x, y); // Les deux variables sont valides  
}
```



Transfert de Propriété dans les Fonctions

Quand on passe une valeur à une fonction, la propriété de cette valeur peut être transférée. Si une fonction prend la propriété d'une valeur, la variable d'origine devient invalide après l'appel de la fonction.

Dans cet exemple, `s1` est déplacé dans la fonction `afficher_chaine`, et après cet appel, `s1` n'est plus accessible.

```
fn afficher_chaine(s: String) {  
    println!("{}", s);  
}  
  
fn main() {  
    let s1 = String::from("Hello");  
    afficher_chaine(s1); // s1 est déplacé dans la fonction  
  
    // println!("{}", s1); // Erreur : s1 n'est plus valide  
}
```



Retour de Propriété depuis une Fonction

Pour récupérer la propriété d'une valeur déplacée dans une fonction, vous pouvez la renvoyer depuis la fonction.

Ici, la fonction `donner_chaine` renvoie la chaîne, et `main` récupère la propriété de cette chaîne.

```
fn donner_chaine() -> String {  
    let s = String::from("Retour");  
    s // Renvoie la chaîne, transférant la propriété  
}  
  
fn main() {  
    let s = donner_chaine(); // Récupère la propriété renvoyée  
    println!("{}", s);  
}
```



Résumé de Ownership

Le concept de propriété en Rust garantit une gestion sûre et efficace de la mémoire :

- Chaque valeur a un propriétaire unique, ce qui contrôle l'accès et la libération de la mémoire.
- Le transfert de propriété se produit lors d'assignations ou de passages en fonction, rendant la variable d'origine invalide.
- La méthode `clone()` permet de dupliquer les données sans transférer la propriété.
- Types Copy (simples et de petite taille) sont copiés automatiquement, tandis que les types move (alloués sur le tas) nécessitent un transfert de propriété ou un clonage explicite.

En maîtrisant la propriété, les développeurs Rust écrivent des programmes sûrs sans fuite de mémoire, sans ramasse-miettes et avec une performance optimale.



L'Emprunt (Borrowing)

En Rust, le concept d'emprunt (borrowing) permet d'accéder aux valeurs sans en transférer la propriété, ce qui rend les programmes plus flexibles et évite les copies inutiles de données. Les emprunts permettent d'utiliser des références pour lire ou modifier des valeurs possédées par d'autres variables sans avoir à transférer la propriété. Cela aide à garder une gestion de la mémoire efficace tout en permettant de partager des données entre différentes parties du code.

Rust propose deux types d'emprunts :

1. Emprunt immuable (&T) : Permet de lire la valeur sans la modifier.
2. Emprunt mutable (&mut T) : Permet de modifier la valeur empruntée.

Les Règles de l'Emprunt

Pour garantir la sécurité et la cohérence des données, Rust impose des règles strictes lors de l'emprunt de valeurs :

1. On peut avoir plusieurs emprunts immuables d'une même valeur en même temps.
2. On ne peut avoir qu'un seul emprunt mutable d'une valeur à la fois.
3. Les emprunts immuables et mutables ne peuvent pas coexister en même temps pour la même valeur.

Ces règles empêchent que des données soient modifiées pendant qu'elles sont en lecture, évitant ainsi des comportements indéfinis et des accès concurrents non sécurisés.



Base de l'emprunt immuable

L'emprunt immuable (&T) permet de lire une valeur sans en transférer la propriété et sans pouvoir la modifier. Vous pouvez avoir plusieurs emprunts immuables d'une même valeur en même temps.

Dans cet exemple :

- r1 et r2 empruntent s de manière immuable, ce qui signifie qu'ils peuvent accéder à s sans modifier son contenu.
- Rust permet plusieurs emprunts immuables simultanés.

```
fn main() {  
    let s = String::from("Bonjour");  
  
    let r1 = &s; // Emprunt immuable  
    let r2 = &s; // Un autre emprunt immuable  
  
    println!("r1: {}, r2: {}", r1, r2); // Utilisation des emprunts  
}
```



Base de l'emprunt mutable

L'emprunt mutable (&mut T) permet de modifier la valeur empruntée, mais impose qu'une seule référence mutable existe pour une valeur donnée à la fois. Cela garantit que personne d'autre n'utilise la valeur pendant qu'elle est modifiée.

Dans cet exemple :

- r1 emprunte s de manière mutable, permettant ainsi d'ajouter du texte à la chaîne s.
- Pendant la durée de vie de r1, il est interdit d'emprunter s de toute autre manière, que ce soit immuable ou mutable.

```
fn main() {  
    let mut s = String::from("Bonjour");  
  
    let r1 = &mut s; // Emprunt mutable  
    r1.push_str(", tout le monde!");  
  
    println!("s: {}", r1); // Accès à l'emprunt mutable  
}
```



Durée de vie des emprunts

Les emprunts sont valides tant que la référence (ou l'emprunt) reste dans le champ de validité (scope) où elle a été créée. Lorsque l'emprunt sort de ce champ de validité, Rust considère qu'il n'est plus utilisé et rend la valeur empruntée disponible pour de nouveaux emprunts.

Dans cet exemple, `r1` est un emprunt mutable limité au bloc intérieur. Une fois ce bloc terminé, `r1` est libéré, permettant la création de l'emprunt immuable `r2` sans erreur.

```
fn main() {  
    let mut s = String::from("Bonjour");  
  
    {  
        let r1 = &mut s; // Emprunt mutable dans un bloc  
        r1.push_str(", tout le monde!");  
    } // r1 sort de son champ de validité ici  
  
    let r2 = &s; // Maintenant on peut créer un emprunt  
    immutable  
    println!("s: {}", r2);  
}
```



Exemple : Emprunt dans une Fonction

Vous pouvez emprunter une valeur en tant que paramètre de fonction sans en transférer la propriété. Voici comment cela fonctionne avec des emprunts immuables et mutables.

Dans cet exemple, longueur emprunte la variable s en tant que référence immuable, de sorte qu'elle puisse accéder à s sans en prendre la propriété. Cela signifie que s reste valide et accessible dans main après l'appel de longueur.

```
fn longueur(s: &String) -> usize {
    s.len()
}

fn main() {
    let s = String::from("Rust");
    let taille = longueur(&s); // Emprunt immuable de s

    println!("La longueur de '{}' est de {}.", s, taille);
}
```



Résumé de Borrowing

L'emprunt est l'un des mécanismes centraux de Rust pour assurer la sécurité de la mémoire tout en maintenant de bonnes performances :

- Les emprunts immuables (&T) permettent la lecture sans transfert de propriété et permettent plusieurs références en même temps.
- Les emprunts mutables (&mut T) permettent la modification mais limitent à une seule référence mutable pour assurer la sécurité des accès concurrents.
- Les règles d'emprunt de Rust garantissent qu'il n'y a pas de références invalides ou de comportements indéfinis lors de l'accès aux données, ce qui améliore la sécurité et la stabilité du programme.



Exemple pratique

En Rust, la gestion de la mémoire évite les "dangling pointers" (pointeurs suspendus ou pendants) grâce à son système de propriétés, d'emprunts, et de durée de vie.

Un dangling pointer est une référence qui pointe vers un emplacement mémoire libéré ou invalide, ce qui est une source fréquente de bugs dans des langages qui ne gèrent pas strictement la mémoire.

Dans cet exemple, nous allons illustrer comment le système d'emprunt et de durée de vie en Rust protège contre l'apparition de pointeurs suspendus.

Situation Risquée (dans d'autres langages) : Référence à un Objet Supprimé

Imaginons un scénario classique où nous créons une référence vers une variable, mais libérons ensuite la mémoire liée à cette variable. Cela causerait des erreurs si on accédait à cette référence après la libération.



Ce que cela donnerait dans un autre langage

Dans un langage comme C ou C++, cela pourrait ressembler à ceci :

Ici, p pointe vers une mémoire déjà libérée, causant un comportement indéfini si on essaie de l'utiliser.

```
int* dangling_pointer() {  
    int* p = new int(10); // Allocation de mémoire sur le tas  
    delete p; // Libération de la mémoire  
    return p; // Retourne un pointeur suspendu  
}  
  
int main() {  
    int* p = dangling_pointer();  
    std::cout << *p; // Erreur : accès à un pointeur suspendu  
}
```



Et en Rust?

En Rust, les règles de propriété et de durée de vie empêchent ce type d'erreur. Voici un exemple similaire en Rust, mais où Rust nous empêche de créer des références invalides.

Rust va signaler des erreurs si on tente de créer un pointeur suspendu. Pour illustrer cela, créons une fonction qui renvoie une référence sur une variable.

Dans cet exemple :

1. `s` est créé dans la fonction `dangle` et vit seulement à l'intérieur de celle-ci.
2. La tentative de renvoyer `&s` échoue parce que `s` sera libéré dès que `dangle` termine.
3. Rust détecte cela et empêche le retour de la référence, signalant une erreur de durée de vie, car `s` sort du champ de validité (scope) lorsque la fonction se termine.

