

Chapitre 1. Les programmeurs de systèmes peuvent avoir de belles choses

Dans certains contextes, par exemple le contexte ciblé par Rust, être 10 fois plus rapide, voire 2 fois plus rapide que la concurrence, est une chose décisive. Il décide du sort d'un système sur le marché, autant qu'il le ferait sur le marché du matériel.

—[Graydon Hoare](#)

Tous les ordinateurs sont maintenant parallèles...

*La programmation parallèle **est** la programmation.*

—Michael McCool et al., *Programmation parallèle structurée*

Faille de l'analyseur TrueType utilisée par un attaquant d'Etat-nation pour la surveillance ; tous les logiciels sont sensibles à la sécurité.

—[Andy Wingo](#)

Nous avons choisi d'ouvrir notre livre avec les trois citations ci-dessus pour une raison. Mais commençons par un mystère. Que fait le programme C suivant?

```
int main(int argc, char **argv) {
    unsigned long a[1];
    a[3] = 0x7ffff7b36cebUL;
    return 0;
}
```

Sur l'ordinateur portable de Jim ce matin, ce programme a imprimé:

```
undef: Error: .netrc file is readable by others.  
undef: Remove password or make file unreadable by others.
```

Puis il s'est écrasé. Si vous l'essayez sur votre machine, il peut faire autre chose. Qu'est-ce qui se passe?

Le programme est défectueux. Le tableau n'a qu'un seul élément de long, donc l'utilisation est, selon la norme du langage de programmation C, *un comportement indéfini* : a a[3]

Comportement, lors de l'utilisation d'une construction de programme non portable ou erronée ou de données erronées, pour lesquelles la présente Norme internationale n'impose aucune exigence

Un comportement indéfini n'a pas seulement un résultat imprévisible : la norme autorise explicitement le programme à faire n'importe quoi. Dans notre cas, le stockage de cette valeur particulière dans le quatrième élément de ce tableau particulier corrompt la pile d'appels de fonction de sorte que le retour de la fonction, au lieu de quitter le programme gracieusement comme il se doit, saute au milieu du code de la bibliothèque C standard pour récupérer un mot de passe à partir d'un fichier dans le répertoire de base de l'utilisateur. Ça ne se passe pas bien. main

C et C++ ont des centaines de règles pour éviter un comportement indéfini. Ils sont surtout du bon sens: n'accédez pas à la mémoire que vous ne devriez pas, ne laissez pas les opérations arithmétiques déborder, ne divisez pas par zéro, etc. Mais le compilateur n'applique pas ces règles ; il n'a aucune obligation de détecter des violations, même flagrantes. En effet, le programme précédent compile sans erreurs ni avertissements. La responsabilité d'éviter un comportement indéfini incombe entièrement à vous, le programmeur.

Empiriquement parlant, nous, les programmeurs, n'avons pas un excellent bilan à cet égard. Alors qu'il était étudiant à l'Université de l'Utah,

le chercheur Peng Li a modifié les compilateurs C et C ++ pour que les programmes qu'ils traduisaient indiquent s'ils ont exécuté certaines formes de comportement indéfini. Il a constaté que presque tous les programmes le font, y compris ceux de projets très respectés qui maintiennent leur code à des normes élevées. Supposer que vous pouvez éviter un comportement indéfini en C et C ++, c'est comme supposer que vous pouvez gagner une partie d'échecs simplement parce que vous connaissez les règles.

Le message étrange occasionnel ou le crash peut être un problème de qualité, mais un comportement indéfini par inadvertance a également été une cause majeure de failles de sécurité depuis que le ver Morris de 1988 a utilisé une variante de la technique montrée précédemment pour se propager d'un ordinateur à un autre sur les débuts d'Internet.

C et C++ mettent donc les programmeurs dans une position délicate : ces langages sont les normes de l'industrie pour la programmation de systèmes, mais les exigences qu'ils imposent aux programmeurs garantissent pratiquement un flux constant de plantages et de problèmes de sécurité. Répondre à notre mystère soulève simplement une question plus importante : ne pouvons-nous pas faire mieux ?

Rust supporte la charge pour vous

Notre réponse est encadrée par nos trois citations d'ouverture. La troisième citation fait référence à des rapports selon lesquels Stuxnet, un ver informatique trouvé s'introduisant dans des équipements de contrôle industriel en 2010, a pris le contrôle des ordinateurs des victimes en utilisant, entre autres techniques, un comportement indéfini dans le code qui analysait les polices TrueType intégrées dans les documents de traitement de texte. Il y a fort à parier que les auteurs de ce code ne s'attendaient pas à ce qu'il soit utilisé de cette façon, illustrant que ce ne sont pas seulement les systèmes d'exploitation et les serveurs

qui doivent se soucier de la sécurité: tout logiciel qui pourrait gérer des données provenant d'une source non fiable pourrait être la cible d'un exploit.

Le langage Rust vous fait une promesse simple : si votre programme passe les vérifications du compilateur, il est exempt de comportement indéfini. Les pointeurs pendants, les doubles libres et les déréférencements de pointeur nul sont tous détectés au moment de la compilation. Les références de tableau sont sécurisées avec un mélange de vérifications au moment de la compilation et de l'exécution, de sorte qu'il n'y a pas de dépassements de tampon: l'équivalent Rust de notre malheureux programme C se ferme en toute sécurité avec un message d'erreur.

De plus, Rust vise à être à la fois *sûr* et *agréable à utiliser*. Afin de donner des garanties plus solides sur le comportement de votre programme, Rust impose plus de restrictions sur votre code que C et C++, et ces restrictions nécessitent de la pratique et de l'expérience pour s'y habituer. Mais le langage dans son ensemble est flexible et expressif. Ceci est attesté par l'étendue du code écrit dans Rust et la gamme de domaines d'application auxquels il est appliqué.

D'après notre expérience, être capable de faire confiance à la langue pour attraper plus d'erreurs nous encourage à essayer des projets plus ambitieux. La modification de programmes volumineux et complexes est moins risquée lorsque vous savez que les problèmes de gestion de la mémoire et de validité du pointeur sont résolus. Et le débogage est beaucoup plus simple lorsque les conséquences potentielles d'un bogue n'incluent pas la corruption de parties non liées de votre programme.

Bien sûr, il y a encore beaucoup de bugs que Rust ne peut pas détecter. Mais dans la pratique, retirer un comportement indéfini de la table change considérablement le caractère du développement pour le mieux.

La programmation parallèle est apprivoisée

La concurrence d'accès est notoirement difficile à utiliser correctement en C et C++. Les développeurs se tournent généralement vers la concurrence uniquement lorsque le code monothread s'est avéré incapable d'atteindre les performances dont ils ont besoin. Mais la deuxième citation d'ouverture soutient que le parallélisme est trop important pour les machines modernes pour être traité comme une méthode de dernier recours.

Il s'avère que les mêmes restrictions qui garantissent la sécurité de la mémoire dans Rust garantissent également que les programmes Rust sont exempts de courses de données. Vous pouvez partager des données librement entre les threads, tant qu'elles ne changent pas. Les données qui changent ne sont accessibles qu'à l'aide de primitives de synchronisation. Tous les outils d'accès concurrentiel traditionnels sont disponibles : mutex, variables de condition, canaux, atomes, etc. Rust vérifie simplement que vous les utilisez correctement.

Cela fait de Rust un excellent langage pour exploiter les capacités des machines multicœurs modernes. L'écosystème Rust offre des bibliothèques qui vont au-delà des primitives d'accès concurrentiel habituelles et vous aident à répartir uniformément les charges complexes entre les pools de processeurs, à utiliser des mécanismes de synchronisation sans verrouillage tels que Lecture-Copie-Mise à jour, etc.

Et pourtant, la rouille est toujours rapide

Ceci, enfin, est notre première citation d'ouverture. Rust partage les ambitions que Bjarne Stroustrup articule pour C++ dans son article « Ab-

straction and the C++ Machine Model » :

En général, les implémentations C++ obéissent au principe zéro-surcharge : ce que vous n'utilisez pas, vous ne le payez pas. Et plus loin : ce que vous utilisez, vous ne pourriez pas mieux coder à la main.

La programmation des systèmes consiste souvent à pousser la machine à ses limites. Pour les jeux vidéo, toute la machine doit être consacrée à la création de la meilleure expérience pour le joueur. Pour les navigateurs Web, l'efficacité du navigateur fixe le plafond de ce que les auteurs de contenu peuvent faire. Dans les limites inhérentes à la machine, autant d'attention que possible sur la mémoire et le processeur doit être laissée au contenu lui-même. Le même principe s'applique aux systèmes d'exploitation : le noyau doit mettre les ressources de la machine à la disposition des programmes utilisateurs, et non les consommer lui-même.

Mais quand nous disons que Rust est « rapide », qu'est-ce que cela signifie vraiment ? On peut écrire du code lent dans n'importe quel langage à usage général. Il serait plus précis de dire que, si vous êtes prêt à faire l'investissement pour concevoir votre programme afin de tirer le meilleur parti des capacités de la machine sous-jacente, Rust vous soutient dans cet effort. Le langage est conçu avec des valeurs par défaut efficaces et vous donne la possibilité de contrôler comment la mémoire est utilisée et comment l'attention du processeur est dépensée.

Rust facilite la collaboration

Nous avons caché une quatrième citation dans le titre de ce chapitre : « Les programmeurs de systèmes peuvent avoir de belles choses. » Cela fait référence à la prise en charge par Rust du partage et de la réutilisation du code.

Le gestionnaire de paquets et l'outil de construction de Rust, Cargo, facilitent l'utilisation des bibliothèques publiées par d'autres sur le référentiel public de paquets de Rust, le site Web crates.io. Il vous suffit d'ajouter le nom de la bibliothèque et le numéro de version requis à un fichier, et Cargo se charge de télécharger la bibliothèque, ainsi que toutes les autres bibliothèques qu'il utilise à son tour, et de lier le tout ensemble. Vous pouvez considérer Cargo comme la réponse de Rust à NPM ou RubyGems, en mettant l'accent sur la gestion des versions et les versions reproductibles. Il existe des bibliothèques Rust populaires fournissant tout, de la sérialisation standard aux clients et serveurs HTTP et aux API graphiques modernes.

Pour aller plus loin, le langage lui-même est également conçu pour prendre en charge la collaboration : les traits et les génériques de Rust vous permettent de créer des bibliothèques avec des interfaces flexibles afin qu'elles puissent servir dans de nombreux contextes différents. Et la bibliothèque standard de Rust fournit un ensemble de types fondamentaux qui établissent des conventions partagées pour des cas communs, ce qui rend différentes bibliothèques plus faciles à utiliser ensemble.

Le chapitre suivant vise à concrétiser les revendications générales que nous avons faites dans ce chapitre, avec une visite de plusieurs petits programmes Rust qui montrent les forces de la langue.

Chapitre 2. Un tour de Rust

Rust présente aux auteurs d'un livre comme celui-ci un défi: ce qui donne à la langue son caractère n'est pas une caractéristique spécifique et étonnante que nous pouvons montrer sur la première page, mais plutôt la façon dont toutes ses parties sont conçues pour fonctionner ensemble en douceur au service des objectifs que nous avons exposés dans le dernier chapitre: programmation de systèmes sûrs et performants. Chaque partie de la langue est mieux justifiée dans le contexte de tout le reste.

Ainsi, plutôt que d'aborder une fonctionnalité de langue à la fois, nous avons préparé une visite de quelques programmes petits mais complets, dont chacun introduit d'autres fonctionnalités de la langue, dans le contexte:

- En guise d'échauffement, nous avons un programme qui effectue un calcul simple sur ses arguments de ligne de commande, avec des tests unitaires. Cela montre les types de base de Rust et introduit des *traits*.
- Ensuite, nous construisons un serveur Web. Nous utiliserons une bibliothèque tierce pour gérer les détails de HTTP et introduire la gestion des chaînes, les fermetures et la gestion des erreurs.
- Notre troisième programme trace une belle fractale, répartissant le calcul sur plusieurs threads pour la vitesse. Cela inclut un exemple de fonction générique, illustre comment gérer quelque chose comme un tampon de pixels et montre la prise en charge de la concurrence par Rust.
- Enfin, nous montrons un outil de ligne de commande robuste qui traite les fichiers à l'aide d'expressions régulières. Cela présente les fonctionnalités de la bibliothèque standard Rust pour travailler avec des fichiers et la bibliothèque d'expressions régulières tierce la plus couramment utilisée.

La promesse de Rust d'empêcher les comportements indéfinis avec un impact minimal sur les performances influence la conception de chaque partie du système, des structures de données standard telles que les vecteurs et les chaînes à la façon dont les programmes Rust utilisent des bibliothèques tierces. Les détails de la façon dont cela est géré sont couverts tout au long du livre. Mais pour l'instant, nous voulons vous montrer que Rust est un langage capable et agréable à utiliser.

Tout d'abord, bien sûr, vous devez installer Rust sur votre ordinateur.

rustup et Cargo

La meilleure façon d'installer Rust est d'utiliser . Allez dans <https://rust-up.rs> et suivez les instructions qui s'y trouvent. rustup

Vous pouvez également accéder au [site Web de Rust](#) pour obtenir des packages prédefinis pour Linux, macOS et Windows. Rust est également inclus dans certaines distributions de système d'exploitation. Nous préférons parce que c'est un outil pour gérer les installations Rust, comme RVM pour Ruby ou NVM pour Node. Par exemple, lorsqu'une nouvelle version de Rust est publiée, vous pourrez effectuer une mise à niveau sans aucun clic en tapant `rustup rustup update`

Dans tous les cas, une fois l'installation terminée, vous devriez avoir trois nouvelles commandes disponibles sur votre ligne de commande:

```
$ cargo --version
cargo 1.49.0 (d00d64df9 2020-12-05)
$ rustc --version
rustc 1.49.0 (e1884a8e3 2020-12-29)
$ rustdoc --version
rustdoc 1.49.0 (e1884a8e3 2020-12-29)
```

Ici, l'invite de commande est là; sur Windows, ce serait ou quelque chose de similaire. Dans cette transcription, nous exécutons les trois commandes que nous avons installées, en demandant à chacune de signaler de quelle version il s'agit. Prendre chaque commande à tour de rôle : \$ c:\>

- `cargo` est le gestionnaire de compilation, le gestionnaire de paquets et l'outil général de Rust. Vous pouvez utiliser Cargo pour démarrer un nouveau projet, générer et exécuter votre programme et gérer toutes les bibliothèques externes dont dépend votre code.
- `rustc` est le compilateur Rust. Habituellement, nous laissons Cargo invoquer le compilateur pour nous, mais parfois il est utile de l'exécuter directement.
- `rustdoc` est l'outil de documentation Rust. Si vous écrivez de la documentation dans des commentaires de la forme appropriée dans le code source de votre programme, vous pouvez créer du HTML bien formaté à partir d'eux. Comme , nous laissons généralement Cargo courir pour nous. `rustdoc rustc rustdoc`

Pour plus de commodité, Cargo peut créer un nouveau package Rust pour nous, avec des métadonnées standard organisées de manière appropriée :

```
$ cargo new hello
Created binary (application) `hello` package
```

Cette commande crée un nouveau répertoire de package nommé *hello*, prêt à générer un exécutable de ligne de commande.

En regardant à l'intérieur du répertoire de niveau supérieur du package :

```
$ cd hello
$ ls -la
total 24
drwxrwxr-x. 4 jimb jimb 4096 Sep 22 21:09 .
drwx-----. 62 jimb jimb 4096 Sep 22 21:09 ..
drwxrwxr-x. 6 jimb jimb 4096 Sep 22 21:09 .git
-rw-rw-r--. 1 jimb jimb 7 Sep 22 21:09 .gitignore
-rw-rw-r--. 1 jimb jimb 88 Sep 22 21:09 Cargo.toml
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:09 src
```

Nous pouvons voir que Cargo a créé un fichier *Cargo.toml* pour contenir les métadonnées du package. Pour le moment, ce fichier ne contient pas grand-chose:

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

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

[dependencies]
```

Si jamais notre programme acquiert des dépendances avec d'autres bibliothèques, nous pouvons les enregistrer dans ce fichier, et Cargo se chargera de télécharger, de construire et de mettre à jour ces bibliothèques pour nous. Nous couvrirons le fichier *Cargo.toml* en détail au [chapitre 8](#).

Cargo a configuré notre package pour une utilisation avec le système de contrôle de version, en créant un sous-répertoire de métadonnées *.git* et un fichier *.gitignore*. Vous pouvez dire à Cargo de sauter cette étape en passant à sur la ligne de commande `git --vcs none cargo new`

Le sous-répertoire *src* contient le code Rust réel :

```
$ cd src
$ ls -l
total 4
-rw-rw-r--. 1 jimb jimb 45 Sep 22 21:09 main.rs
```

Il semble que Cargo ait commencé à écrire le programme en notre nom. Le fichier *main.rs* contient le texte suivant :

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

Dans Rust, vous n'avez même pas besoin d'écrire votre propre programme « Hello, World! » Et c'est l'étendue du passe-partout pour un nouveau programme Rust: deux fichiers, totalisant treize lignes.

Nous pouvons appeler la commande à partir de n'importe quel répertoire du package pour construire et exécuter notre programme: `cargo run`

```
$ cargo run
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.28s
      Running `~/home/jimb/rust/hello/target/debug/hello`
Hello, world!
```

Ici, Cargo a appelé le compilateur Rust, puis exécute l'exécutable qu'il a produit. Cargo place l'exécutable dans le sous-répertoire *cible* en haut du package : `rustc`

```
$ ls -l ../target/debug
total 580
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 build
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 deps
drwxrwxr-x. 2 jimb jimb 4096 Sep 22 21:37 examples
-rwxrwxr-x. 1 jimb jimb 576632 Sep 22 21:37 hello
-rw-rw-r--. 1 jimb jimb 198 Sep 22 21:37 hello.d
drwxrwxr-x. 2 jimb jimb 68 Sep 22 21:37 incremental
$ ../target/debug/hello
Hello, world!
```

Lorsque nous avons terminé, Cargo peut nettoyer les fichiers générés pour nous:

```
$ cargo clean
$ ../target/debug/hello
bash: ../target/debug/hello: No such file or directory
```

Fonctions de rouille

La syntaxe de Rust est délibérément peu originale. Si vous êtes familier avec C, C++, Java ou JavaScript, vous pouvez probablement trouver votre chemin à travers la structure générale d'un programme Rust. Voici une fonction qui calcule le plus grand diviseur commun de deux entiers, en utilisant [l'algorithme d'Euclide](#). Vous pouvez ajouter ce code à la fin de `src/main.rs`:

```
fn gcd(mut n: u64, mut m: u64) -> u64 {
    assert!(n != 0 && m != 0);
    while m != 0 {
        if m < n {
```

```

        let t = m;
        m = n;
        n = t;
    }
    m = m % n;
}
n
}

```

Le mot-clé (prononcé « fun ») introduit une fonction. Ici, nous définissons une fonction nommée `gcd`, qui prend deux paramètres `n` et `m`, dont chacun est de type `u64`, un entier 64 bits non signé. Le jeton précède le type de retour : notre fonction renvoie une valeur. L'indentation à quatre espaces est de style Rust standard.

`fn gcd n m u64 -> u64`

Les noms de type entier de machine de Rust reflètent leur taille et leur signature : `i32` est un entier 32 bits signé ; `u8` est un entier 8 bits non signé (utilisé pour les valeurs « octet »), et ainsi de suite. Les types `isize` et `usize` contiennent des entiers signés et non signés de la taille d'un pointeur, 32 bits de long sur les plates-formes 32 bits et 64 bits de long sur les plates-formes 64 bits. Rust a également deux types à virgule flottante, `f32` et `f64`, qui sont les types à virgule flottante IEEE simple et double précision, comme en C et C

`++.i32 u8 isize usize f32 f64 float double`

Par défaut, une fois qu'une variable est initialisée, sa valeur ne peut pas être modifiée, mais placer le mot-clé (prononcé « mute », abréviation de *mutable*) avant les paramètres et permet à notre corps de fonction de leur attribuer. En pratique, la plupart des variables ne sont pas affectées à ; le mot-clé sur ceux qui le font peut être un indice utile lors de la lecture du code.

`mut n m mut`

Le corps de la fonction commence par un appel à la macro, vérifiant qu'aucun des deux arguments n'est nul. Le caractère marque cela comme un appel de macro, et non comme un appel de fonction. Comme la macro en C et C++, Rust vérifie que son argument est vrai et, s'il ne l'est pas, met fin au programme avec un message utile comprenant l'emplacement source de la vérification défaillante ; ce genre de résiliation brutale s'appelle une *panique*. Contrairement à C et C++, dans lesquels les assertions peuvent être ignorées, Rust vérifie toujours les assertions, quelle que soit la façon dont le programme a été compilé. Il existe également une macro, dont les assertions sont ignorées lorsque le programme est compilé pour la vitesse.

`assert! ! assert assert! debug_assert!`

Le cœur de notre fonction est une boucle contenant une instruction et une affectation. Contrairement à C et C++, Rust n'exige pas de parenthèses autour des expressions conditionnelles, mais il nécessite des accolades autour des instructions qu'ils contrôlent.

`while if`

Une instruction déclare une variable locale, comme dans notre fonction. Nous n'avons pas besoin d'écrire le type de 's, tant que Rust peut le déduire de la façon dont la variable est utilisée. Dans notre fonction, le seul type qui fonctionne pour est , matching et . Rust ne déduit que des types dans les corps de fonction: vous devez écrire les types de paramètres de fonction et les valeurs de retour, comme nous l'avons fait auparavant. Si nous voulions épeler le type de ' nous pourrions écrire

```
:let t t t u64 m n t
```

```
let t: u64 = m;
```

Rust a une instruction, mais la fonction n'en a pas besoin. Si un corps de fonction se termine par une expression qui *n'est pas* suivie d'un point-virgule, il s'agit de la valeur de retour de la fonction. En fait, tout bloc entouré d'accolades peut fonctionner comme une expression. Par exemple, il s'agit d'une expression qui imprime un message, puis donne comme valeur:
return gcd x.cos()

```
{
    println!("evaluating cos x");
    x.cos()
}
```

Il est typique dans Rust d'utiliser cette forme pour établir la valeur de la fonction lorsque le contrôle « tombe de la fin » de la fonction, et d'utiliser des instructions uniquement pour les premiers retours explicites du milieu d'une fonction. return

Écriture et exécution de tests unitaires

Rust dispose d'un support simple pour les tests intégrés dans le langage. Pour tester notre fonction, nous pouvons ajouter ce code à la fin de *src/main.rs* : gcd

```
#[test]
fn test_gcd() {
    assert_eq!(gcd(14, 15), 1);

    assert_eq!(gcd(2 * 3 * 5 * 11 * 17,
                  3 * 7 * 11 * 13 * 19),
                  3 * 11);
}
```

Ici, nous définissons une fonction nommée , qui appelle et vérifie qu'elle renvoie des valeurs correctes. Le haut de la définition marque comme une fonction de test, à sauter dans les compilations normales, mais inclus et appelé automatiquement si nous exécutons notre programme avec la

commande. Nous pouvons avoir des fonctions de test dispersées dans notre arborescence source, placées à côté du code qu'elles exercent, et nous les rassemblerons automatiquement et les exécuterons toutes.

```
test_gcd gcd #[test] test_gcd cargo test cargo test
```

Le marqueur est un exemple *d'attribut*. Les attributs sont un système ouvert permettant de marquer des fonctions et d'autres déclarations avec des informations supplémentaires, telles que des attributs en C++ et C#, ou des annotations en Java. Ils sont utilisés pour contrôler les avertissements du compilateur et les vérifications de style de code, inclure le code conditionnellement (comme en C et C++), indiquer à Rust comment interagir avec le code écrit dans d'autres langages, etc. Nous verrons d'autres exemples d'attributs au fur et à mesure.

```
#[test] #ifdef
```

Avec nos définitions ajoutées au package *hello* que nous avons créé au début du chapitre, et notre répertoire actuel quelque part dans la sous-arborescence du package, nous pouvons exécuter les tests comme suit:

```
gcd test_gcd
```

```
$ cargo test
   Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
      Running unitests (/home/jimb/rust/hello/target/debug/deps/hello-2375...
running 1 test
test test_gcd ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Gestion des arguments de ligne de commande

Pour que notre programme prenne une série de nombres comme arguments de ligne de commande et imprime leur plus grand diviseur commun, nous pouvons remplacer la fonction dans *src/main.rs* par ce qui suit:

```
main
```

```
use std::str::FromStr;
use std::env;

fn main() {
    let mut numbers = Vec::new();

    for arg in env::args().skip(1) {
        numbers.push(u64::from_str(&arg)
                     .expect("error parsing argument"));
    }
}
```

```

if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}

let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}

println!("The greatest common divisor of {} is {}",
         numbers, d);
}

```

Il s'agit d'un gros bloc de code, alors prenons-le pièce par pièce:

```

use std::str::FromStr;
use std::env;

```

La première déclaration introduit le *trait* de bibliothèque standard dans la portée. Un trait est un ensemble de méthodes que les types peuvent implémenter. Tout type qui implémente le trait possède une méthode qui tente d'analyser une valeur de ce type à partir d'une chaîne. Le type implémente , et nous allons appeler pour analyser nos arguments de ligne de commande. Bien que nous n'utilisions jamais le nom ailleurs dans le programme, un trait doit être dans la portée afin d'utiliser ses méthodes.