```
fn dangle() -> &String {  
    let s = String::from("Bonjour");  
    &s // Erreur : retourne une référence à une variable locale  
}  
  
fn main() {  
    let reference = dangle(); // Rust ne permet pas cette opération  
}
```



Solution

Pour résoudre ce problème, on peut retourner la propriété de `s` plutôt qu'une référence :

Ici :

- En renvoyant `s` directement sans référence, la propriété de `s` est transférée à la variable `ownership` dans `main`.
- Empêche Rust de détecter un possible pointeur suspendu.

Conclusion:

Rust évite les pointeurs suspendus en imposant des règles strictes sur la propriété et la durée de vie :

- Retourner une référence à une variable locale est interdit, car cela crée un pointeur suspendu.
- La durée de vie des références est vérifiée par le compilateur, qui s'assure que toutes les références restent valides tant qu'elles sont utilisées.
- En cas de besoin, on peut transférer la propriété d'une valeur ou utiliser un clonage pour éviter tout risque.

```
fn dangle_fixed() -> String {  
    let s = String::from("Bonjour");  
    s // Retourne la chaîne en transférant la propriété  
}  
  
fn main() {  
    let ownership = dangle_fixed();  
    println!("{}", ownership); // Affiche "Bonjour" sans erreur  
}
```



Gestion des erreurs



Introduction

Rust propose une approche unique dans le traitement des erreurs qui se concentre sur la sécurité et la robustesse. Rust n'utilise pas d'exceptions comme d'autres langages, mais repose plutôt sur des types de données spécifiques pour gérer les erreurs : Result et Option.

Ces types rendent le code plus explicite en forçant le programmeur à traiter les erreurs de manière proactive.

Il existe deux types principaux d'erreurs en Rust :

- Erreurs récupérables : Ce sont des erreurs que le programme peut gérer. Par exemple, une erreur d'entrée/sortie (I/O) ou un échec lors de la recherche d'un fichier. Ces erreurs peuvent être retournées sous forme de Result.
- Erreurs non récupérables : Ce sont des erreurs que le programme ne peut pas raisonnablement gérer. Par exemple, une tentative de débordement d'une pile ou un index hors limites. Ces erreurs provoquent généralement l'arrêt du programme et peuvent être gérées via le panique (panic!).



Avec Result

Rappel: Le type Result est un énuméré qui représente soit une réussite (Ok), soit une erreur (Err). La définition de Result ressemble à ceci :

```
enum Result<T, E> {  
    Ok(T), // Représente une valeur de succès  
    Err(E), // Représente une valeur d'erreur  
}
```

Voici un exemple simple qui montre comment utiliser le type Result pour gérer les erreurs lors de la conversion d'une chaîne en un entier :

Dans cet exemple :

- La fonction `convert_to_integer` tente de convertir une chaîne en entier.
- Elle retourne un Result qui peut être soit `Ok(value)`, soit `Err(e)`.
- Dans `main`, on utilise un `match` pour gérer les deux cas : un cas de succès et un cas d'erreur.

```
fn convert_to_integer(s: &str) -> Result<i32,  
std::num::ParseIntError> {  
    s.parse::<i32>() // Essaie de convertir la chaîne en entier  
}  
  
fn main() {  
    let input = "42";  
    match convert_to_integer(input) {  
        Ok(value) => println!("La conversion a réussi : {}", value),  
        Err(e) => println!("Erreur de conversion : {}", e),  
    }  
  
    let bad_input = "abc";  
    match convert_to_integer(bad_input) {  
        Ok(value) => println!("La conversion a réussi : {}", value),  
        Err(e) => println!("Erreur de conversion : {}", e),  
    }  
}
```



Avec Option

Rappel: Le type Option est utilisé lorsque la valeur peut être présente ou non. C'est souvent utilisé pour des cas où l'absence de valeur est une possibilité légitime. Voici la définition de Option

```
enum Option<T> {  
    Some(T), // Contient une valeur  
    None,   // Aucune valeur  
}
```

Dans cet exemple :

- La fonction find_item cherche un élément dans une liste et retourne un Option.
- Si l'élément est trouvé, elle retourne Some(index) ; sinon, elle retourne None.
- Dans main, on utilise un match pour gérer les deux cas.

```
fn find_item(list: &[&str], item: &str) -> Option<usize> {  
    for (index, &value) in list.iter().enumerate() {  
        if value == item {  
            return Some(index); // Renvoie l'index si l'élément est trouvé  
        }  
    }  
    None // Renvoie None si l'élément n'est pas trouvé  
}  
  
fn main() {  
    let items = ["pomme", "banane", "orange"];  
  
    match find_item(&items, "banane") {  
        Some(index) => println!("Élément trouvé à l'index : {}", index),  
        None => println!("Élément non trouvé."),  
    }  
  
    match find_item(&items, "kiwi") {  
        Some(index) => println!("Élément trouvé à l'index : {}", index),  
        None => println!("Élément non trouvé."),  
    }  
}
```



Traitement des erreurs avec ?

Rust propose l'opérateur ? pour simplifier le traitement des erreurs. Il peut être utilisé à l'intérieur des fonctions qui retournent un type Result. Si l'expression avec ? est Ok, la valeur est extraite ; si elle est Err, l'erreur est retournée de la fonction.

Dans cet exemple :

- La fonction `read_file_contents` utilise ? pour ouvrir un fichier et lire son contenu.
- Si `File::open` ou `file.read_to_string` échoue, l'erreur est retournée automatiquement, simplifiant le code.
-

```
use std::fs::File;
use std::io::{self, Read};

fn read_file_contents(file_path: &str) -> Result<String, io::Error> {
    let mut file = File::open(file_path)?; // Utilise `?` pour gérer l'erreur
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents) // Retourne les contenus du fichier
}

fn main() {
    match read_file_contents("fichier.txt") {
        Ok(contents) => println!("Contenu du fichier : {}", contents),
        Err(e) => println!("Erreur de lecture du fichier : {}", e),
    }
}
```



Traitement des erreurs avec panic!

Pour les erreurs non récupérables, Rust propose la macro `panic!`, qui arrête le programme en cas de problème critique. Par exemple :

Ici, le programme panique car il essaie d'accéder à un index qui n'existe pas dans le vecteur.

```
fn main() {  
    let value = vec![1, 2, 3];  
    println!("Le quatrième élément est : {}", value[3]); // Cause un panic  
}
```



Traitement des erreurs avec unwrap

La méthode unwrap est utilisée pour extraire la valeur contenue dans un Option ou un Result. Si la valeur est Some ou Ok, unwrap retourne cette valeur.

En revanche, si la valeur est None ou Err, le programme panique avec un message d'erreur par défaut.

Dans cet exemple :

- Si get_first_item retourne None, l'appel à unwrap provoque un panic. Cela peut être acceptable si vous êtes sûr que la liste n'est jamais vide. Toutefois, dans des scénarios où cela pourrait se produire, l'utilisation de unwrap est risquée.

```
fn get_first_item<'a>(list: &'a [&'a str]) -> Option<&'a str> {  
    list.get(0).cloned() // Renvoie le premier élément ou None  
}  
  
fn main() {  
    let items = ["pomme", "banane", "orange"];  
  
    // Utilisation de unwrap pour obtenir le premier élément  
    let first_item = get_first_item(&items).unwrap(); // Panique si None  
    println!("Le premier élément est : {}", first_item);  
}
```



Traitement des erreurs avec expect

La méthode `expect` fonctionne de manière similaire à `unwrap`, mais elle vous permet de fournir un message d'erreur personnalisé. Cela peut être utile pour le débogage, car il vous donne un contexte sur l'erreur survenue.

Dans cet exemple :

- Si l'ouverture ou la lecture du fichier échoue, `expect` panique et affiche le message que vous avez fourni.
- Cela rend le débogage plus facile, car vous avez une meilleure idée de l'endroit où l'erreur s'est produite.

```
use std::fs::File;
use std::io::{self, Read};

fn read_file_contents(file_path: &str) -> Result<String, io::Error> {
    let mut file = File::open(file_path).expect("Échec de l'ouverture du fichier");
    // Message personnalisé
    let mut contents = String::new();
    file.read_to_string(&mut contents).expect("Échec de la lecture du fichier");
    // Message personnalisé
    Ok(contents)
}

fn main() {
    let contents = read_file_contents("fichier.txt").expect("Erreur lors de la lecture du fichier");
    println!("Contenu du fichier : {}", contents);
}
```



Quand utiliser unwrap et expect

Bien que unwrap et expect soient utiles pour simplifier le code, il est important de les utiliser judicieusement :

- Utilisez unwrap si vous êtes absolument certain que la valeur sera présente et qu'une absence de valeur est considérée comme un bug. Par exemple, lorsque vous êtes sûr qu'une liste n'est jamais vide.
- Utilisez expect lorsque vous voulez plus de clarté sur l'endroit où le programme pourrait échouer. Cela permet de rendre le code plus robuste et plus facile à déboguer en fournissant un message d'erreur contextuel.



Structs



Introduction

Les structs (structures) en Rust sont des types de données personnalisés qui permettent de regrouper plusieurs valeurs sous un même nom.

Elles sont utilisées pour créer des objets qui contiennent des attributs et des méthodes.

Les structs sont essentielles pour organiser les données et représenter des entités avec des caractéristiques spécifiques.



Déclaration d'une Struct

Pour déclarer une struct, vous utilisez le mot-clé struct, suivi du nom de la structure et de ses champs.

Chaque champ a un nom et un type.

```
struct Person {  
    name: String,  
    age: u32,  
}
```



Création d'une Instance de Struct

Pour créer une instance d'une struct, vous utilisez une syntaxe similaire à la création d'un objet dans d'autres langages.

```
fn main() {  
    let person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    // Affichage des informations  
    println!("Nom : {}, Âge : {}", person.name, person.age);  
}
```



Accès aux Champs

Pour accéder aux champs d'une struct, vous utilisez la notation pointée.

```
fn main() {  
    let person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    // Accès aux champs  
    println!("Nom : {}", person.name);  
    println!("Âge : {}", person.age);  
}
```



Modifier les Champs

Grâce à la notation pointée ainsi qu'à la mutabilité, vous pouvez modifier les champs.

```
fn main() {  
    let mut person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    person.name = "Jean".to_string();  
  
    // Affichage des informations  
    println!("Nom : {}, Âge : {}", person.name, person.age);  
}  
  
struct Person {  
    name: String,  
    age: u32,  
}
```



Méthodes Associées aux Structs

Les structs ne peuvent pas voir directement des méthodes au sein de la structure mais peuvent avoir des méthodes associées, ce qui vous permet de définir des comportements pour vos objets.

Pour définir des méthodes, vous utilisez le mot-clé `impl` suivi du nom de la struct.

```
impl Person {  
    // Méthode pour afficher les informations de la personne  
    fn display(&self) {  
        println!("Nom : {}, Âge : {}", self.name, self.age);  
    }  
    // Méthode pour augmenter l'âge  
    fn birthday(&mut self) {  
        self.age += 1; // Augmente l'âge de 1  
    }  
}  
  
fn main() {  
    let mut person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    // Appel de la méthode display  
    person.display();  
    // Appel de la méthode birthday  
    person.birthday();  
    person.display(); // Affiche l'âge mis à jour  
}
```



Structs et Constructeurs

Rust ne dispose pas de constructeurs au sens classique, mais vous pouvez créer une fonction associée pour instancier une struct plus facilement.

```
impl Person {  
    // Constructeur pour créer une nouvelle personne  
    fn new(name: String, age: u32) -> Person {  
        Person { name, age }  
    }  
}  
  
fn main() {  
    let person = Person::new(String::from("Alice"), 30);  
    person.display();  
}
```



Structs Tuple

En plus des structs avec des champs nommés, Rust permet également de définir des structs tuple, qui sont similaires aux tuples, mais avec des noms de type.

```
struct Color(u8, u8, u8); // Struct tuple pour représenter une couleur RGB

fn main() {
    let black = Color(0, 0, 0);
    println!("Noir - R: {}, G: {}, B: {}", black.0, black.1, black.2);
}
```



Structs avec Champs Optionnels

Rust permet également d'utiliser des types Option comme champs dans une struct, ce qui peut être utile pour représenter des valeurs optionnelles.

```
struct Person {  
    name: String,  
    age: Option<u32>, // Champ optionnel pour l'âge  
}  
  
fn main() {  
    let person_with_age = Person {  
        name: String::from("Alice"),  
        age: Some(30),  
    };  
  
    let person_without_age = Person {  
        name: String::from("Bob"),  
        age: None,  
    };  
  
    // Affichage des informations  
    println!("Nom : {}", person_with_age.name);  
    match person_with_age.age {  
        Some(age) => println!("Âge : {}", age),  
        None => println!("Âge non spécifié"),  
    }  
  
    println!("Nom : {}", person_without_age.name);  
    match person_without_age.age {  
        Some(age) => println!("Âge : {}", age),  
        None => println!("Âge non spécifié"),  
    }  
}
```



Conclusion

Les structs en Rust sont un moyen puissant de créer des types de données complexes et d'organiser les informations de manière structurée.

Elles permettent de :

- Définir des types personnalisés avec des champs.
- Créer des méthodes associées pour encapsuler des comportements.
- Utiliser des types comme Option pour des champs facultatifs.
- Créer des structs tuple pour des ensembles de valeurs sans noms de champs explicites.



Enums



Introduction

Les énumérations (ou enums) sont des types de données qui permettent de définir un ensemble de variantes possibles.

Chaque variante d'une énumération peut contenir des données, ce qui en fait un outil puissant pour modéliser des choix, des états ou des types complexes dans votre programme.

Les énumérations sont particulièrement utiles lorsqu'un type peut prendre plusieurs formes différentes.



Déclaration

Pour déclarer une énumération, vous utilisez le mot-clé `enum`, suivi du nom de l'énumération et de ses variantes.

Dans cet exemple, l'énumération `Direction` définit quatre variantes possibles : `Up`, `Down`, `Left` et `Right`.

```
enum Direction {  
    Up,  
    Down,  
    Left,  
    Right,  
}
```



Utilisation

Les énumérations peuvent être utilisées dans le code de manière similaire aux types de données traditionnels.

Vous pouvez créer des variables qui prennent l'une des variantes définies dans l'énumération.

Dans cet exemple :

- La fonction `move_player` prend un paramètre de type `Direction` et utilise une instruction `match` pour déterminer quelle action effectuer en fonction de la direction donnée.
- Cela permet d'effectuer des actions différentes en fonction de la valeur de l'énumération.

```
fn move_player(direction: Direction) {  
    match direction {  
        Direction::Up => println!("Le joueur monte"),  
        Direction::Down => println!("Le joueur descend"),  
        Direction::Left => println!("Le joueur va à gauche"),  
        Direction::Right => println!("Le joueur va à droite"),  
    }  
}  
  
fn main() {  
    let direction = Direction::Up;  
    move_player(direction);  
}
```



Données associées

Les énumérations peuvent également contenir des données associées.

Cela vous permet de stocker des valeurs supplémentaires avec chaque variante.

Dans cet exemple :

- L'énumération Message définit trois variantes : Quit, ChangeColor, et Move.
- ChangeColor prend trois i32 pour représenter les valeurs RGB, tandis que Move utilise une structure anonyme avec des champs nommés.
- L'instruction match permet d'extraire facilement les données associées à chaque variante.

```
enum Message {  
    Quit,  
    ChangeColor(i32, i32, i32), // RGB  
    Move { x: i32, y: i32 },    // Structure anonyme  
}  
  
fn main() {  
    let msg = Message::ChangeColor(255, 0, 0);  
  
    match msg {  
        Message::Quit => println!("Quitter le programme"),  
        Message::ChangeColor(r, g, b) => {  
            println!("Changer la couleur en : R={}, G={}, B={}", r, g, b);  
        },  
        Message::Move { x, y } => println!("Déplacer à : x={}, y={}", x, y),  
    }  
}
```



Enumérateurs en tant que Types de Résultat

Les énumérations sont souvent utilisées pour représenter des types de résultats, comme le succès ou l'échec d'une opération. Les types Result et Option en Rust sont des exemples courants d'énumérations.

```
enum Result<T, E> {  
    Ok(T), // Variante pour le succès  
    Err(E), // Variante pour l'échec  
}  
  
fn divide(numerator: f64, denominator: f64) -> Result<f64, String> {  
    if denominator == 0.0 {  
        Result::Err(String::from("Division par zéro"))  
    } else {  
        Result::Ok(numerator / denominator)  
    }  
}  
  
fn main() {  
    let result = divide(10.0, 0.0);  
  
    match result {  
        Result::Ok(value) => println!("Le résultat est : {}", value),  
        Result::Err(message) => println!("Erreur : {}", message),  
    }  
}
```



Conclusion

Avantages des Enums

Les énumérations offrent plusieurs avantages :

- Clarté : Elles rendent le code plus lisible et explicite en définissant clairement les variantes possibles.
- Sécurité : Le compilateur Rust vérifie que toutes les variantes sont gérées dans une instruction match, réduisant ainsi les erreurs potentielles.
- Flexibilité : Les énumérations peuvent contenir des données associées, permettant de créer des types complexes.

Conclusion

Les énumérations en Rust sont un outil puissant pour créer des types de données complexes et modéliser des comportements variés dans votre code. Elles permettent de :

- Définir des types avec un ensemble limité de variantes.
- Ajouter des données associées à chaque variante.
- Utiliser des instructions match pour gérer les différentes variantes de manière sécurisée.



Traits



Un Trait ?

Les traits sont une façon dans le langage Rust d'implémenter des comportements partagés entre différents types.

Un trait définit donc un ensemble de méthodes que les types doivent implémenter pour être considérés comme "possédant" ce trait.

Cela permet de créer des abstractions et de garantir que des types différents peuvent être utilisés de manière interchangeable si ils implémentent le même trait.

Vous pouvez par exemple comparer ceci aux interfaces ou dans une moindre mesure aux protocoles selon les langages



Trait

Un trait est semblable à une interface dans d'autres langages de programmation.

Il peut contenir des signatures de méthodes, des méthodes par défaut et des types associés.