Nous couvrirons les traits en détail dans [le chapitre](#)

[11.](#) use FromStr FromStr from_str u64 FromStr u64::from_str FromStr

La deuxième déclaration apporte le module, qui fournit plusieurs fonctions et types utiles pour interagir avec l'environnement d'exécution, y compris la fonction, qui nous donne accès aux arguments de ligne de commande du programme. use std::env args

Passons maintenant à la fonction du programme : main

```

fn main() {

```

Notre fonction ne renvoie pas de valeur, nous pouvons donc simplement omettre le type et return qui suivrait normalement la liste des paramètres. main ->

```

let mut numbers = Vec::new();

```

Nous déclarons une variable locale mutable et l'initialisons sur un vecteur vide. est le type de vecteur cultivable de Rust, analogue à C++, une liste Python ou un tableau JavaScript. Même si les vecteurs sont conçus pour être cultivés et rétrécis dynamiquement, nous devons toujours mar-

```
quer la variable pour Rust pour nous permettre de pousser les nombres à  
la fin de celle-ci. numbers Vec<std::vector<mut
```

Le type de est , un vecteur de valeurs, mais comme auparavant, nous n'avons pas besoin de l'écrire. Rust va le déduire pour nous, en partie parce que ce que nous poussons sur le vecteur sont des valeurs, mais aussi parce que nous passons les éléments du vecteur à , qui n'accepte que des valeurs. numbers Vec<u64> u64 u64 gcd u64

```
for arg in env::args().skip(1) {
```

Ici, nous utilisons une boucle pour traiter nos arguments de ligne de commande, en définissant la variable à chaque argument à tour de rôle et en évaluant le corps de la boucle. for arg

La fonction du module renvoie un *itérateur*, une valeur qui produit chaque argument à la demande et indique quand nous avons terminé. Les itérateurs sont omniprésents dans Rust; la bibliothèque standard comprend d'autres itérateurs qui produisent les éléments d'un vecteur, les lignes d'un fichier, les messages reçus sur un canal de communication et presque tout ce qui a du sens à boucler. Les itérateurs de Rust sont très efficaces : le compilateur est généralement capable de les traduire dans le même code qu'une boucle manuscrite. Nous montrerons comment cela fonctionne et donnerons des exemples au [chapitre 15](#). std::env args

Au-delà de leur utilisation avec des boucles, les itérateurs incluent une large sélection de méthodes que vous pouvez utiliser directement. Par exemple, la première valeur produite par l'itérateur renvoyé par est toujours le nom du programme en cours d'exécution. Nous voulons ignorer cela, alors nous appelons la méthode de l'itérateur pour produire un nouvel itérateur qui omet cette première valeur. for args skip

```
numbers.push(u64::from_str(&arg)  
    .expect("error parsing argument"));
```

Ici, nous appelons pour tenter d'analyser notre argument de ligne de commande en tant qu'entier 64 bits non signé. Plutôt qu'une méthode que nous appelons sur une valeur que nous avons à portée de main, est une fonction associée au type, semblable à une méthode statique en C ++ ou Java. La fonction ne renvoie pas une valeur directe, mais plutôt une valeur qui indique si l'analyse a réussi ou échoué. Une valeur est l'une des deux variantes suivantes

```
:u64::from_str arg u64 u64::from_str u64 from_str u64 Result  
t Result
```

- Valeur écrite , indiquant que l'analyse a réussi et est la valeur produite ok(v) v

- Valeur écrite , indiquant que l'analyse a échoué et est une valeur d'erreur expliquant pourquoi Err(e) e

Les fonctions qui font tout ce qui pourrait échouer, comme effectuer une entrée ou une sortie ou interagir avec le système d'exploitation, peuvent renvoyer des types dont les variantes portent des résultats réussis (le nombre d'octets transférés, le fichier ouvert, etc.) et dont les variantes portent un code d'erreur indiquant ce qui s'est mal passé. Contrairement à la plupart des langues modernes, Rust n'a pas d'exceptions: toutes les erreurs sont traitées en utilisant l'un ou l'autre ou la panique, comme indiqué dans [le chapitre 7](#). Result Ok Err Result

Nous utilisons la méthode de ' pour vérifier le succès de notre analyse. Si le résultat est un , imprime un message qui inclut une description du programme et le quitte immédiatement. Cependant, si le résultat est , se renvoie simplement lui-même, que nous sommes finalement capables de pousser à la fin de notre vecteur de

nombres. Result expect Err(e) expect e Ok(v) expect v

```
if numbers.len() == 0 {
    eprintln!("Usage: gcd NUMBER ...");
    std::process::exit(1);
}
```

Il n'y a pas de plus grand diviseur commun d'un ensemble vide de nombres, nous vérifions donc que notre vecteur a au moins un élément et quittons le programme avec une erreur si ce n'est pas le cas. Nous utilisons la macro pour écrire notre message d'erreur dans le flux de sortie d'erreur standard. eprintln!

```
let mut d = numbers[0];
for m in &numbers[1..] {
    d = gcd(d, *m);
}
```

Cette boucle utilise comme valeur d'exécution, la mettant à jour pour rester le plus grand diviseur commun de tous les nombres que nous avons traités jusqu'à présent. Comme précédemment, nous devons marquer comme mutable afin que nous puissions l'affecter dans la boucle. d d

La boucle a deux bits surprenants. Tout d'abord, nous avons écrit ; à quoi sert l'opérateur? Deuxièmement, nous avons écrit ; qu'est-ce que c'est? Ces deux détails sont complémentaires l'un de l'autre. for for m in &numbers[1..] & gcd(d, *m) * *m

Jusqu'à présent, notre code n'a fonctionné que sur des valeurs simples comme des entiers qui tiennent dans des blocs de mémoire de taille fixe.

Mais maintenant, nous sommes sur le point d'itérer sur un vecteur, qui pourrait être de n'importe quelle taille, peut-être très grand. Rust est prudent lorsqu'il manipule de telles valeurs : il veut laisser au programmeur le contrôle de la consommation de mémoire, en précisant combien de temps chaque valeur vit, tout en veillant à ce que la mémoire soit libérée rapidement lorsqu'elle n'est plus nécessaire.

Donc, lorsque nous itérons, nous voulons dire à Rust que *la propriété* du vecteur devrait rester avec ; nous empruntons *simplement ses éléments* pour la boucle. L'opérateur `dans` emprunte une *référence* aux éléments du vecteur à partir de la seconde. La boucle itère sur les éléments référencés, permettant d'emprunter chaque élément successivement. L'opérateur `dans les déréférencements`, donnant la valeur à laquelle il se réfère; c'est le prochain que nous voulons passer à . Enfin, puisque possède le vecteur, Rust le libère automatiquement lorsqu'il sort du champ d'application à la fin de

```
. numbers & &numbers[1..] for m * *m m u64 gcd numbers number  
s main
```

Les règles de propriété et de références de Rust sont essentielles à la gestion de la mémoire et à la concurrence d'accès sécurisée de Rust ; nous en discutons en détail au [chapitre 4](#) et son compagnon, [le chapitre 5](#). Vous devrez être à l'aise avec ces règles pour être à l'aise dans Rust, mais pour cette visite d'introduction, tout ce que vous devez savoir est qu'il emprunte une référence à , et c'est la valeur à laquelle la référence se réfère. `&x x *r r`

Poursuivant notre promenade à travers le programme:

```
println!( "The greatest common divisor of {:+?} is {}" ,  
        numbers , d );
```

Après avoir itéré sur les éléments de , le programme imprime les résultats dans le flux de sortie standard. La macro prend une chaîne de modèle, remplace les versions formatées des arguments restants par les formulaires tels qu'ils apparaissent dans la chaîne de modèle et écrit le résultat dans le flux de sortie standard. `numbers println! { ... }`

Contrairement à C et C++, qui nécessitent de retourner zéro si le programme s'est terminé avec succès, ou un état de sortie non nul si quelque chose s'est mal passé, Rust suppose que si le programme revient du tout, le programme s'est terminé avec succès. Ce n'est qu'en appelant explicitement des fonctions comme ou pouvons-nous provoquer l'arrêt du programme avec un code d'état

```
d'erreur. main main expect std::process::exit
```

La commande nous permet de passer des arguments à notre programme, afin que nous puissions essayer notre gestion de ligne de

commande: cargo run

```
$ cargo run 42 56
    Compiling hello v0.1.0 (/home/jimb/rust/hello)
    Finished dev [unoptimized + debuginfo] target(s) in 0.22s
        Running `~/home/jimb/rust/hello/target/debug/hello 42 56`
The greatest common divisor of [42, 56] is 14
$ cargo run 799459 28823 27347
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
        Running `~/home/jimb/rust/hello/target/debug/hello 799459 28823 27347`
The greatest common divisor of [799459, 28823, 27347] is 41
$ cargo run 83
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
        Running `~/home/jimb/rust/hello/target/debug/hello 83`
The greatest common divisor of [83] is 83
$ cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.02s
        Running `~/home/jimb/rust/hello/target/debug/hello`
```

Usage: gcd NUMBER ...

Nous avons utilisé quelques fonctionnalités de la bibliothèque standard de Rust dans cette section. Si vous êtes curieux de savoir ce qui est disponible, nous vous encourageons fortement à essayer la documentation en ligne de Rust. Il dispose d'une fonction de recherche en direct qui facilite l'exploration et inclut même des liens vers le code source. La commande installe automatiquement une copie sur votre ordinateur lorsque vous installez Rust lui-même. Vous pouvez afficher la documentation de la bibliothèque standard sur le [site Web](#) de Rust ou dans votre navigateur avec la commande : rustup

```
$ rustup doc --std
```

Servir des pages sur le Web

L'une des forces de Rust est la collection de paquets de bibliothèque disponibles gratuitement publiés sur le site Web [crates.io](#). La commande permet à votre code d'utiliser facilement un package crates.io : il téléchargera la bonne version du package, le construira et le mettra à jour comme demandé. Un paquet Rust, qu'il s'agisse d'une bibliothèque ou d'un exécutable, est appelé une *caisse*; Cargo et crates.io tirent tous deux leurs noms de ce terme. cargo

Pour montrer comment cela fonctionne, nous allons mettre en place un serveur Web simple en utilisant la caisse du framework Web, la caisse de sérialisation et diverses autres caisses dont elles dépendent. Comme le montre [la figure 2-1](#), notre site Web demandera à l'utilisateur deux nombres et calculera son plus grand diviseur commun. actix-web serde

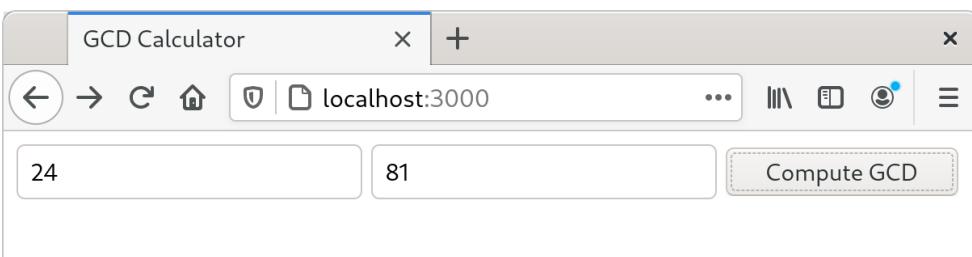


Figure 2-1. Page Web offrant le calcul de GCD

Tout d'abord, nous demanderons à Cargo de créer un nouveau package pour nous, nommé : actix-gcd

```
$ cargo new actix-gcd
     Created binary (application) `actix-gcd` package
$ cd actix-gcd
```

Ensuite, nous allons modifier le fichier *Cargo.toml* de notre nouveau projet pour répertorier les paquets que nous voulons utiliser; son contenu devrait être le suivant:

```
[package]
name = "actix-gcd"
version = "0.1.0"
edition = "2021"

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

[dependencies]
actix-web = "1.0.8"
serde = { version = "1.0", features = [ "derive" ] }
```

Chaque ligne de la section *cargo.toml* donne le nom d'une caisse sur crates.io, et la version de cette caisse que nous aimions utiliser. Dans ce cas, nous voulons la version de la caisse et la version de la caisse. Il se peut qu'il existe des versions de ces caisses sur crates.io plus récentes que celles montrées ici, mais en nommant les versions spécifiques contre lesquelles nous avons testé ce code, nous pouvons nous assurer que le code continuera à se compiler même si de nouvelles versions des paquets sont publiées. Nous aborderons la gestion des versions plus en détail au [chapitre 8](#).

[dependencies] 1.0.8 actix-web 1.0 serde

Les caisses peuvent avoir des fonctionnalités optionnelles: des parties de l'interface ou de l'implémentation dont tous les utilisateurs n'ont pas besoin, mais qui ont néanmoins du sens à inclure dans cette caisse. La caisse offre un moyen merveilleusement laconique de gérer les données des formulaires Web, mais selon la documentation de , elle n'est disponible que si nous sélectionnons la fonctionnalité de la caisse, nous l'avons donc demandée dans notre fichier *Cargo.toml* comme indiqué. `serde` `serde derive`

Notez que nous n'avons qu'à nommer les caisses que nous utiliserons directement; prend soin d'apporter toutes les autres caisses dont ils ont besoin à leur tour. cargo

Pour notre première itération, nous garderons le serveur Web simple: il ne servira que la page qui invite l'utilisateur à calculer des nombres.

Dans *actix-gcd/src/main.rs*, nous allons placer le texte suivant :

```
use actix_web::{web, App, HttpResponse, HttpServer};

fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000")
        .expect("error binding server to address")
        .run()
        .expect("error running server");
}

fn get_index() -> HttpResponse {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(
            r#"
                <title>GCD Calculator</title>
                <form action="/gcd" method="post">
                    <input type="text" name="n"/>
                    <input type="text" name="m"/>
                    <button type="submit">Compute GCD</button>
                </form>
            "#,
        )
}
```

Nous commençons par une déclaration pour rendre certaines des définitions de la caisse plus faciles à obtenir. Lorsque nous écrivons , chacun des noms énumérés entre parenthèses devient directement utilisable dans notre code; au lieu d'avoir à épeler le nom complet chaque fois que nous l'utilisons, nous pouvons simplement nous y référer comme . (Nous ironnons à la caisse dans un instant.) use actix-web use actix_web:: { ... } actix_web::HttpResponse HttpResponse serde

Notre fonction est simple : elle appelle à créer un serveur qui répond aux demandes pour un seul chemin, ; imprime un message nous rappelant comment nous y connecter; , puis définit l'écoute sur le port TCP 3000 sur l'ordinateur local. main HttpServer::new "/"

L'argument auquel nous passons est l'expression de *fermeture* de Rust . Une fermeture est une valeur qui peut être appelée comme s'il s'agissait d'une fonction. Cette fermeture ne prend aucun argument, mais si c'était le cas, leurs noms apparaîtraient entre les barres verticales. Le est le corps de la fermeture. Lorsque nous démarrons notre serveur, Actix démarre un pool de threads pour gérer les demandes entrantes. Chaque thread appelle notre fermeture pour obtenir une nouvelle copie de la valeur qui lui indique comment acheminer et gérer les demandes.

```
HttpServer::new || { App::new() ... } || { ... }
```

```
} App
```

La fermeture appelle à créer un nouveau, vide, puis appelle sa méthode pour ajouter un itinéraire unique pour le chemin d'accès . Le gestionnaire fourni pour cet itinéraire , traite les requêtes HTTP en appelant la fonction . La méthode renvoie la même chose qu'elle a été appelée, maintenant améliorée avec le nouvel itinéraire. Comme il n'y a pas de point-virgule à la fin du corps de la fermeture, la valeur de retour de la fermeture est prête à être utilisée par le

```
fil. App::new App route "/" web::get().to(get_index) GET get_index route App App HttpServer
```

La fonction génère une valeur représentant la réponse à une requête HTTP. représente un état HTTP, indiquant que la requête a réussi. Nous appelons ses méthodes et méthodes pour remplir les détails de la réponse; chaque appel renvoie celui auquel il a été appliqué, avec les modifications apportées. Enfin, la valeur renvoyée de sert de valeur de retour à .

```
get_index HttpResponse GET / HttpResponse::Ok() 200 OK content_type body HttpResponse body get_index
```

Étant donné que le texte de réponse contient beaucoup de guillemets doubles, nous l'écrivons en utilisant la syntaxe Rust « chaîne brute »: la lettre , zéro ou plusieurs marques de hachage (c'est-à-dire le caractère), un guillemet double, puis le contenu de la chaîne, terminé par un autre guillemet double suivi du même nombre de marques de hachage. Tout caractère peut apparaître dans une chaîne brute sans être échappé, y compris les guillemets doubles ; en fait, aucune séquence d'échappement comme celle-ci n'est reconnue. Nous pouvons toujours nous assurer que la chaîne se termine là où nous le voulons en utilisant plus de marques de hachage autour des guillemets que jamais n'apparaissent dans le texte.

```
r # \"
```

Après avoir écrit *main.rs*, nous pouvons utiliser la commande pour faire tout ce qui est nécessaire pour le mettre en marche: récupérer les caisses nécessaires, les compiler, construire notre propre programme, tout relier ensemble et le démarrer: cargo run

```
$ cargo run
    Updating crates.io index
  Downloading crates ...
    Downloaded serde v1.0.100
    Downloaded actix-web v1.0.8
    Downloaded serde_derive v1.0.100
...
  Compiling serde_json v1.0.40
  Compiling actix-router v0.1.5
  Compiling actix-http v0.2.10
  Compiling awc v0.2.7
  Compiling actix-web v1.0.8
  Compiling gcd v0.1.0 (/home/jimb/rust/actix-gcd)
  Finished dev [unoptimized + debuginfo] target(s) in 1m 24s
    Running `~/home/jimb/rust/actix-gcd/target/debug/actix-gcd` 
Serving on http://localhost:3000...
```

À ce stade, nous pouvons visiter l'URL donnée dans notre navigateur et voir la page illustrée plus haut dans [la figure 2-1](#).

Malheureusement, cliquer sur Calculer GCD ne fait rien, si ce n'est naviguer dans notre navigateur vers une page vierge. Corrigeons cela ensuite, en ajoutant un autre itinéraire à notre pour gérer la demande à partir de notre formulaire. App POST

Il est enfin temps d'utiliser la caisse que nous avons répertoriée dans notre fichier *Cargo.toml*: elle fournit un outil pratique qui nous aidera à traiter les données du formulaire. Tout d'abord, nous devrons ajouter la directive suivante en haut de *src/main.rs* : `serde use`

```
use serde::Deserialize;
```

Les programmeurs Rust rassemblent généralement toutes leurs déclarations vers le haut du fichier, mais ce n'est pas strictement nécessaire : Rust permet aux déclarations de se produire dans n'importe quel ordre, tant qu'elles apparaissent au niveau d'imbrication approprié. `use`

Ensuite, définissons un type de structure Rust qui représente les valeurs que nous attendons de notre formulaire :

```
#[derive(Deserialize)]
struct GcdParameters {
    n: u64,
    m: u64,
}
```

Cela définit un nouveau type nommé qui a deux champs, et , dont chacun est un —le type d'argument attendu par notre fonction. `GcdParameters n m u64 gcd`

L'annotation au-dessus de la définition est un attribut, comme l'attribut que nous avons utilisé précédemment pour marquer les fonctions de test. Placer un attribut au-dessus d'une définition de type indique à la caisse d'examiner le type lorsque le programme est compilé et de générer automatiquement du code pour analyser une valeur de ce type à partir de données dans le format utilisé par les formulaires HTML pour les demandes. En fait, cet attribut est suffisant pour vous permettre d'analyser une valeur à partir de presque tous les types de données structurées : JSON, YAML, TOML ou l'un des nombreux autres formats textuels et binaires. La caisse fournit également un attribut qui génère du code pour faire l'inverse, en prenant des valeurs Rust et en les écrivant dans un format structuré.

```
struct #[test] #  
[derive(Deserialize)] serde POST GcdParameters serde Serialize
```

Avec cette définition en place, nous pouvons écrire notre fonction de gestionnaire assez facilement:

```
fn post_gcd(form: web::Form<GcdParameters>) -> HttpResponse {  
    if form.n == 0 || form.m == 0 {  
        return HttpResponse::BadRequest()  
            .content_type("text/html")  
            .body("Computing the GCD with zero is boring.");  
    }  
  
    let response =  
        format!("The greatest common divisor of the numbers {} and {} \  
               is <b>{}</b>\n",  
               form.n, form.m, gcd(form.n, form.m));  
  
    HttpResponse::Ok()  
        .content_type("text/html")  
        .body(response)  
}
```

Pour qu'une fonction serve de gestionnaire de requêtes Actix, ses arguments doivent tous avoir des types qu'Actix sait extraire d'une requête HTTP. Notre fonction prend un argument, , dont le type est . Actix sait comment extraire une valeur de n'importe quel type à partir d'une requête HTTP si, et seulement si, T peut être déserialisé à partir de données de formulaire HTML. Depuis que nous avons placé l'attribut sur notre définition de type, Actix peut le déserialiser à partir des données de formulaire, de sorte que les gestionnaires de demandes peuvent s'attendre à une valeur en tant que paramètre. Ces relations entre les types et les fonctions sont toutes élaborées au moment de la compilation; Si vous écrivez une fonction de gestionnaire avec un type d'argument qu'Actix ne sait pas gérer, le compilateur Rust vous informe immédiatement de votre erreur.

```
post_gcd form web::Form<GcdParameters> web::Form<T> P
```

```

OST #

[derive(Deserialize)] GcdParameters web::Form<GcdParameters
>

```

En regardant à l'intérieur, la fonction renvoie d'abord une erreur HTTP si l'un des paramètres est nul, car notre fonction paniquera s'ils le sont. Ensuite, il construit une réponse à la demande à l'aide de la macro. La macro est comme la macro, sauf qu'au lieu d'écrire le texte dans la sortie standard, elle le renvoie sous forme de chaîne. Une fois qu'il a obtenu le texte de la réponse, l'enveloppe dans une réponse HTTP, définit son type de contenu et le renvoie pour être remis à l'expéditeur.

```
post_gcd 400
BAD REQUEST gcd format! format! println! post_gcd 200 OK
```

Nous devons également nous inscrire en tant que gestionnaire du formulaire. Nous allons remplacer notre fonction par cette version

```
:post_gcd main
```

```

fn main() {
    let server = HttpServer::new(|| {
        App::new()
            .route("/", web::get().to(get_index))
            .route("/gcd", web::post().to(post_gcd))
    });

    println!("Serving on http://localhost:3000...");
    server
        .bind("127.0.0.1:3000")
        .expect("error binding server to address")
        .run()
        .expect("error running server");
}

```

Le seul changement ici est que nous avons ajouté un autre appel à , établissant comme gestionnaire pour le chemin

```
.route web::post().to(post_gcd) "/gcd"
```

La dernière pièce restante est la fonction que nous avons écrite précédemment, qui va dans le fichier *actix-gcd/src/main.rs*. Avec cela en place, vous pouvez interrompre tous les serveurs que vous avez peut-être laissés en cours d'exécution et reconstruire et redémarrer le programme:

```
$ cargo run
Compiling actix-gcd v0.1.0 (/home/jimb/rust/actix-gcd)
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/actix-gcd`
Serving on http://localhost:3000...
```

Cette fois, en visitant *http://localhost:3000*, en entrant des chiffres et en cliquant sur le bouton Calculer GCD, vous devriez réellement voir des résultats ([Figure 2-2](#)).

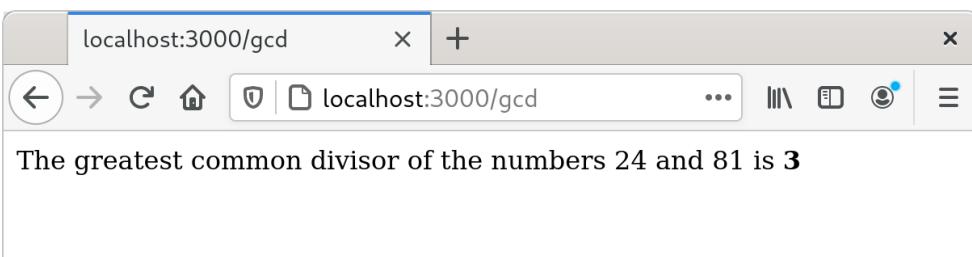


Figure 2-2. Page Web montrant les résultats du calcul gcd

Concurrence

L'une des grandes forces de Rust est sa prise en charge de la programmation simultanée. Les mêmes règles qui garantissent que les programmes Rust sont exempts d'erreurs de mémoire garantissent également que les threads ne peuvent partager la mémoire que de manière à éviter les courses de données. Par exemple:

- Si vous utilisez un mutex pour coordonner les threads qui apportent des modifications à une structure de données partagée, Rust s'assure que vous ne pouvez pas accéder aux données sauf lorsque vous maintenez le verrou enfoncé et libère automatiquement le verrou lorsque vous avez terminé. En C et C++, la relation entre un mutex et les données qu'il protège est laissée aux commentaires.
- Si vous souhaitez partager des données en lecture seule entre plusieurs threads, Rust garantit que vous ne pouvez pas modifier les données accidentellement. En C et C++, le système de type peut aider à cela, mais il est facile de se tromper.
- Si vous transférez la propriété d'une structure de données d'un thread à un autre, Rust s'assure que vous avez effectivement renoncé à tout accès à celle-ci. En C et C++, c'est à vous de vérifier que rien sur le thread d'envoi ne touchera plus jamais les données. Si vous ne le faites pas correctement, les effets peuvent dépendre de ce qui se trouve dans le cache du processeur et du nombre d'écritures en mémoire que vous avez effectuées récemment. Non pas que nous soyons amers.

Dans cette section, nous vous guiderons tout au long du processus d'écriture de votre deuxième programme multithread.