```
// Déclaration d'un trait
trait Descriptible {
    fn description(&self) -> String; // Méthode sans corps (signature)
}
```



Implémentation

Pour qu'un type implémente un trait, vous devez fournir des définitions pour toutes les méthodes déclarées dans le trait.

// Déclaration d'un trait

trait Descriptible {

fn description(&self) -> String; // Méthode sans corps

(signature)

}

```
struct Person {
    name: String,
    age: u32,
}

impl Descriptible for Person {
    fn description(&self) -> String {
        format!("{}", self.name, self.age)
    }
}

fn main() {
    let person = Person {
        name: String::from("Alice"),
        age: 30,
    };

    // Appel de la méthode du trait
}
```



Traits avec Méthodes par Défaut

Les traits peuvent également définir des méthodes avec des implémentations par défaut. Les types qui implémentent le trait peuvent choisir de remplacer ces méthodes ou d'utiliser l'implémentation par défaut.

```
trait Descriptible {  
    fn description(&self) -> String;  
  
    // Méthode par défaut  
    fn detailed_description(&self) -> String {  
        format!("Description détaillée non disponible.")  
    }  
}
```

```
struct Person {  
    name: String,  
    age: u32,  
}  
  
impl Descriptible for Person {  
    fn description(&self) -> String {  
        format!("{}", self.name, self.age)  
    }  
}  
  
fn main() {  
    let person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    println!("{}", person.description());  
    println!("{}", person.detailed_description()); // Utilise la méthode par défaut  
}
```



Trait Bounds

Les trait bounds sont utilisés pour spécifier que des types doivent implémenter certains traits afin d'être utilisés dans des fonctions ou des structures. Cela permet d'écrire du code générique qui fonctionne avec différents types tant qu'ils respectent les contraintes spécifiées.

```
fn afficher_description<T: Descriptible>(item: T) {  
    println!("{}", item.description());  
}  
  
fn main() {  
    let person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    afficher_description(person);  
}
```



Traits et Polymorphisme

Les traits permettent également d'atteindre le polymorphisme. En utilisant des références ou des pointeurs intelligents, vous pouvez manipuler des objets de différents types qui implémentent le même trait.

```
fn afficher_description(item: &dyn Descriptible) {  
    println!("{}", item.description());  
}  
  
fn main() {  
    let person = Person {  
        name: String::from("Alice"),  
        age: 30,  
    };  
  
    afficher_description(&person); // Utilise le polymorphisme  
}
```



Traits pour des Types Associés

Les traits peuvent également définir des types associés, ce qui permet de créer des relations entre des types au sein d'un trait.

```
trait Identifiable {  
    type IdType; // Type associé  
  
    fn id(&self) -> Self::IdType;  
}
```

```
struct Person {  
    id: u32,  
    name: String,  
}  
  
impl Identifiable for Person {  
    type IdType = u32;  
  
    fn id(&self) -> Self::IdType {  
        self.id  
    }  
}  
  
fn main() {  
    let person = Person {  
        id: 1,  
        name: String::from("Alice"),  
    };  
  
    println!("ID de la personne : {}", person.id());  
}
```



Faisons une pause

Les traits en Rust sont un mécanisme fait pour l'abstraction et la réutilisation du code. Ils permettent d'implémenter des comportements partagés de manière sûre et efficace, facilitant ainsi la conception de systèmes complexes.

En utilisant des traits, vous pouvez :

- Définir des interfaces pour des comportements partagés.
- Fournir des implémentations par défaut pour des méthodes.
- Utiliser des trait bounds pour écrire du code générique.
- Obtenir du polymorphisme en utilisant des références ou des pointeurs intelligents.
- Créer des types associés pour établir des relations entre types.

Cela fait des traits un outil à ne pas négliger pour écrire du code Rust modulaire, maintenable et extensible.

Rust possède aussi plusieurs traits standard intégrés, qui facilitent l'implémentation de comportements courants pour différents types de données. Ces traits permettent d'ajouter des fonctionnalités spécifiques et peuvent être automatiquement dérivés pour la plupart des types.

Voici les traits standard les plus utilisés : Clone, Copy, Debug et PartialEq.



Le Trait Clone

Le trait Clone est utilisé pour créer une copie d'un objet en allouant de la nouvelle mémoire, ce qui est nécessaire pour les types complexes qui gèrent des ressources. Si un type implémente le trait Clone, vous pouvez utiliser la méthode `.clone()` pour créer une copie indépendante de l'instance.

```
#[derive(Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point1 = Point { x: 5, y: 10 };
    let point2 = point1.clone(); // Copie indépendante de `point1`

    println!("Point1: ({} , {})", point1.x, point1.y);
    println!("Point2: ({} , {})", point2.x, point2.y);
}
```



Le Trait Copy

Le trait Copy permet de copier des valeurs sans allouer de mémoire supplémentaire ni nécessiter d'appel explicite à `.clone()`. Ce trait est principalement utilisé pour les types simples (comme les entiers et les booléens) et les types composés de valeurs qui elles-mêmes implémentent Copy.

En revanche, un type qui contient des données dynamiques (comme `String`) ne peut pas implémenter Copy.

Pour qu'un type soit Copy, il doit également implémenter Clone, ce qui est fait automatiquement pour les types primitifs en Rust.

```
#[derive(Copy, Clone)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point1 = Point { x: 5, y: 10 };
    let point2 = point1; // `point1` peut toujours être utilisé après cette affectation

    println!("Point1: ({} , {})", point1.x, point1.y);
    println!("Point2: ({} , {})", point2.x, point2.y);
}
```



Le Trait Debug

Le trait Debug permet d'afficher les informations internes d'une instance, ce qui est essentiel pour le débogage.

En dérivant Debug, vous pouvez utiliser le format `{:?}` dans `println!` pour obtenir une représentation lisible du contenu de l'instance.

La plupart des types en Rust implémentent Debug, y compris les types de base comme `i32` et `String`.

Pour obtenir un affichage plus lisible, vous pouvez utiliser `:#?` pour formater en mode multi-lignes :

```
println!("{:#?}", point);
```

```
#[derive(Debug)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point = Point { x: 5, y: 10 };
    println!("{:?}", point); // Affiche "Point { x: 5, y: 10 }"
}
```



Le Trait PartialEq

Le trait PartialEq permet de comparer deux instances pour vérifier si elles sont égales (==) ou différentes (!=).

Si un type implémente PartialEq, vous pouvez utiliser les opérateurs de comparaison == et !=.

La dérivation automatique de PartialEq rend cette comparaison simple et rapide pour les structures de données composées.

```
#[derive(PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point1 = Point { x: 5, y: 10 };
    let point2 = Point { x: 5, y: 10 };
    let point3 = Point { x: 7, y: 10 };

    println!("point1 == point2 : {}", point1 == point2); // true
    println!("point1 == point3 : {}", point1 == point3); // false
}
```



Combinaisons de traits

Rust permet de combiner plusieurs traits en utilisant `#[derive(...)]`, ce qui simplifie leur utilisation pour les types de données personnalisés. Cela est particulièrement utile pour les types simples ou les structures de données sans gestion complexe de la mémoire.

Conclusion

Les traits standard tels que `Clone`, `Copy`, `Debug`, et `PartialEq` sont essentiels pour simplifier et sécuriser la manipulation des données en Rust. Ils permettent de :

- Créer des copies de valeurs (`Clone` et `Copy`)
- Afficher le contenu des instances pour le débogage (`Debug`)
- Comparer les instances pour l'égalité (`PartialEq`)

En combinant ces traits, vous pouvez écrire du code idiomatique, sûr et plus facile à maintenir dans vos projets Rust.

```
#[derive(Debug, Clone, Copy, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point1 = Point { x: 5, y: 10 };
    let point2 = point1; // `Copy` permet la copie sans déplacer la valeur

    println!("{:?}", point1); // `Debug` permet l'affichage
    println!("Les points sont égaux : {}", point1 == point2); // `PartialEq` permet la
    comparaison
}
```



Durée de Vie



Comprendre les Durées de Vie

Les durées de vie (ou lifetimes) sont une fonctionnalité de sécurité mémoire qui permet de s'assurer que les références dans un programme sont toujours valides.

Les durées de vie servent donc à gérer la durée pendant laquelle une référence reste accessible et valide. Rust utilise les durées de vie pour éviter les références non valides (dangling pointers) et les accès concurrents aux données, garantissant ainsi la sûreté mémoire sans nécessiter de ramasse-miettes (garbage collector).

Pourquoi les Durées de Vie sont-elles Nécessaires ?

Les durées de vie sont particulièrement utiles pour éviter deux types de problèmes :

- Les références invalides (dangling references), où une référence pointerait vers une donnée libérée ou déplacée.
- Les conflits d'accès concurrent dans le cas d'emprunts multiples, où l'accès mutable et l'accès immuable pourraient causer des comportements imprévisibles.



Syntaxe de Durées de Vie

Les durées de vie en Rust sont indiquées par des annotations de durée de vie, représentées par une apostrophe (') suivie d'un nom, comme 'a. Ces annotations sont généralement utilisées dans les signatures de fonction pour indiquer les relations de durée de vie entre les paramètres.

Dans cet exemple, 'a est une annotation de durée de vie. Elle indique que la durée de vie de la référence renvoyée par la fonction `plus_longue` est la même que celle des références `s1` et `s2`. Cela signifie que le résultat sera valide tant que `s1` et `s2` sont valides.