Vous avez déjà écrit votre premier : le framework Web Actix que vous avez utilisé pour implémenter le serveur Greatest Common Divisor utilise un pool de threads pour exécuter les fonctions de gestionnaire de requêtes. Si le serveur reçoit des requêtes simultanées, il peut exécuter les fonctions et dans plusieurs threads à la fois. Cela peut être un peu choquant, car nous n'avions certainement pas la concurrence à l'esprit lorsque nous avons écrit ces fonctions. Mais Rust garantit que cela est sûr à faire, peu importe l'élaboration de votre serveur: si votre programme compile, il est exempt de courses de données. Toutes les fonctions Rust sont thread-safe. `get_form post_gcd`

Le programme de cette section trace l'ensemble de Mandelbrot, une fractale produite en itérant une fonction simple sur des nombres complexes. Tracer l'ensemble de Mandelbrot est souvent appelé un algorithme *parallèle embarrassant*, parce que le modèle de communication entre les fils est si simple; nous couvrirons des modèles plus complexes dans [le chapitre 19](#), mais cette tâche démontre certains des éléments essentiels.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$ cargo new mandelbrot
Created binary (application) `mandelbrot` package
$ cd mandelbrot
```

Tout le code ira dans *mandelbrot / src / main.rs*, et nous ajouterons quelques dépendances à *mandelbrot / Cargo.toml*.

Avant d'entrer dans l'implémentation simultanée de Mandelbrot, nous devons décrire le calcul que nous allons effectuer.

Qu'est-ce que l'ensemble Mandelbrot est réellement

Lors de la lecture de code, il est utile d'avoir une idée concrète de ce qu'il essaie de faire, alors faisons une courte excursion dans les mathématiques pures. Nous allons commencer par un cas simple, puis ajouter des détails compliqués jusqu'à ce que nous arrivions au calcul au cœur de l'ensemble mandelbrot.

Voici une boucle infinie, écrite en utilisant la syntaxe dédiée de Rust pour cela, une déclaration: `loop`

```
fn square_loop(mut x: f64) {
    loop {
        x = x * x;
    }
}
```

Dans la vraie vie, Rust peut voir qu'il n'est jamais utilisé pour quoi que ce soit et pourrait donc ne pas se donner la peine de calculer sa valeur. Mais pour le moment, supposons que le code s'exécute comme écrit. Qu'adviennent-il de la valeur de x ? La quadrature de tout nombre inférieur à 1 le rend plus petit, de sorte qu'il se rapproche de zéro; la quadrature 1 donne 1; la quadrature d'un nombre supérieur à 1 le rend plus grand, de sorte qu'il se rapproche de l'infini; et la quadrature d'un nombre négatif le rend positif, après quoi il se comporte comme l'un des cas précédents ([Figure 2-3](#)). $x \times x$

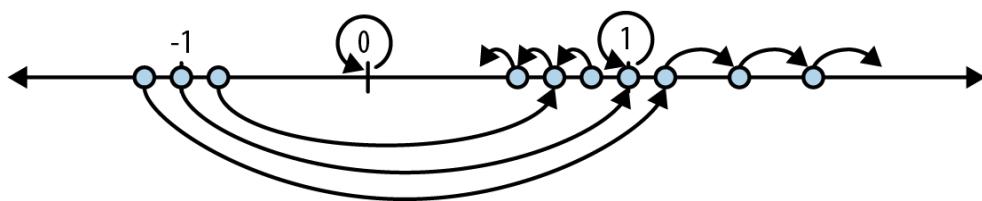


Figure 2-3. Effets de la quadrature répétée d'un certain nombre

Ainsi, selon la valeur à laquelle vous passez, reste à zéro ou à un, s'approche de zéro ou s'approche de l'infini. `square_loop x`

Considérons maintenant une boucle légèrement différente:

```
fn square_add_loop(c: f64) {
    let mut x = 0.0;
    loop {
        x = x * x + c;
    }
}
```

Cette fois, commence à zéro, et nous modifions sa progression dans chaque itération en l'ajoutant après l'avoir quadragéné. Cela rend plus difficile de voir comment les tarifs, mais certaines expérimentations montrent que si est supérieur à 0,25 ou inférieur à -2,0, alors finit par devenir infiniment grand; sinon, il reste quelque part dans le voisinage de zéro. `x c x c x`

La ride suivante : au lieu d'utiliser des valeurs, considérez la même boucle en utilisant des nombres complexes. La caisse sur crates.io fournit un type de nombre complexe que nous pouvons utiliser, nous devons donc ajouter une ligne pour à la section dans le fichier *Cargo.toml* de notre programme. Voici l'intégralité du fichier, jusqu'à présent (nous en ajouteron d'autres plus tard) : `f64 num num [dependencies]`

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

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

[dependencies]
num = "0.4"
```

Maintenant, nous pouvons écrire l'avant-dernière version de notre boucle:

```
use num::Complex;

fn complex_square_add_loop(c: Complex<f64>) {
    let mut z = Complex { re: 0.0, im: 0.0 };
```

```

    loop {
        z = z * z + c;
    }
}

```

Il est traditionnel à utiliser pour les nombres complexes, nous avons donc renommé notre variable en boucle. L'expression est la façon dont nous écrivons un zéro complexe en utilisant le type de caisse. est un type de structure Rust (ou *struct*), défini comme ceci : `z Complex { re: 0.0, im: 0.0 } num Complex Complex`

```

struct Complex<T> {
    /// Real portion of the complex number
    re: T,
    /// Imaginary portion of the complex number
    im: T,
}

```

Le code précédent définit une structure nommée , avec deux champs et . est une structure *générique* : vous pouvez lire le nom après le type comme « pour n'importe quel type ». Par exemple, est un nombre complexe dont les champs sont des valeurs, utiliserait des flottants 32 bits, etc. Compte tenu de cette définition, une expression like produit une valeur avec son champ initialisé à 0,24 et son champ initialisé à

```
0.3. Complex re im Complex <T> T Complex<f64> re im f64 Complex<f32> Complex { re: 0.24, im: 0.3 } Complex re im
```

La caisse s'arrange pour que , et d'autres opérateurs arithmétiques travaillent sur des valeurs, de sorte que le reste de la fonction fonctionne comme la version précédente, sauf qu'elle fonctionne sur des points du plan complexe, pas seulement sur des points le long de la ligne de nombre réel. Nous vous expliquerons comment vous pouvez faire en sorte que les opérateurs de Rust travaillent avec vos propres types au [chapitre 12.](#)

`num * + Complex`

Enfin, nous avons atteint la destination de notre excursion en mathématiques pures. L'ensemble de Mandelbrot est défini comme l'ensemble des nombres complexes pour lesquels il ne s'envole pas vers l'infini. Notre boucle de quadrature simple d'origine était assez prévisible : tout nombre supérieur à 1 ou inférieur à -1 s'envole. Jeter un dans chaque itération rend le comportement un peu plus difficile à anticiper: comme nous l'avons dit plus tôt, des valeurs supérieures à 0,25 ou inférieures à -2 provoquent un vol. Mais étendre le jeu à des nombres complexes produit des motifs vraiment bizarres et beaux, qui sont ce que nous voulons tracer. `c z + c c z`

Étant donné qu'un nombre complexe a à la fois des composantes réelles et imaginaires et , nous les traiterons comme les coordonnées d'un point sur le plan cartésien, et colorerons le point en noir si est dans l'ensemble de Mandelbrot, ou une couleur plus claire sinon. Ainsi, pour chaque pixel de notre image, nous devons exécuter la boucle précédente sur le point correspondant sur le plan complexe, voir s'il s'échappe à l'infini ou orbite autour de l'origine pour toujours, et le colorier en conséquence.

`c.c.re c.c.im x y c`

La boucle infinie prend un certain temps à courir, mais il y a deux astuces pour les impatients. Tout d'abord, si nous renonçons à exécuter la boucle pour toujours et essayons simplement un nombre limité d'itérations, il s'avère que nous obtenons toujours une approximation décente de l'ensemble. Le nombre d'itérations dont nous avons besoin dépend de la précision avec laquelle nous voulons tracer la frontière. Deuxième-ment, il a été démontré que, si jamais le cercle de rayon 2 est centré sur l'origine, il finira certainement par voler infinitement loin de l'origine.

Voici donc la version finale de notre boucle, et le cœur de notre programme :

```
use num::Complex;

/// Try to determine if `c` is in the Mandelbrot set, using at most `limit`
/// iterations to decide.
///
/// If `c` is not a member, return `Some(i)`, where `i` is the number of
/// iterations it took for `c` to leave the circle of radius 2 centered on the
/// origin. If `c` seems to be a member (more precisely, if we reached the
/// iteration limit without being able to prove that `c` is not a member),
/// return `None`.
fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let mut z = Complex { re: 0.0, im: 0.0 };
    for i in 0..limit {
        if z.norm_sqr() > 4.0 {
            return Some(i);
        }
        z = z * z + c;
    }

    None
}
```

Cette fonction prend le nombre complexe que nous voulons tester pour l'appartenance à l'ensemble de Mandelbrot et une limite sur le nombre d'itérations à essayer avant d'abandonner et de déclarer être probablement membre.

`c.c`

La valeur renvoyée de la fonction est un fichier . La bibliothèque standard de Rust définit le type comme suit :

`Option<usize>`

```
enum Option<T> {
    None,
    Some(T),
}
```

Option est un *type énuméré*, souvent appelé *enum*, car sa définition énumère plusieurs variantes qu'une valeur de ce type pourrait être : pour tout type , une valeur de type est soit , où est une valeur de type , soit , indiquant qu'aucune valeur n'est disponible. Comme le type dont nous avons parlé précédemment, est un type générique: vous pouvez utiliser pour représenter une valeur facultative de n'importe quel type que vous aimez. T Option<T> Some(v) v T None T Complex Option Option<T> > T

Dans notre cas, renvoie un pour indiquer si se trouve dans l'ensemble de Mandelbrot – et si ce n'est pas le cas, combien de temps nous avons dû itérer pour le savoir. Si n'est pas dans l'ensemble, renvoie , où est le numéro de l'itération à laquelle a quitté le cercle de rayon 2. Sinon, est apparemment dans l'ensemble, et renvoie

```
.escape_time Option<usize> c c escape_time Some(i) i z c escape_time None
```

```
for i in 0..limit {
```

Les exemples précédents montraient des boucles itérant sur des arguments de ligne de commande et des éléments vectoriels ; cette boucle itère simplement sur la plage d'entiers commençant par et jusqu'à (mais n'incluant pas). for for 0 limit

L'appel de méthode renvoie le carré de la distance de 'de l'origine. Pour décider si a quitté le cercle de rayon 2, au lieu de calculer une racine carree, nous comparons simplement la distance au carré avec 4,0, ce qui est plus rapide. z.norm_sqr() z z

Vous avez peut-être remarqué que nous utilisons pour marquer les lignes de commentaire au-dessus de la définition de la fonction; les commentaires ci-dessus les membres de la structure commencent également par. Il s'agit de *commentaires sur la documentation*; l'utilitaire sait comment les analyser, ainsi que le code qu'ils décrivent, et produire une documentation en ligne. La documentation de la bibliothèque standard de Rust est écrite sous cette forme. Nous décrivons en détail les commentaires de documentation au [chapitre 8.](#) /// Complex /// rustdoc

Le reste du programme consiste à décider quelle partie de l'ensemble tracer à quelle résolution et à répartir le travail sur plusieurs threads pour accélérer le calcul.

Analyse des arguments de ligne de commande de paire

Le programme prend plusieurs arguments de ligne de commande contrôlant la résolution de l'image que nous allons écrire et la partie de l'ensemble Mandelbrot que l'image montre. Étant donné que ces arguments de ligne de commande suivent tous une forme commune, voici une fonction pour les analyser :

```
use std::str::FromStr;

/// Parse the string `s` as a coordinate pair, like `"400x600"` or `"1.0,0.5"`
///
/// Specifically, `s` should have the form <left><sep><right>, where <sep> is
/// the character given by the `separator` argument, and <left> and <right> are
/// both strings that can be parsed by `T::from_str`. `separator` must be an
/// ASCII character.
///
/// If `s` has the proper form, return `Some<(x, y)>`. If it doesn't parse
/// correctly, return `None`.
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
    match s.find(separator) {
        None => None,
        Some(index) => {
            match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {
                (Ok(l), Ok(r)) => Some((l, r)),
                _ => None
            }
        }
    }
}

#[test]
fn test_parse_pair() {
    assert_eq!(parse_pair::<i32>("",      ','), None);
    assert_eq!(parse_pair::<i32>("10,",    ','), None);
    assert_eq!(parse_pair::<i32>(",10",   ','), None);
    assert_eq!(parse_pair::<i32>("10,20",  ','), Some((10, 20)));
    assert_eq!(parse_pair::<i32>("10,20xy", ','), None);
    assert_eq!(parse_pair::<f64>("0.5x",   'x'), None);
    assert_eq!(parse_pair::<f64>("0.5x1.5", 'x'), Some((0.5, 1.5)));
}
```

La définition de est une *fonction générique* : `parse_pair`

```
fn parse_pair<T: FromStr>(s: &str, separator: char) -> Option<(T, T)> {
```

Vous pouvez lire la clause à haute voix comme suit : « Pour tout type qui implémente le trait... Cela nous permet effectivement de définir toute une famille de fonctions à la fois : c'est une fonction qui analyse des paires de

valeurs, analyse des paires de valeurs en virgule flottante, etc. Cela ressemble beaucoup à un modèle de fonction en C++. Un programmeur Rust appellera un *paramètre de type* . Lorsque vous utilisez une fonction générique, Rust sera souvent en mesure de déduire des paramètres de type pour vous, et vous n'aurez pas besoin de les écrire comme nous l'avons fait dans le code de test. <T:

```
FromStr> T FromStr parse_pair:::<i32> i32 parse_pair:::  
<f64> T parse_pair
```

Notre type de retour est : soit ou une valeur , où est un tuple de deux valeurs, toutes deux de type . La fonction n'utilise pas d'instruction return explicite, de sorte que sa valeur return est la valeur de la dernière (et unique) expression de son corps : Option<(T, T)> None Some((v1, v2)) (v1, v2) T parse_pair

```
match s.find(separator) {  
    None => None,  
    Some(index) => {  
        ...  
    }  
}
```

La méthode type recherche dans la chaîne un caractère qui correspond à . Si renvoie , ce qui signifie que le caractère séparateur ne se produit pas dans la chaîne, l'expression entière est évaluée à , indiquant que l'analyse a échoué. Sinon, nous prenons pour être la position du séparateur dans la chaîne. String find separator find None match None index

```
match (T::from_str(&s[..index]), T::from_str(&s[index + 1..])) {  
    (Ok(l), Ok(r)) => Some((l, r)),  
    _ => None  
}
```

Cela commence à montrer la puissance de l'expression. L'argument de la correspondance est cette expression de tuple : match

```
(T::from_str(&s[..index]), T::from_str(&s[index + 1..]))
```

Les expressions et sont des tranches de la chaîne, précédent et suivant le séparateur. La fonction associée au paramètre type prend chacun d'entre eux et tente de les analyser comme une valeur de type , produisant un tuple de résultats. Voici ce à quoi nous nous opposons

```
: &s[..index] &s[index + 1..] T from_str T
```

```
(Ok(l), Ok(r)) => Some((l, r)),
```

Ce modèle ne correspond que si les deux éléments du tuple sont des variantes du type, ce qui indique que les deux analyses ont réussi. Si c'est le cas, est la valeur de l'expression de correspondance et donc la valeur de retour de la fonction. Ok Result Some((l, r))

_ => None

Le modèle générique correspond à n'importe quoi et ignore sa valeur. Si nous atteignons ce point, alors a échoué, nous évaluons donc à , four-nissant à nouveau la valeur de retour de la fonction. _ parse_pair None

Maintenant que nous avons , il est facile d'écrire une fonction pour analyser une paire de coordonnées à virgule flottante et les renvoyer sous forme de valeur : parse_pair Complex<f64>

```
// Parse a pair of floating-point numbers separated by a comma as a complex
// number.
fn parse_complex(s: &str) -> Option<Complex<f64>> {
    match parse_pair(s, ',') {
        Some((re, im)) => Some(Complex { re, im }),
        None => None
    }
}

#[test]
fn test_parse_complex() {
    assert_eq!(parse_complex("1.25,-0.0625"),
               Some(Complex { re: 1.25, im: -0.0625 }));
    assert_eq!(parse_complex(", -0.0625"), None);
}
```

La fonction appelle , crée une valeur si les coordonnées ont été analysées avec succès et transmet les échecs à son appelant. parse_complex parse_pair Complex

Si vous lisiez attentivement, vous avez peut-être remarqué que nous avons utilisé une notation abrégée pour construire la valeur. Il est courant d'initialiser les champs d'une struct avec des variables du même nom, donc plutôt que de vous forcer à écrire, Rust vous permet simplement d'écrire . Ceci est calqué sur des notations similaires en JavaScript et Haskell. Complex Complex { re: re, im: im } Complex { re, im }

Mappage des pixels aux nombres complexes

Le programme doit travailler dans deux espaces de coordonnées connexes: chaque pixel de l'image de sortie correspond à un point sur le plan complexe. La relation entre ces deux espaces dépend de la partie de l'ensemble de Mandelbrot que nous allons tracer et de la résolution de l'im-

age demandée, telle que déterminée par les arguments de ligne de commande. La fonction suivante convertit *l'espace d'image en espace de nombres complexes* :

```
// Given the row and column of a pixel in the output image, return the
// corresponding point on the complex plane.
//
// `bounds` is a pair giving the width and height of the image in pixels.
// `pixel` is a (column, row) pair indicating a particular pixel in that image.
// The `upper_left` and `lower_right` parameters are points on the complex
// plane designating the area our image covers.
fn pixel_to_point(bounds: (usize, usize),
                  pixel: (usize, usize),
                  upper_left: Complex<f64>,
                  lower_right: Complex<f64>)
    -> Complex<f64>
{
    let (width, height) = (lower_right.re - upper_left.re,
                           upper_left.im - lower_right.im);
    Complex {
        re: upper_left.re + pixel.0 as f64 * width / bounds.0 as f64,
        im: upper_left.im - pixel.1 as f64 * height / bounds.1 as f64
        // Why subtraction here? pixel.1 increases as we go down,
        // but the imaginary component increases as we go up.
    }
}

#[test]
fn test_pixel_to_point() {
    assert_eq!(pixel_to_point((100, 200), (25, 175),
                             Complex { re: -1.0, im: 1.0 },
                             Complex { re: 1.0, im: -1.0 }),
               Complex { re: -0.5, im: -0.75 });
}
```

[La figure 2-4](#) illustre le calcul effectué `pixel_to_point`

Le code de est simplement un calcul, nous ne l'expliquerons donc pas en détail. Cependant, il y a quelques points à souligner. Les expressions de cette forme font référence à des éléments de tuple : `pixel_to_point`

`pixel.0`

Il s'agit du premier élément du tuple `.pixel`

`pixel.0 as f64`

C'est la syntaxe de Rust pour une conversion de type : cela convertit en valeur. Contrairement à C et C++, Rust refuse généralement de convertir implicitement entre les types numériques; vous devez écrire les conver-

sions dont vous avez besoin. Cela peut être fastidieux, mais être explicite sur les conversions qui se produisent et quand est étonnamment utile. Les conversions implicites d'entiers semblent assez innocentes, mais historiquement, elles ont été une source fréquente de bogues et de failles de sécurité dans le code C et C ++ du monde réel. `pixel.0 f64`

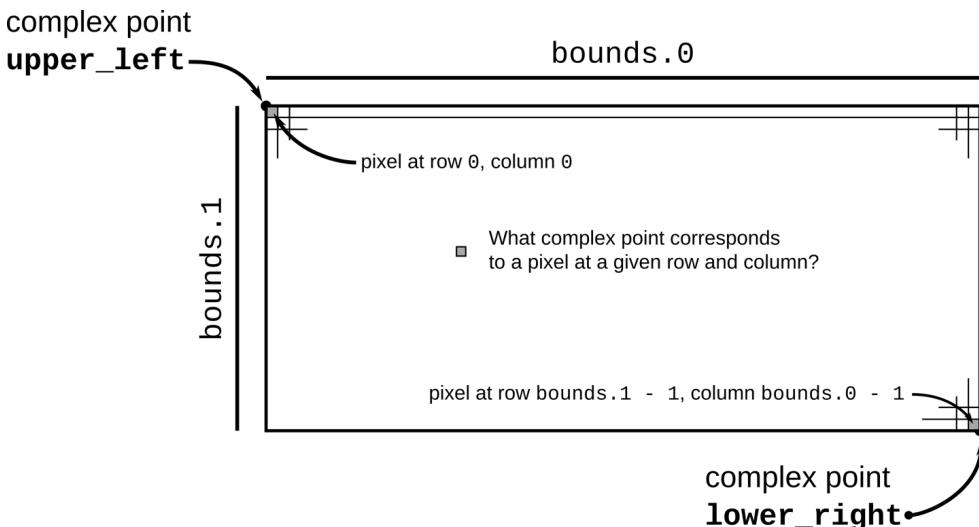


Figure 2-4. La relation entre le plan complexe et les pixels de l'image

Tracé de l'ensemble

Pour tracer l'ensemble de Mandelbrot, pour chaque pixel de l'image, il suffit d'appliquer au point correspondant sur le plan complexe, et de colorier le pixel en fonction du résultat: `escape_time`

```

/// Render a rectangle of the Mandelbrot set into a buffer of pixels.
///
/// The `bounds` argument gives the width and height of the buffer `pixels`,
/// which holds one grayscale pixel per byte. The `upper_left` and `lower_right`
/// arguments specify points on the complex plane corresponding to the upper-left
/// and lower-right corners of the pixel buffer.
fn render(pixels: &mut [u8],
          bounds: (usize, usize),
          upper_left: Complex<f64>,
          lower_right: Complex<f64>)
{
    assert!(pixels.len() == bounds.0 * bounds.1);

    for row in 0..bounds.1 {
        for column in 0..bounds.0 {
            let point = pixel_to_point(bounds, (column, row),
                                       upper_left, lower_right);
            pixels[row * bounds.0 + column] =
                match escape_time(point, 255) {
                    None => 0,
                    Some(count) => 255 - count as u8
                };
        }
    }
}

```

Tout cela devrait sembler assez familier à ce stade.

```
pixels[row * bounds.0 + column] =
    match escape_time(point, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };
}
```

Si dit que cela appartient à l'ensemble, colore le pixel correspondant en noir 0. Sinon, attribue des couleurs plus sombres aux nombres qui ont mis plus de temps à s'échapper du cercle.
escape_time point render 0 render

Écriture de fichiers image

La caisse fournit des fonctions de lecture et d'écriture d'une grande variété de formats d'image, ainsi que certaines fonctions de manipulation d'image de base. En particulier, il comprend un encodeur pour le format de fichier image PNG, que ce programme utilise pour enregistrer les résultats finaux du calcul. Pour utiliser , ajoutez la ligne suivante à la section de *Cargo.toml*:
image image [dependencies]

```
image = "0.13.0"
```

Avec cela en place, nous pouvons écrire:

```
use image::ColorType;
use image::png::PNGEncoder;
use std::fs::File;

/// Write the buffer `pixels`, whose dimensions are given by `bounds`, to the
/// file named `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (u32, u32))
    -> Result<(), std::io::Error>
{
    let output = File::create(filename)?;

    let encoder = PNGEncoder::new(output);
    encoder.encode(pixels,
                    bounds.0 as u32, bounds.1 as u32,
                    ColorType::Gray(8))?;

    Ok(())
}
```

Le fonctionnement de cette fonction est assez simple: il ouvre un fichier et essaie d'y écrire l'image. Nous passons à l'encodeur les données de pixels réelles de , ainsi que sa largeur et sa hauteur à partir de , puis un argument final qui dit comment interpréter les octets dans : la valeur indique

```
que chaque octet est une valeur en niveaux de gris de huit  
bits. pixels bounds pixels ColorType::Gray(8)
```

Tout cela est simple. Ce qui est intéressant à propos de cette fonction, c'est la façon dont elle fait face lorsque quelque chose ne va pas. Si nous rencontrons une erreur, nous devons la signaler à notre appelant. Comme nous l'avons mentionné précédemment, les fonctions faillibles dans Rust doivent renvoyer une valeur, qui est soit sur le succès, où est la valeur réussie, soit sur l'échec, où est un code d'erreur. Alors, quels sont les types de succès et d'erreurs de ?Result Ok(s) s Err(e) e write_image

Quand tout se passe bien, notre fonction n'a aucune valeur utile à retourner; il a écrit tout ce qui était intéressant pour le dossier. Donc, son type de succès est le type *d'unité*, ainsi appelé parce qu'il n'a qu'une seule valeur, également écrite . Le type d'unité est similaire à C et C

```
++.write_image () () void
```

Lorsqu'une erreur se produit, c'est parce que vous n'avez pas pu créer le fichier ou que vous n'avez pas pu y écrire l'image ; l'opération d'E/S a renvoyé un code d'erreur. Le type de retour de est , tandis que celui de est , donc les deux partagent le même type d'erreur, . Il est logique que notre fonction fasse de même. Dans les deux cas, l'échec devrait entraîner un retour immédiat, en transmettant la valeur décrivant ce qui s'est mal passé.
`File::create encoder.encode File::create Result<std::fs::File, std::io::Error> encoder.encode Result<(), std::io::Error> std::io::Error write_image std::io::Error`

Donc, pour gérer correctement le résultat de , nous devons sur sa valeur de retour, comme ceci:
`File::create match`

```
let output = match File::create(filename) {  
    Ok(f) => f,  
    Err(e) => {  
        return Err(e);  
    }  
};
```

Sur le succès, que soit le porté dans la valeur. En cas d'échec, transmettez l'erreur à notre propre appelant. `output File Ok`

Ce type d'énoncé est un modèle si courant dans Rust que le langage fournit l'opérateur comme raccourci pour l'ensemble. Ainsi, plutôt que d'écrire cette logique explicitement chaque fois que nous tentons quelque chose qui pourrait échouer, vous pouvez utiliser l'instruction équivalente et beaucoup plus lisible suivante: `match ?`

```
let output = File::create(filename)?;
```

En cas d'échec, l'opérateur renvoie à partir de , en transmettant l'erreur.