```
fn plus_longue<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

fn main() {
    let string1 = String::from("Bonjour");
    let string2 = String::from("Salut");

    // Appel de la fonction `plus_longue` avec des références vers `string1` et `string2`
    let resultat = plus_longue(&string1, &string2);

    println!("La plus longue chaîne est : {}", resultat);
}
```



Règles des Durées de Vie

Rust suit certaines règles pour déterminer automatiquement les durées de vie (appelées lifetime elision rules), mais dans les cas complexes, il est nécessaire d'ajouter des annotations explicites pour aider le compilateur à comprendre les relations entre les références.

Les règles sont :

1. Pour les fonctions sans retour de référence : Rust infère que chaque paramètre de référence a sa propre durée de vie.
2. Pour les fonctions avec une référence de retour : Rust infère que la durée de vie de la référence de retour est celle du paramètre de référence s'il n'y a qu'un seul paramètre.
3. Pour les fonctions avec plusieurs références de retour et paramètres : des annotations de durées de vie explicites sont souvent nécessaires.



Exemple de Fonction avec Durée de Vie Explicite

Lorsque plusieurs références sont passées en paramètres, Rust ne peut pas toujours déterminer les relations de durée de vie. Une annotation explicite devient alors nécessaire pour éviter les ambiguïtés.

Cet exemple soulève une erreur car resultat référence string2, qui est libérée à la fin du bloc. En spécifiant que la durée de vie de resultat est celle de s1, Rust empêche une référence vers des données invalides.

```
fn choisir_premiere<'a>(s1: &'a str, s2: &str) -> &'a str {  
    s1 // retourne une référence qui a la même durée de vie que `s1`  
}  
  
fn main() {  
    let string1 = String::from("longue chaîne");  
    let resultat;  
    {  
        let string2 = String::from("courte");  
        resultat = choisir_premiere(&string1, &string2);  
    } // `string2` n'est plus valide ici, `resultat` non plus  
}
```



Durées de Vie et Structures

Dans les structures, les annotations de durée de vie sont nécessaires lorsque vous voulez stocker des références dans les champs de la structure.

Dans cet exemple, l'annotation 'a indique que la référence texte a la même durée de vie que la référence passée pour initialiser l'instance Extrait.

```
struct Extrait<'a> {  
    texte: &'a str,  
}  
  
fn main() {  
    let phrase = String::from("Bonjour tout le monde !");  
    let extrait = Extrait { texte: &phrase };  
    println!("Extrait : {}", extrait.texte);  
}
```



Durées de Vie dans les méthodes

Dans les méthodes associées aux structures, les annotations de durée de vie peuvent être nécessaires pour relier les durées de vie des références entrantes et de celles des champs.

Ici, l'annotation 'a est appliquée à la méthode afficher pour indiquer que la durée de vie de self (une référence vers Extrait) dépend de celle de 'a, c'est-à-dire de la durée de vie de texte.

```
impl<'a> Extrait<'a> {  
    fn afficher(&self) {  
        println!("Extrait : {}", self.texte);  
    }  
}
```



Modules



Modules?

En Rust, les modules permettent de structurer le code de manière claire et organisée, ce qui améliore la lisibilité, la réutilisabilité, et la maintenabilité des projets, surtout lorsque le projet devient complexe.

Les modules (`mod`) aident à regrouper les fonctionnalités logiquement et à séparer le code en plusieurs fichiers, tout en définissant des scopes pour éviter les conflits de noms.

Les modules en Rust sont définis en utilisant le mot-clé `mod`. Par défaut, tous les éléments définis dans un module sont privés ; il faut les rendre publics avec le mot-clé `pub` pour les exposer en dehors du module.



Définir un module

Dans cet exemple :

- math est un module contenant deux fonctions : addition et soustraction.
- addition est publique (pub), donc accessible dans main.
- soustraction est privée, donc inaccessible en dehors de math.

```
mod math {  
  pub fn addition(a: i32, b: i32) -> i32 {  
    a + b  
  }  
  
  fn soustraction(a: i32, b: i32) -> i32 {  
    a - b  
  }  
}  
  
fn main() {  
  let result = math::addition(5, 3);  
  println!("Résultat de l'addition : {}", result);  
  // let result_sub = math::soustraction(5, 3); // Erreur : soustraction est privée  
}
```



Module et fichier

En Rust, un module est généralement défini dans un fichier séparé.

Pour cela, créez un fichier portant le même nom que le module et **placez-le dans le dossier src**. Rust chargera automatiquement ce fichier comme module.

Par exemple, pour créer un module math dans un fichier :

1. Créez un fichier src/math.rs.
2. Déplacez le code du module math dans ce fichier en omettant `mod math {}`. Prenez juste le code entre les accolades
3. Dans main.rs, vous pouvez référencer ce module comme suit : **`mod math;`**



Module imbriqué

Les modules peuvent être imbriqués pour organiser encore plus finement le code.

```
mod utilitaires {  
  pub mod math {  
    pub fn addition(a: i32, b: i32) -> i32 {  
      a + b  
    }  
  }  
  
  pub mod affichage {  
    pub fn afficher_message(message: &str) {  
      println!("{}", message);  
    }  
  }  
}  
  
fn main() {  
  let result = utilitaires::math::addition(5, 3);  
  utilitaires::affichage::afficher_message("Calcul terminé !");  
  println!("Résultat de l'addition : {}", result);  
}
```



Re-Exporter des Modules avec pub use

Pour simplifier l'accès aux modules, vous pouvez les re-exporter avec pub use, ce qui permet de créer des chemins plus courts.

```
mod utilitaires {  
  pub mod math {  
    pub fn addition(a: i32, b: i32) -> i32 {  
      a + b  
    }  
  }  
  
  pub mod affichage {  
    pub fn afficher_message(message: &str) {  
      println!("{}", message);  
    }  
  }  
  
  pub use self::math::addition; // Re-export de `addition`  
}  
  
fn main() {  
  let result = utilitaires::addition(5, 3); // `addition` est directement accessible  
  utilitaires::affichage::afficher_message("Calcul terminé !");  
  println!("Résultat de l'addition : {}", result);  
}
```



Organisation dans des Sous-Dossiers

Pour les projets de plus grande envergure, il est courant de placer les modules dans des sous-dossiers. Si le module `math` devient complexe, par exemple, on pourrait le placer dans un dossier `src/utilitaires/` et diviser les fonctionnalités en plusieurs fichiers.

Exemple d'Organisation dans des Sous-Dossiers :

1. Créez un dossier `src/utilitaires/`.
2. Ajoutez un fichier `src/utilitaires/mod.rs` (qui est nécessaire pour que Rust reconnaisse le dossier comme module).
3. Placez `math.rs` et `affichage.rs` dans ce dossier.

```
src/  
├── main.rs  
└── utilitaires/  
    ├── mod.rs  
    ├── affichage.rs  
    └── math.rs
```



Organisation dans des Sous-Dossiers

Dans main.rs, vous pouvez maintenant appeler les fonctions de math et affichage :

```
pub mod math;  
pub mod affichage;
```

Dans main.rs, vous pouvez maintenant appeler les fonctions de math et affichage :

```
mod utilitaires;
```

```
fn main() {  
    let result = utilitaires::math::addition(5, 3);  
    utilitaires::affichage::afficher_message("Calcul terminé !");  
    println!("Résultat de l'addition : {}", result);  
}
```



Crates et Packages



Définitions

En Rust, crates et packages sont des concepts liés, mais ils ne sont pas identiques. Ils jouent chacun un rôle distinct dans l'organisation et la distribution du code.



Définition de Crate

Crate

- Une crate est l'unité de compilation minimale en Rust. Elle représente un ensemble de code source Rust que le compilateur Rust (rustc) traite pour générer un binaire ou une bibliothèque.
- Une crate peut être soit :
 - Une crate binaire, qui contient un point d'entrée (fn main()) et produit un exécutable.
 - Une crate de bibliothèque, qui n'a pas de fonction main et est destinée à être utilisée comme bibliothèque de code (définie par lib.rs).
- Une crate est définie par son fichier source principal : main.rs pour les crates binaires ou lib.rs pour les crates de bibliothèque, situé dans le dossier src.

Exemple de structure de crate :

src/

```
|—— main.rs    // Crate binaire  
|—— lib.rs     // Crate bibliothèque
```



Définition de Package

Package

- Un package est un ensemble qui peut contenir une ou plusieurs crates. Il est géré par Cargo, le gestionnaire de packages de Rust, et contient toujours un fichier Cargo.toml, qui décrit le package et ses dépendances.
- Un package peut contenir une crate binaire et/ou plusieurs crates de bibliothèque, mais il peut uniquement avoir une seule crate binaire par défaut (définie dans main.rs).
- Le fichier Cargo.toml est le fichier de configuration central du package. C'est lui qui permet de définir les dépendances, les métadonnées, et les éventuels chemins pour plusieurs crates au sein d'un même package.

Exemple de Structure de Package

Imaginons un package qui contient à la fois une crate binaire et une crate de bibliothèque :

```
my_package/  
├── Cargo.toml    // Fichier de configuration du package  
└── src/  
    ├── main.rs   // Crate binaire  
    └── lib.rs     // Crate bibliothèque
```



Définition de Package

Il est également possible d'ajouter d'autres crates binaires dans le dossier src/bin, chacune ayant son propre fichier *.rs.

my_package/

|—— **Cargo.toml**

|—— **src/**

|—— **main.rs** **// Crate binaire principale**

|—— **lib.rs** **// Crate bibliothèque**

|—— **bin/**

|—— **other_bin.rs** **// Autre crate binaire dans le même package**

|—— **another_bin.rs**



Résumé

En résumé, un package est une enveloppe qui permet d'organiser et de distribuer une ou plusieurs crates.

Aspect	Crate	Package
Définition	Unité de compilation Rust	Ensemble de crates géré par Cargo
Nombre	Une seule crate par compilation	Un package peut contenir plusieurs crates
Point d'entrée	main.rs pour les binaires, lib.rs pour les bibliothèques	Cargo.toml gère la configuration et les dépendances
Utilisation	Compilation en exécutable ou bibliothèque	Organisation de crates avec dépendances et configuration



Introduction à Cargo

Cargo est le gestionnaire de packages et outil de build intégré de Rust. Nous nous en sommes déjà servis à plusieurs reprises comme par exemple:

- Créer un nouveau projet
- Lancer le projet

Il gère le téléchargement, la compilation et les dépendances des packages Rust, permettant de structurer, partager et exécuter facilement du code.



Créer un Nouveau Package

Pour créer un package, utilisez la commande `cargo new`. Cette commande génère une structure de base pour votre projet.

```
cargo new nom_du_package  
cd nom_du_package
```

Par défaut, `cargo new` crée un package avec une crate binaire. Si vous souhaitez créer une crate de bibliothèque (sans `main.rs`), utilisez `--lib` :

```
cargo new --lib nom_du_package
```

Cela crée la structure de fichiers suivante :

```
nom_du_package/  
├── Cargo.toml    // Fichier de configuration  
└── src/  
    └── main.rs    // Point d'entrée pour une crate binaire (ou lib.rs pour une bibliothèque)
```



Construire le Package

Pour compiler un package et générer les binaires ou bibliothèques, utilisez cargo build. Cela compile le package ainsi que ses dépendances et place le résultat dans le dossier target/debug.

cargo build

Pour compiler en mode release (avec des optimisations pour une meilleure performance), utilisez :

cargo build --release



Exécuter et tester le Package

Pour lancer le programme, utilisez `cargo run`. Cela compile automatiquement le code si nécessaire avant de l'exécuter :

`cargo run`

Cargo permet également de gérer les tests intégrés au code grâce à `cargo test`. Cette commande exécute tous les tests définis dans le package.

`cargo test`



Ajouter des dépendances

Les dépendances externes sont gérées dans le fichier Cargo.toml sous la section [dependencies]. Pour ajouter une dépendance, vous pouvez soit l'éditer manuellement, soit utiliser cargo add :

Dans Cargo.toml:

[dependencies]

rand = "0.8" # Ajoute la bibliothèque `rand` pour la génération de nombres aléatoires

Utiliser cargo add: Avec la commande cargo add (disponible avec cargo-edit), vous pouvez ajouter des dépendances sans modifier manuellement Cargo.toml :

cargo add rand

Pour s'assurer que toutes les dépendances du package sont à jour, utilisez :

cargo update

Cela mettra à jour le fichier Cargo.lock avec les versions les plus récentes compatibles avec les contraintes spécifiées dans Cargo.toml.



Documentation

Cargo peut générer une documentation HTML pour le code Rust. Pour créer et ouvrir la documentation locale, utilisez :

cargo doc --open



Crates.io

Publier et utiliser des **crates** de la communauté Rust est un aspect clé du développement en Rust. Avec <https://crates.io>, le dépôt public de crates, vous pouvez facilement partager des packages et bénéficier de milliers de bibliothèques open-source créées par la communauté.

Mieux que des mots, allons voir de quoi il d'agit



Utiliser des Crates de la Communauté

Pour ajouter une crate de la communauté à votre projet :

1. Recherchez la Crate :
 - Allez sur crates.io pour trouver une crate adaptée à vos besoins. Vous y trouverez des informations sur la version, les fonctionnalités et les dépendances de chaque crate.
2. Ajouter la Crate dans Cargo.toml :
 - Ouvrez le fichier Cargo.toml et ajoutez la crate sous [dependencies]. Par exemple, pour ajouter serde, une bibliothèque de sérialisation, utilisez :
3. Utiliser la Crate dans le Code :
 - Importez la crate dans votre code et commencez à l'utiliser. Par exemple, avec serde :
4. Compiler et Exécuter :
 - En utilisant **cargo build** ou **cargo run**, Cargo télécharge automatiquement la crate et ses dépendances, les compile et les inclut dans le projet.

Gestion des Versions et Mise à Jour

- Fixer une Version : Dans Cargo.toml, vous pouvez spécifier une version fixe ou un intervalle de versions. Par exemple, "1.0" signifie n'importe quelle version compatible avec 1.0.x.
- Mise à Jour des Crates : Utilisez cargo update pour obtenir les dernières versions compatibles de vos dépendances.

```
//Dans le toml
[dependencies]
serde = "1.0" # version recommandée

//Dans un fichier.rs
use serde::Serialize;

#[derive(Serialize)]
struct Person {
    name: String,
    age: u8,
}
```



Publier un Crate

Pour partager une crate avec la communauté via crates.io, suivez ces étapes :

Préparer la Crate

1. **Vérifiez Cargo.toml** : Assurez-vous que le fichier Cargo.toml contient toutes les informations nécessaires : nom, version, description, licence, auteurs, etc.
2. **Ajoutez un Fichier README.md** (optionnel mais recommandé).
Le contenu du fichier README.md sera affiché sur la page de votre crate sur crates.io.
3. **Testez et Documentez** : Avant la publication, assurez-vous que votre code est bien testé (cargo test) et que la documentation est à jour (cargo doc).
4. **Créer un Compte sur crates.io**
 - a. Inscrivez-vous sur crates.io.
 - b. Obtenez un Jeton API : Dans vos paramètres de compte crates.io, générez un jeton d'authentification API.
 - c. Connectez Cargo à crates.io : Dans votre terminal, connectez Cargo à crates.io en utilisant ce jeton : **cargo login votre_jeton_api**
5. **Publier la Crate**
 - a. Valider la Crate : Utilisez cargo package pour vérifier que votre crate peut être empaquetée correctement sans erreurs.
 - b. Publier la Crate : Pour publier votre crate, exécutez : **cargo login votre_jeton_api**

```
//Infos Cargo
[package]
name = "ma_crate"
version = "0.1.0"
authors = ["Votre Nom <votre.email@example.com>"]
description = "Une crate utile pour ..."
license = "MIT OR Apache-2.0"
repository = "https://github.com/votre-utilisateur/ma_crate"
```



La bibliothèque standard



Introduction

La bibliothèque standard de Rust, appelée std, fournit un ensemble de modules pour effectuer des opérations essentielles comme l'I/O, la manipulation de chaînes, la gestion des collections, la gestion des erreurs, et bien plus.

Nous allons dans ce chapitre voir une présentation des principaux modules de std qui sont fréquemment utilisés dans le développement en Rust



Module std::io - Entrée et Sortie (I/O)

Le module std::io fournit des outils pour gérer les opérations d'entrée/sortie, comme la lecture et l'écriture de données dans des fichiers ou via le terminal.

- Types principaux :
 - std::io::stdin : pour lire les données depuis l'entrée standard.
 - std::io::stdout et std::io::stderr : pour écrire des données vers la sortie et l'erreur standards.
 - std::io::Read et std::io::Write : traits pour la lecture et l'écriture de données.

```
use std::io::{self, Write};

fn main() {
    let mut input = String::new();
    print!("Entrez quelque chose : ");
    io::stdout().flush().unwrap(); // Nécessaire pour forcer l'affichage immédiat
    io::stdin().read_line(&mut input).expect("Échec de la lecture");
    println!("Vous avez entré : {}", input.trim());
}
```



Module std::fs - Système de Fichiers

Le module std::fs fournit des fonctions pour manipuler les fichiers et répertoires.

- Fonctions principales :
 - std::fs::File : représente un fichier et permet de lire/écrire dans celui-ci.
 - std::fs::read_to_string et std::fs::write : pour lire un fichier en tant que String ou écrire une String dans un fichier.
 - std::fs::create_dir et std::fs::remove_file : pour créer ou supprimer des répertoires et fichiers.

```
fn main() -> Result<(), std::io::Error> {  
    let contenu = std::fs::read_to_string("fichier.txt")?; // Lecture du fichier  
    println!("Contenu du fichier : {}", contenu);  
    Ok(())  
}
```



Module std::string - Manipulation de Chaînes

Le module std::string contient les types String et &str utilisés pour manipuler des chaînes de caractères en Rust.

- Fonctions principales :
 - String::new, String::from : créer des chaînes de caractères.
 - push et push_str : ajouter des caractères ou des chaînes à une String.
 - len, is_empty : obtenir la longueur ou vérifier si la chaîne est vide.