Sinon, maintient le fichier .File::create ?

```
write_image output File
```

NOTE

C'est une erreur courante du débutant d'essayer d'utiliser dans la fonction. Cependant, comme elle-même ne renvoie pas de valeur, cela ne fonctionnera pas; au lieu de cela, vous devez utiliser une instruction ou l'une des méthodes abrégées telles que et . Il y a aussi la possibilité de simplement changer pour retourner un , que nous couvrirons plus tard. ?

```
main main match unwrap expect main Result
```

Un programme Mandelbrot simultané

Toutes les pièces sont en place, et nous pouvons vous montrer la fonction, où nous pouvons mettre la concurrence au travail pour nous. Tout d'abord, une version non récurrente pour plus de simplicité : main

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    if args.len() != 5 {
        eprintln!("Usage: {} FILE PIXELS UPPERLEFT LOWERRIGHT",
                  args[0]);
        eprintln!("Example: {} mandel.png 1000x750 -1.20,0.35 -1,0.20",
                  args[0]);
        std::process::exit(1);
    }

    let bounds = parse_pair(&args[2], 'x')
        .expect("error parsing image dimensions");
    let upper_left = parse_complex(&args[3])
        .expect("error parsing upper left corner point");
    let lower_right = parse_complex(&args[4])
        .expect("error parsing lower right corner point");

    let mut pixels = vec![0; bounds.0 * bounds.1];

    render(&mut pixels, bounds, upper_left, lower_right);

    write_image(&args[1], &pixels, bounds)
        .expect("error writing PNG file");
}
```

Après avoir rassemblé les arguments de ligne de commande dans un vecteur de s, nous analysons chacun d'eux, puis commençons les calculs. String

```
let mut pixels = vec![0; bounds.0 * bounds.1];
```

Un appel de macro crée un élément vectoriel long dont les éléments sont initialisés à , de sorte que le code précédent crée un vecteur de zéros dont la longueur est , où est la résolution de l'image analysée à partir de la ligne de commande. Nous utiliserons ce vecteur comme un tableau rectangulaire de valeurs de pixels en niveaux de gris d'un octet, comme illustré à [la figure 2-5](#). `vec![v; n] n v bounds.0 * bounds.1 bounds`

La ligne d'intérêt suivante est la suivante:

```
render(&mut pixels, bounds, upper_left, lower_right);
```

Cela appelle la fonction pour calculer réellement l'image. L'expression emprunte une référence mutable à notre tampon de pixels, permettant de le remplir avec des valeurs d'échelles de gris calculées, même si reste le propriétaire du vecteur. Les arguments restants passent les dimensions de l'image et le rectangle du plan complexe que nous avons choisi de tracer. `render &mut pixels render pixels`

```
write_image(&args[1], &pixels, bounds)
    .expect("error writing PNG file");
```

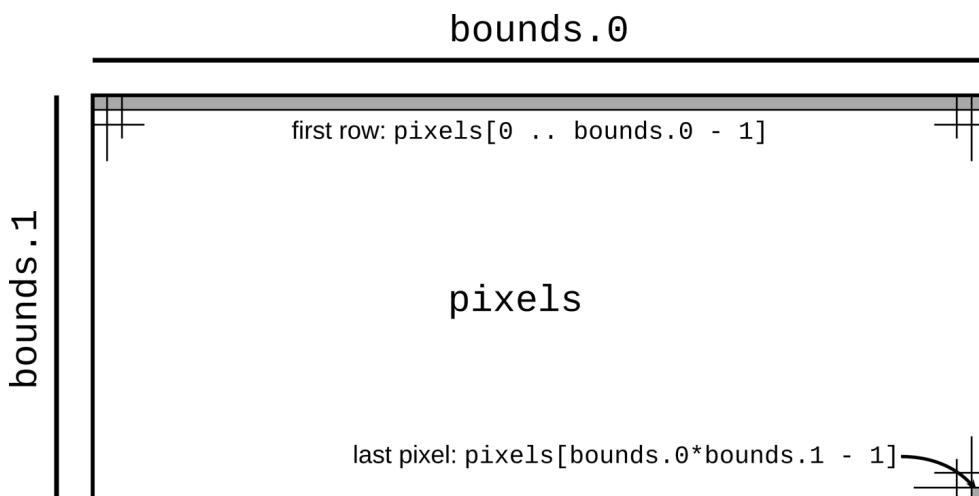


Figure 2-5. Utilisation d'un vecteur comme tableau rectangulaire de pixels

Enfin, nous écrivons le tampon de pixels sur le disque sous forme de fichier PNG. Dans ce cas, nous passons une référence partagée (non modifiable) au tampon, car il ne devrait pas être nécessaire de modifier le contenu du tampon. `write_image`

À ce stade, nous pouvons construire et exécuter le programme en mode release, ce qui permet de nombreuses optimisations puissantes du compilateur, et après plusieurs secondes, il écrira une belle image dans le fichier *mandel.png*:

```
$ cargo build --release
Updating crates.io index
```

```

Compiling autocfg v1.0.1
...
Compiling image v0.13.0
Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
    Finished release [optimized] target(s) in 25.36s
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m4.678s
user    0m4.661s
sys     0m0.008s

```

Cette commande doit créer un fichier appelé *mandel.png*, que vous pouvez afficher avec le programme de visualisation d'images de votre système ou dans un navigateur Web. Si tout s'est bien passé, cela devrait ressembler [à la figure 2-6](#).

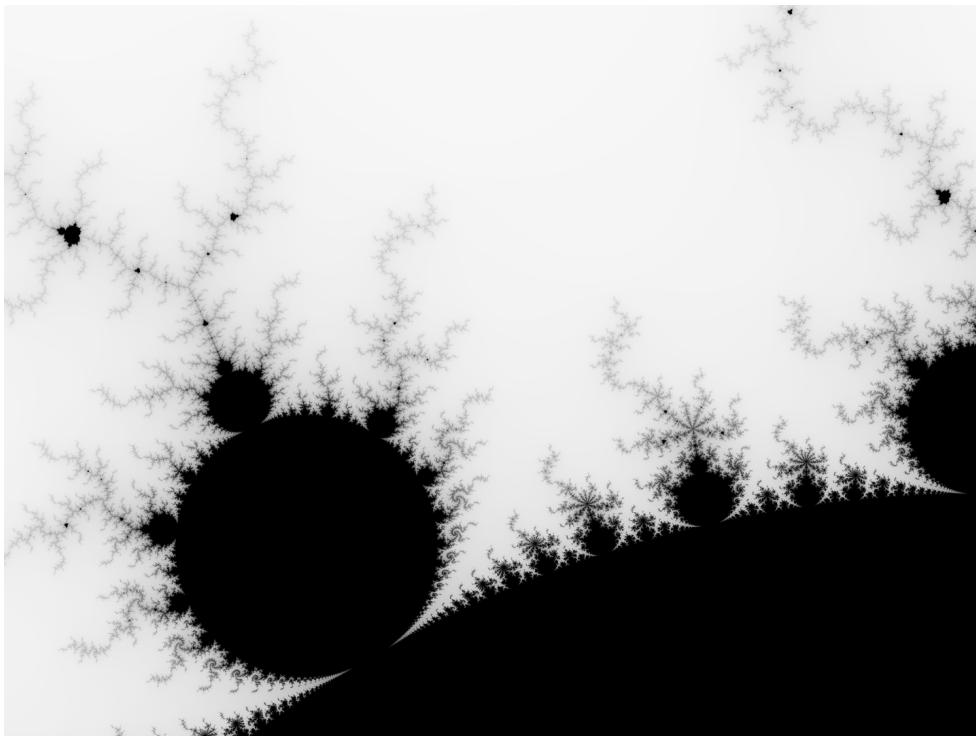


Figure 2-6. Résultats du programme parallèle Mandelbrot

Dans la transcription précédente, nous avons utilisé le programme Unix pour analyser la durée d'exécution du programme: il a fallu environ cinq secondes au total pour exécuter le calcul de Mandelbrot sur chaque pixel de l'image. Mais presque toutes les machines modernes ont plusieurs coeurs de processeur, et ce programme n'en utilisait qu'un. Si nous pouvions répartir le travail sur toutes les ressources informatiques que la machine a à offrir, nous devrions être en mesure de compléter l'image beaucoup plus rapidement. `time`

À cette fin, nous allons diviser l'image en sections, une par processeur, et laisser chaque processeur colorer les pixels qui lui sont attribués. Pour plus de simplicité, nous allons le diviser en bandes horizontales, comme illustré à [la figure 2-7](#). Lorsque tous les processeurs ont terminé, nous pouvons écrire les pixels sur le disque.

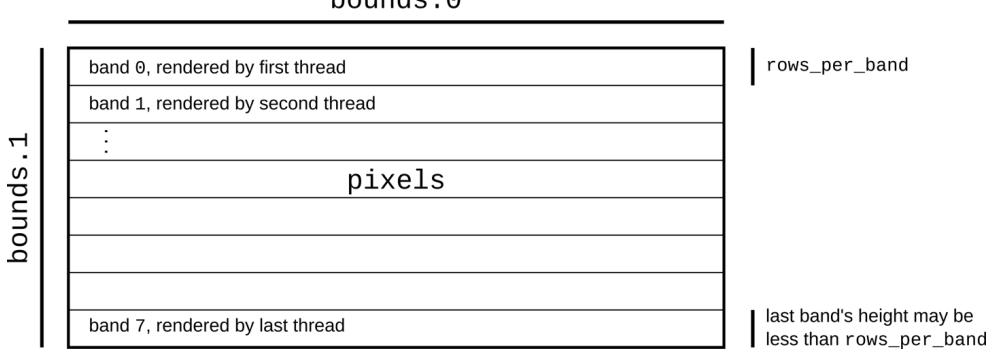


Figure 2-7. Diviser le tampon de pixels en bandes pour un rendu parallèle

La caisse fournit un certain nombre d'installations de concurrence précieuses, y compris une fonction *de thread étendue* qui fait exactement ce dont nous avons besoin ici. Pour l'utiliser, nous devons ajouter la ligne suivante à notre fichier *Cargo.toml*: `crossbeam`

```
crossbeam = "0.8"
```

Ensuite, nous devons supprimer l'appel de ligne unique et le remplacer par ce qui suit: `render`

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;

{
    let bands: Vec<&mut [u8]> =
        pixels.chunks_mut(rows_per_band * bounds.0).collect();
    crossbeam::scope(|spawner| {
        for (i, band) in bands.into_iter().enumerate() {
            let top = rows_per_band * i;
            let height = band.len() / bounds.0;
            let band_bounds = (bounds.0, height);
            let band_upper_left =
                pixel_to_point(bounds, (0, top), upper_left, lower_right);
            let band_lower_right =
                pixel_to_point(bounds, (bounds.0, top + height),
                               upper_left, lower_right);

            spawner.spawn(move |_| {
                render(band, band_bounds, band_upper_left, band_lower_right)
            });
        }
    }).unwrap();
}
```

Décomposer cela de la manière habituelle:

```
let threads = 8;
let rows_per_band = bounds.1 / threads + 1;
```

Ici, nous décidons d'utiliser huit fils.¹ Ensuite, nous calculons le nombre de lignes de pixels que chaque bande devrait avoir. Nous arrondissons le nombre de rangées vers le haut pour nous assurer que les bandes couvrent toute l'image, même si la hauteur n'est pas un multiple de .threads

```
let bands: Vec<&mut [u8]> =  
    pixels.chunks_mut(rows_per_band * bounds.0).collect();
```

Ici, nous divisons le tampon de pixels en bandes. La méthode du tampon renvoie un itérateur produisant des tranches mutables et non superposées du tampon, chacune contenant des pixels, c'est-à-dire des lignes complètes de pixels. La dernière tranche qui produit peut contenir moins de lignes, mais chaque ligne contiendra le même nombre de pixels. Enfin, la méthode de l'itérateur construit un vecteur contenant ces tranches mutables et non superposées. chunks_mut rows_per_band * bounds.0 rows_per_band chunks_mut collect

Maintenant, nous pouvons mettre la bibliothèque au travail: crossbeam

```
crossbeam::scope(|spawner| {  
    ...  
}).unwrap();
```

L'argument est une fermeture de Rust qui attend un seul argument, . Notez que, contrairement aux fonctions déclarées avec , nous n'avons pas besoin de déclarer les types d'arguments d'une fermeture ; La rouille les déduira, ainsi que son type de retour. Dans ce cas, appelle la fermeture, en passant comme argument une valeur que la fermeture peut utiliser pour créer de nouveaux threads. La fonction attend que tous ces threads terminent l'exécution avant de se retourner. Ce comportement permet à Rust d'être sûr que ces threads n'accéderont pas à leurs parties après qu'il soit sorti de la portée, et nous permet d'être sûr que lorsque les retours, le calcul de l'image est terminé. Si tout se passe bien, retourne , mais si l'un des fils que nous avons engendrés a paniqué, il renvoie un fichier . Nous faisons appel à cela pour que, dans ce cas, nous paniquions aussi, et l'utilisateur recevra un rapport. |spawner| { ... } spawner fn crossbeam::scope spawner crossbeam::scope pixels crossbeam::scope crossbeam::scope Ok(()) Err unwrap Result

```
for (i, band) in bands.into_iter().enumerate() {
```

Ici, nous itérons sur les bandes du tampon de pixels. L'itérateur donne à chaque itération du corps de boucle la propriété exclusive d'une bande, garantissant qu'un seul thread peut y écrire à la fois. Nous expliquons comment cela fonctionne en détail au [chapitre 5](#). Ensuite, l'adaptateur

produit des tuples appairant chaque élément vectoriel avec son index. `into_iter()` `enumerate`

```
let top = rows_per_band * i;
let height = band.len() / bounds.0;
let band_bounds = (bounds.0, height);
let band_upper_left =
    pixel_to_point(bounds, (0, top), upper_left, lower_right);
let band_lower_right =
    pixel_to_point(bounds, (bounds.0, top + height),
                    upper_left, lower_right);
```

Compte tenu de l'index et de la taille réelle de la bande (rappelons que la dernière peut être plus courte que les autres), nous pouvons produire un cadre de sélection du type requis, mais qui se réfère uniquement à cette bande du tampon, pas à l'image entière. De même, nous réutilisons la fonction du moteur de rendu pour trouver où les coins supérieur gauche et inférieur droit de la bande tombent sur le plan complexe. `render pixel_to_point`

```
spawner.spawn(move |_|
    render(band, band_bounds, band_upper_left, band_lower_right));
});
```