```
fn main() {  
    let mut s = String::from("Bonjour");  
    s.push_str(", monde !");  
    println!("{}", s);  
}
```



Module std::collections - Collections

Le module std::collections propose des structures de données de collections comme les listes, piles, ensembles et tables de hachage.

- Principales collections :
 - Vec<T> : un tableau dynamique.
 - HashMap<K, V> : une table de hachage (associative).
 - HashSet<T> : un ensemble de valeurs uniques.
 - LinkedList<T>, VecDeque<T> : pour les listes chaînées et les files d'attente.

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

fn main() {
    let mut scores = HashMap::new();
    scores.insert("Alice", 50);
    scores.insert("Bob", 40);

    for (name, score) in &scores {
        println!("{}", name, score);
    }
}
```



Module std::cmp - Comparer et Ordonner

Le module std::cmp contient des traits et des fonctions pour gérer les comparaisons entre valeurs.

- Fonctions principales :
 - std::cmp::min, std::cmp::max : obtenir le minimum ou le maximum de deux valeurs.
 - Traits PartialEq, Eq, PartialOrd, et Ord : pour définir l'égalité et l'ordre entre types.

```
use std::cmp::Ordering;

fn compare_numbers(a: i32, b: i32) {
    match a.cmp(&b) {
        Ordering::Less => println!("{}", a, b),
        Ordering::Greater => println!("{}", a, b),
        Ordering::Equal => println!("{}", a, b),
    }
}

fn main() {
    compare_numbers(4, 56);
}
```



Module std::result - Gestion des Résultats et Erreurs

Le module std::result contient le type Result, utilisé pour gérer les opérations pouvant échouer.

- Types principaux :
 - Result<T, E> : représente un résultat qui est soit Ok(T) en cas de succès, soit Err(E) en cas d'erreur.
 - Option<T> : pour gérer les valeurs facultatives (Some ou None).

```
fn division(a: f64, b: f64) -> Result<f64, &'static str> {  
    if b == 0.0 {  
        Err("division par zéro")  
    } else {  
        Ok(a / b)  
    }  
}  
  
fn main() {  
    match division(4.0, 2.0) {  
        Ok(result) => println!("Résultat : {}", result),  
        Err(e) => println!("Erreur : {}", e),  
    }  
}
```



Module std::time - Manipulation du Temps

Le module std::time permet de travailler avec le temps et les durées.

- Types principaux :
 - std::time::Duration : représente une durée en secondes ou nanosecondes.
 - std::time::Instant : un point dans le temps pour mesurer des intervalles.

```
use std::time::{Instant, Duration};
use std::thread::sleep;

fn main() {
    let start = Instant::now();
    sleep(Duration::from_secs(2));
    println!("Temps écoulé : {:?}", start.elapsed());
}
```



Module std::thread - Programmation Concurrente

Le module std::thread fournit des outils pour créer et gérer des threads en Rust.

- Fonctions principales :
 - std::thread::spawn : pour créer un nouveau thread.
 - std::thread::sleep : pour mettre en pause un thread.

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..5 {
            println!("Thread secondaire : {}", i);
            thread::sleep(Duration::from_millis(500));
        }
    });

    handle.join().unwrap();
    println!("Thread principal terminé");
}
```



Programmation concurrentielle & asynchrone



Introduction

La programmation concurrentielle et asynchrone en Rust repose sur l'idée d'exécuter plusieurs tâches simultanément ou de manière non-bloquante pour optimiser l'efficacité des programmes, notamment ceux qui nécessitent des opérations d'I/O ou des calculs intensifs.

Rust propose plusieurs outils et bibliothèques pour mettre en place des fonctionnalités concurrentes et asynchrones tout en garantissant la sécurité des accès mémoire.

Nous allons donc ici parler de :

- Concurrency
- Asynchronicité
- Threads
- Async/await
- Future



La programmation concurrente

La programmation concurrente consiste à exécuter plusieurs tâches ou opérations simultanément, en alternant entre elles rapidement, pour donner l'illusion qu'elles s'exécutent en parallèle.

Elle permet ainsi d'optimiser l'utilisation des ressources et d'améliorer les performances des applications.

- Principe : En concurrence, les tâches peuvent être démarrées indépendamment les unes des autres. Elles peuvent progresser de manière entrelacée (parfois sur un seul cœur de processeur ou en utilisant plusieurs cœurs).
- Objectif : Réduire le temps d'attente en entrelaçant les exécutions de plusieurs tâches.
- Exemple : Dans un programme concurrent, la lecture de fichiers, le calcul et l'affichage des données peuvent être traités de manière simultanée.



La programmation asynchrone

La programmation asynchrone est un modèle d'exécution qui permet à une tâche de s'exécuter sans bloquer le programme principal, ce qui est particulièrement utile pour les opérations d'I/O (input/output) longues, comme les appels réseau ou la lecture de fichiers.

- Principe : Une tâche asynchrone est initiée et s'exécute en arrière-plan jusqu'à ce qu'elle soit prête à reprendre. Elle ne bloque pas le programme pendant qu'elle attend la fin d'une opération.
- Objectif : Améliorer la réactivité du programme en permettant au programme principal de continuer son exécution pendant que les tâches d'I/O s'exécutent en parallèle.
- Exemple : Dans une application web, une requête pour obtenir des données peut être initiée de manière asynchrone, permettant au programme de continuer d'afficher l'interface utilisateur.



Le Thread

Un thread (fil d'exécution en français) est l'unité de base d'exécution d'un programme, permettant de diviser une tâche en sous-tâches qui s'exécutent en parallèle dans un même programme.

- Principe : Un programme peut créer plusieurs threads, chacun ayant sa propre séquence d'instructions, mais partageant l'espace mémoire du programme. Les threads peuvent être gérés de manière indépendante ou synchronisée, selon les besoins du programme.
- Objectif : Exploiter pleinement les capacités des processeurs multicœurs en exécutant plusieurs threads en parallèle.
- Exemple : Dans un éditeur de texte, un thread pourrait gérer l'affichage de l'interface utilisateur, tandis qu'un autre thread pourrait vérifier l'orthographe en arrière-plan.



Résumé

Concurrent : Mode où plusieurs tâches sont exécutées "en parallèle" pour optimiser les ressources.

Asynchrone : Exécution de tâches sans bloquer le flux principal, idéal pour l'I/O.

Threads : Unités d'exécution qui permettent un parallélisme réel sur des processeurs multicœurs, souvent utilisés dans les programmations concurrente et asynchrone.



Création de Thread

La création et la gestion de threads en Rust permettent d'exécuter des tâches en parallèle, ce qui est particulièrement utile pour les programmes nécessitant des opérations intensives ou des traitements simultanés.

Création de Threads avec `std::thread`

Le module `std::thread` de Rust permet de créer et de gérer des threads.

Pour créer un thread, on utilise la fonction `thread::spawn` qui prend une fermeture (closure) comme argument. Le code dans cette fermeture sera exécuté dans le thread nouvellement créé.

Dans cet exemple :

- `thread::spawn` crée un nouveau thread pour exécuter la boucle `for` du thread secondaire.
- `join()` est utilisé pour attendre la fin de ce thread. Sans `join()`, le thread principal pourrait se terminer avant le thread secondaire.

```
use std::thread;
use std::time::Duration;

fn main() {
    let handle = thread::spawn(|| {
        for i in 1..5 {
            println!("Thread secondaire : itération {}", i);
            thread::sleep(Duration::from_millis(500));
        }
    });

    // Code du thread principal
    for i in 1..5 {
        println!("Thread principal : itération {}", i);
        thread::sleep(Duration::from_millis(500));
    }

    // Attendre que le thread secondaire termine
    handle.join().unwrap();
}
```



Partage de Données entre Threads

Les threads en Rust possèdent leurs propres espaces mémoire. Le partage de données entre threads doit être fait de manière sûre pour éviter les erreurs de concurrence (par exemple, les conditions de course).

Rust propose deux outils principaux pour partager des données de manière sûre entre threads :

- Arc (Atomic Reference Counted) : permet de partager des données en lecture seule.
- Mutex : permet de protéger les données partagées pour un accès en écriture et en lecture de manière exclusive.

Dans cet exemple :

- *Arc::clone(&counter) : crée une référence atomique supplémentaire pour permettre à chaque thread d'accéder au compteur.*
- *Mutex::lock : verrouille le compteur pendant la modification pour éviter que plusieurs threads n'accèdent et ne modifient la valeur en même temps.*

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

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }

    println!("Valeur finale du compteur : {}", *counter.lock().unwrap());
}
```



Définir et Gérer la Durée de Vie des Threads

Rust s'assure que les variables utilisées dans un thread sont **static** ou vivront assez longtemps pour éviter que des références invalides n'existent une fois le thread terminé.

Pour passer des données par valeur au thread (via move), on peut transférer la possession de ces données dans le thread, ce qui garantit que le thread principal n'accédera plus aux données une fois qu'elles seront utilisées dans le thread secondaire.

Dans cet exemple, le mot-clé move est utilisé pour transférer la possession de data au thread secondaire, ce qui garantit que le thread principal n'aura plus accès à data après ce transfert.

```
use std::thread;

fn main() {
    let data = vec![1, 2, 3];

    let handle = thread::spawn(move || {
        println!("Voici le vecteur : {:?}", data);
    });

    handle.join().unwrap();
}
```



Délai et Pauses : `thread::sleep`

On peut mettre en pause un thread avec `thread::sleep`, ce qui est utile pour simuler des délais ou des périodes d'attente.

```
use std::thread;
use std::time::Duration;

fn main() {
    thread::spawn(|| {
        thread::sleep(Duration::from_secs(2));
        println!("Thread secondaire réveillé !");
    });

    println!("Thread principal en cours...");
    thread::sleep(Duration::from_secs(1));
    println!("Thread principal terminé.");
}
```



Exécuter Plusieurs Threads Simultanément

En Rust, il est courant de créer un pool de threads ou plusieurs threads exécutant des tâches similaires. Voici comment lancer plusieurs threads qui effectuent chacun un calcul.

```
use std::thread;

fn main() {
    let mut handles = vec![];

    for i in 0..5 {
        let handle = thread::spawn(move || {
            println!("Calcul du thread {}", i);
            i * i // Effectue un calcul (ici le carré de `i`)
        });
        handles.push(handle);
    }

    for handle in handles {
        let result = handle.join().unwrap();
        println!("Résultat du thread : {}", result);
    }
}
```



Erreurs et Gestion avec Result

La méthode `join()` renvoie un `Result`, car un thread peut rencontrer des erreurs (comme une panique).

En cas d'erreur, `join()` renverra un `Err`, qu'il est possible de gérer.

```
use std::thread;

fn main() {
    let handle = thread::spawn(|| {
        panic!("Erreur dans le thread secondaire !");
    });

    match handle.join() {
        Ok(_) => println!("Thread terminé avec succès."),
        Err(e) => println!("Erreur détectée : {:?}", e),
    }
}
```



Les canaux ou Channel de Threads

Les canaux permettent la communication sécurisée entre threads en transférant des messages d'un thread à un autre.

Ce modèle est souvent privilégié pour éviter la synchronisation compliquée des données partagées et pour promouvoir l'isolation entre les threads, ce qui rend le code plus sûr et moins sujet aux erreurs de concurrence.

Un canal en Rust se compose de deux parties :

- Un expéditeur (sender), qui envoie les messages.
- Un récepteur (receiver), qui reçoit les messages.

Rust fournit un module `std::sync::mpsc` (multi-producer, single-consumer), ce qui signifie que plusieurs threads peuvent envoyer des messages à un même récepteur. Si le récepteur est déconnecté ou s'il n'y a plus de messages, le canal se ferme automatiquement.



Création d'un Canal

Pour créer un canal en Rust

On utilise la fonction `mpsc::channel()` qui retourne un tuple contenant un expéditeur et un récepteur.

Envoi et Réception de Messages

Pour envoyer un message, on utilise la méthode `send` sur l'expéditeur (`sender`). Cette méthode renvoie un `Result` indiquant si l'envoi a réussi.

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

fn main() {
    // Création d'un canal
    let (sender, receiver) = mpsc::channel();

    // Lancer un thread pour envoyer un message
    thread::spawn(move || {
        let message = String::from("Bonjour depuis le thread secondaire !");
        sender.send(message).unwrap();
    });

    // Recevoir le message dans le thread principal
    let received = receiver.recv().unwrap();
    println!("Message reçu : {}", received);
}
```



Exemple avec Plusieurs Messages

Pour créer un canal en Rust, on utilise la fonction `mpsc::channel()` qui retourne un tuple contenant un expéditeur et un récepteur.

Ici:

- Le thread secondaire envoie chaque élément du vecteur messages avec une pause.
- Dans le thread principal, la boucle `for` sur le `receiver` s'assure que chaque message est reçu dès qu'il est disponible.

```
fn main() {
    let (sender, receiver) = mpsc::channel();

    thread::spawn(move || {
        let messages = vec![
            String::from("Bonjour"),
            String::from("depuis"),
            String::from("le thread"),
            String::from("secondaire !"),
        ];

        for message in messages {
            sender.send(message).unwrap();
            thread::sleep(Duration::from_millis(500));
        }
    });

    // Boucle pour recevoir chaque message envoyé
    for received in receiver {
        println!("Message reçu : {}", received);
    }
}
```



Utilisation de plusieurs expéditeurs

Avec mpsc, plusieurs expéditeurs peuvent envoyer des messages au même récepteur en clonant l'expéditeur.

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

fn main() {
    let (sender, receiver) = mpsc::channel();

    // Créer plusieurs threads expéditeurs
    for i in 0..5 {
        let sender_clone = sender.clone();
        thread::spawn(move || {
            let message = format!("Message du thread {}", i);
            sender_clone.send(message).unwrap();
        });
    }

    // Recevoir tous les messages
    for received in receiver.iter().take(5) {
        println!("Message reçu : {}", received);
    }
}
```



La programmation asynchrone

La programmation asynchrone permet d'exécuter des opérations sans bloquer le fil principal d'exécution, particulièrement utile pour les tâches d'I/O comme les appels réseau ou la lecture de fichiers.

Cela permet au programme de gérer d'autres tâches pendant qu'il attend la fin d'une opération asynchrone, rendant les applications plus réactives et efficaces.

Principe de la Programmation Asynchrone en Rust

Rust utilise le mot-clé `async` pour définir des fonctions ou des blocs asynchrones, et `await` pour spécifier les points où l'exécution attendra que la tâche soit terminée. La combinaison de `async` et `await` permet de définir des opérations non bloquantes.

Fonctionnement général :

- **Fonction asynchrone** : Une fonction asynchrone en Rust, déclarée avec `async fn`, retourne une future (ou "futur"), une abstraction représentant une valeur ou une opération qui sera disponible dans le futur.
- **await** : `await` permet de suspendre l'exécution d'une fonction asynchrone jusqu'à ce que la future soit prête (c'est-à-dire que la valeur est disponible).



La programmation asynchrone

Futures : L'Unité de Base de l'Asynchronie

En Rust, une future est un type implémentant le trait Future. Lorsqu'on exécute une future, elle représente une tâche asynchrone qui progresse jusqu'à la complétion.

- Lazy Evaluation : Une future en Rust est "paresseuse" (lazy), c'est-à-dire qu'elle ne démarre pas avant d'être attendue (await).
- Modularité : Les futures permettent de composer des opérations asynchrones et de construire des chaînes d'exécutions non bloquantes.



Écrire une Fonction Asynchrone

Une fonction asynchrone est définie en utilisant `async fn` et retourne une future sans être bloquante.

Dans cet exemple, `say_hello` est une fonction asynchrone qui, une fois attendue (`await`), affiche un message. Pour réellement exécuter cette fonction, un runtime asynchrone est nécessaire.

```
async fn say_hello() {  
  println!("Bonjour depuis une fonction asynchrone !");  
}  
  
fn main() {  
  // Créer une exécution pour la fonction asynchrone  
  let future = say_hello();  
  // Lancer la future (nécessite un runtime asynchrone comme tokio ou async-std)  
}
```



Runtime Asynchrone : Tokio et Async-Std

Rust ne fournit pas de runtime asynchrone intégré dans sa bibliothèque standard. Pour exécuter des fonctions asynchrones, on utilise généralement des bibliothèques comme :

- Tokio : Très performant et largement utilisé, particulièrement adapté aux applications réseau.
- Async-Std : Suit l'API standard de Rust, en se concentrant sur la simplicité d'utilisation.

```
use tokio::time::{sleep, Duration};

#[tokio::main]
async fn main() {
    println!("Début du programme asynchrone");

    // Appel d'une fonction asynchrone
    async_fn().await;

    println!("Fin du programme asynchrone");
}

async fn async_fn() {
    println!("Opération en cours...");
    sleep(Duration::from_secs(2)).await;
    println!("Opération terminée après une pause de 2 secondes");
}
```



Combinaison de Tâches Asynchrones avec `join!` et `select!`

Rust permet de combiner plusieurs tâches asynchrones :

- `join!` : Permet d'attendre plusieurs futures en parallèle, utiles pour les tâches indépendantes.
- `select!` : Attendre la première future qui se termine (utile dans les cas où plusieurs futures sont lancées mais on ne sait pas laquelle finira en premier).

```
use tokio::time::{sleep, Duration};
use tokio::join;

async fn operation1() {
    sleep(Duration::from_secs(2)).await;
    println!("Opération 1 terminée");
}

async fn operation2() {
    sleep(Duration::from_secs(1)).await;
    println!("Opération 2 terminée");
}

#[tokio::main]
async fn main() {
    join!(operation1(), operation2());
    println!("Les deux opérations sont terminées");
}
```



La programmation asynchrone

Avantages de l'Asynchrone en Rust

- Gestion efficace des ressources : L'asynchronie permet de gérer plus d'opérations simultanées sans multiplier les threads.
- Modularité accrue : Les futures permettent de construire des opérations non bloquantes complexes et modulaires.
- Sécurité et performance : Rust assure la sécurité mémoire et des performances élevées dans des opérations réseau asynchrones, utiles pour les serveurs web et les applications réseau.

Limites et Précautions

- Runtime requis : Nécessite une bibliothèque comme Tokio ou Async-Std.
- Complexité de gestion des futures : L'écriture d'opérations asynchrones demande une attention à la synchronisation et à la gestion de l'état.
- Durée de vie : Lors du passage de références dans des fonctions asynchrones, il est parfois nécessaire de spécifier explicitement les durées de vie des données (lifetimes).