Enfin, nous créons un thread, exécutant la fermeture . Le mot-clé à l'avant indique que cette fermeture s'approprie les variables qu'elle utilise ; en particulier, seule la fermeture peut utiliser la tranche mutable . La liste d'arguments signifie que la fermeture prend un argument, qu'elle n'utilise pas (un autre spawner pour créer des threads imbriqués). `move |_| { ... } move band |_|`

Comme nous l'avons mentionné précédemment, l'appel garantit que tous les threads sont terminés avant son retour, ce qui signifie qu'il est sûr d'enregistrer l'image dans un fichier, ce qui est notre prochaine action. `crossbeam::scope`

Exécution du traceur Mandelbrot

Nous avons utilisé plusieurs caisses externes dans ce programme : pour l'arithmétique des nombres complexes, pour l'écriture de fichiers PNG et pour les primitives de création de threads étendus. Voici le fichier `Cargo.toml` final incluant toutes ces dépendances

```
: num image crossbeam
```

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

```
[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

Avec cela en place, nous pouvons construire et exécuter le programme:

```
$ cargo build --release
    Updating crates.io index
    Compiling crossbeam-queue v0.3.2
    Compiling crossbeam v0.8.1
    Compiling mandelbrot v0.1.0 ($RUSTBOOK/mandelbrot)
        Finished release [optimized] target(s) in #.## secs
$ time target/release/mandelbrot mandel.png 4000x3000 -1.20,0.35 -1,0.20
real    0m1.436s
user    0m4.922s
sys     0m0.011s
```

Ici, nous avons utilisé à nouveau pour voir combien de temps le programme a pris pour s'exécuter; Notez que même si nous avons quand même passé près de cinq secondes de temps processeur, le temps réel écoulé n'était que d'environ 1,5 seconde. Vous pouvez vérifier qu'une partie de ce temps est consacrée à l'écriture du fichier image en commentant le code qui le fait et en mesurant à nouveau. Sur l'ordinateur portable où ce code a été testé, la version concurrente réduit le temps de calcul de Mandelbrot proprement dit d'un facteur de près de quatre. Nous montrerons comment améliorer considérablement cela au [chapitre 19](#).

Comme précédemment, ce programme aura créé un fichier appelé *mandel.png*. Avec cette version plus rapide, vous pouvez explorer plus facilement l'ensemble mandelbrot en modifiant les arguments de ligne de commande à votre guise.

La sécurité est invisible

En fin de compte, le programme parallèle avec lequel nous nous sommes retrouvés n'est pas substantiellement différent de ce que nous pourrions écrire dans n'importe quel autre langage: nous répartissons des morceaux du tampon de pixels entre les processeurs, laissons chacun travailler sur son morceau séparément, et quand ils ont tous terminé, présentons le résultat. Alors, qu'y a-t-il de si spécial dans la prise en charge de la concurrence de Rust ?

Ce que nous n'avons pas montré ici, ce sont tous les programmes Rust que nous *ne pouvons pas* écrire. Le code que nous avons examiné dans ce chapitre partitionne correctement le tampon entre les threads, mais il existe de nombreuses petites variations de ce code qui ne le font pas (et introduisent donc des courses de données); aucune de ces variantes ne

passera les contrôles statiques du compilateur Rust. Un compilateur C ou C ++ vous aidera joyeusement à explorer le vaste espace des programmes avec des courses de données subtiles; Rust vous dit, dès le départ, quand quelque chose pourrait mal tourner.

Dans les chapitres [4](#) et [5](#), nous décrirons les règles de Rust en matière de sécurité de la mémoire. [Le chapitre 19](#) explique comment ces règles garantissent également une bonne hygiène de concurrence.

Systèmes de fichiers et outils de ligne de commande

Rust a trouvé un créneau important dans le monde des outils de ligne de commande. En tant que langage de programmation système moderne, sûr et rapide, il offre aux programmeurs une boîte à outils qu'ils peuvent utiliser pour assembler des interfaces de ligne de commande astucieuses qui répliquent ou étendent les fonctionnalités des outils existants. Par exemple, la commande fournit une alternative prenant en charge la coloration syntaxique avec prise en charge intégrée des outils de pagination et peut automatiquement comparer tout ce qui peut être exécuté avec une commande ou un pipeline. `bat cat hyperfine`

Bien que quelque chose d'aussi complexe soit hors de portée pour ce livre, Rust vous permet de plonger facilement vos orteils dans le monde des applications ergonomiques en ligne de commande. Dans cette section, nous allons vous montrer comment créer votre propre outil de recherche et de remplacement, avec une sortie colorée et des messages d'erreur conviviaux.

Pour commencer, nous allons créer un nouveau projet Rust :

```
$ cargo new quickreplace
     Created binary (application) `quickreplace` package
$ cd quickreplace
```

Pour notre programme, nous aurons besoin de deux autres caisses: pour créer une sortie colorée dans le terminal et pour la fonctionnalité de recherche et de remplacement réelle. Comme précédemment, nous avons mis ces caisses dans *Cargo.toml* pour dire que nous en avons besoin: `text-colorizer regex cargo`

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

# See more keys and their definitions at
```

```
[dependencies]
text-colorizer = "1"
regex = "1"
```

Les caisses de rouille qui ont atteint la version , comme celles-ci l'ont fait, suivent les règles de « versioning sémantique »: jusqu'à ce que le numéro de version majeur change, les nouvelles versions doivent toujours être des extensions compatibles de leurs prédecesseurs. Donc, si nous testons notre programme par rapport à la version d'une caisse, il devrait toujours fonctionner avec les versions , , et ainsi de suite; mais la version pourrait introduire des changements incompatibles. Lorsque nous demandons simplement la version d'une caisse dans un fichier *Cargo.toml*, Cargo utilisera la dernière version disponible de la caisse avant

```
.1.0 1 1.2 1.3 1.4 2.0 "1" 2.0
```

L'interface de ligne de commande

L'interface de ce programme est assez simple. Il faut quatre arguments : une chaîne (ou expression régulière) à rechercher, une chaîne (ou expression régulière) pour la remplacer, le nom d'un fichier d'entrée et le nom d'un fichier de sortie. Nous allons commencer notre fichier *main.rs* avec une structure contenant ces arguments :

```
#[derive(Debug)]
struct Arguments {
    target: String,
    replacement: String,
    filename: String,
    output: String,
}
```

L'attribut indique au compilateur de générer du code supplémentaire qui nous permet de formater la structure avec dans .#
[derive(Debug)] Arguments {?:} println!

Dans le cas où l'utilisateur entre le mauvais nombre d'arguments, il est d'usage d'imprimer une explication concise de la façon d'utiliser le programme. Nous allons le faire avec une fonction simple appelée et tout importer à partir de afin que nous puissions ajouter de la couleur: print_usage text-colorizer

```
use text_colorizer::*;

fn print_usage() {
    eprintln!("{} - change occurrences of one string into another",
             "quickreplace".green());
```

```
eprintln!("Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>");
}
```

Le simple fait d'ajouter à la fin d'un littéral de chaîne produit une chaîne enveloppée dans les codes d'échappement ANSI appropriés pour s'afficher en vert dans un émulateur de terminal. Cette chaîne est ensuite interpolée dans le reste du message avant d'être imprimée. `.green()`

Maintenant, nous pouvons collecter et traiter les arguments du programme:

```
use std::env;

fn parse_args() -> Arguments {
    let args: Vec<String> = env::args().skip(1).collect();

    if args.len() != 4 {
        print_usage();
        eprintln!("{} wrong number of arguments: expected 4, got {}.", "Error:".red().bold(), args.len());
        std::process::exit(1);
    }

    Arguments {
        target: args[0].clone(),
        replacement: args[1].clone(),
        filename: args[2].clone(),
        output: args[3].clone()
    }
}
```

Afin d'obtenir les arguments saisis par l'utilisateur, nous utilisons le même itérateur que dans les exemples précédents. Ignorons la première valeur de l'itérateur (le nom du programme en cours d'exécution) afin que le résultat ne comporte que les arguments de ligne de commande. `args .skip(1)`

La méthode produit un nombre d'arguments. Nous vérifions ensuite que le bon numéro est présent et, sinon, imprimons un message et repartons avec un code d'erreur. Nous colorisons à nouveau une partie du message et l'utilisons pour rendre le texte plus lourd. Si le bon nombre d'arguments est présent, nous les mettons dans une structure et le retournons. `collect() Vec .bold() Arguments`

Ensuite, nous ajouterons une fonction qui appelle et imprime simplement la sortie: `main parse_args`

```
fn main() {
    let args = parse_args();
```

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

À ce stade, nous pouvons exécuter le programme et voir qu'il crache le bon message d'erreur:

```
$ cargo run
Updating crates.io index
Compiling libc v0.2.82
Compiling lazy_static v1.4.0
Compiling memchr v2.3.4
Compiling regex-syntax v0.6.22
Compiling thread_local v1.1.0
Compiling aho-corasick v0.7.15
Compiling atty v0.2.14
Compiling text-colorizer v1.0.0
Compiling regex v1.4.3
Compiling quickreplace v0.1.0 (/home/jimb/quickreplace)
Finished dev [unoptimized + debuginfo] target(s) in 6.98s
Running `target/debug/quickreplace`
quickreplace - change occurrences of one string into another
Usage: quickreplace <target> <replacement> <INPUT> <OUTPUT>
Error: wrong number of arguments: expected 4, got 0
```

Si vous donnez au programme quelques arguments, il imprimera plutôt une représentation de la structure : Arguments

```
$ cargo run "find" "replace" file output
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
Running `target/debug/quickreplace find replace file output`
Arguments { target: "find", replacement: "replace", filename: "file", output
```

C'est un très bon début! Les arguments sont correctement repris et placés dans les parties correctes de la structure. Arguments

Lecture et écriture de fichiers

Ensuite, nous avons besoin d'un moyen d'obtenir des données du système de fichiers afin de pouvoir les traiter et les réécrire lorsque nous avons terminé. Rust dispose d'un ensemble robuste d'outils pour l'entrée et la sortie, mais les concepteurs de la bibliothèque standard savent que la lecture et l'écriture de fichiers sont très courantes, et ils l'ont fait à dessein. Tout ce que nous avons à faire est d'importer un module, , et nous avons accès aux fonctions et: std::fs read_to_string write

```
use std::fs;
```

std::fs::read_to_string renvoie un fichier . Si la fonction réussit, elle produit un fichier . En cas d'échec, il produit un , le type de biblio-

```

thème standard pour représenter les problèmes d'E/S. De même, renvoie
un : rien dans le cas de réussite, ou les mêmes détails d'erreur si quelque
chose ne va pas. Result<String,
std::io::Error> String std::io::Error std::fs::write Result
<(), std::io::Error>

fn main() {
    let args = parse_args();

    let data = match fs::read_to_string(&args.filename) {
        Ok(v) => v,
        Err(e) => {
            eprintln!("{} failed to read from file '{}': {:?}", 
                     "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };

    match fs::write(&args.output, &data) {
        Ok(_) => {},
        Err(e) => {
            eprintln!("{} failed to write to file '{}': {:?}", 
                     "Error:".red().bold(), args.filename, e);
            std::process::exit(1);
        }
    };
}

```

Ici, nous utilisons la fonction que nous avons écrite au préalable et passons les noms de fichiers résultats à et . Les instructions sur les sorties de ces fonctions gèrent les erreurs avec élégance, en imprimant le nom du fichier, la raison fournie de l'erreur et une petite touche de couleur pour attirer l'attention de

l'utilisateur. `parse_args()` `read_to_string` `write` `match`

Avec cette fonction mise à jour, nous pouvons exécuter le programme et voir que, bien sûr, le contenu des nouveaux et des anciens fichiers est exactement le même: `main`

```

$ cargo run "find" "replace" Cargo.toml Copy.toml
Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
Finished dev [unoptimized + debuginfo] target(s) in 0.01s
Running `target/debug/quickreplace find replace Cargo.toml Copy.toml`
```

Le programme lit dans le fichier d'entrée `Cargo.toml`, et il écrit dans le fichier de sortie `Copy.toml`, mais comme nous n'avons écrit aucun code pour réellement trouver et remplacer, rien dans la sortie n'a changé. Nous pouvons facilement vérifier en exécutant la commande, qui ne détecte aucune différence: `diff`

```
$ diff Cargo.toml Copy.toml
```

Rechercher et remplacer

La touche finale pour ce programme est de mettre en œuvre sa fonctionnalité réelle: trouver et remplacer. Pour cela, nous utiliserons la caisse, qui compile et exécute des expressions régulières. Il fournit une structure appelée , qui représente une expression régulière compilée. a une méthode , qui fait exactement ce qu'elle dit: elle recherche dans une chaîne toutes les correspondances de l'expression régulière et remplace chacune par une chaîne de remplacement donnée. Nous pouvons extraire cette logique dans une fonction : `regex Regex Regex replace_all`

```
use regex::Regex;
fn replace(target: &str, replacement: &str, text: &str)
    -> Result<String, regex::Error>
{
    let regex = Regex::new(target)?;
    Ok(regex.replace_all(text, replacement).to_string())
}
```

Notez le type de retour de cette fonction. Tout comme les fonctions de bibliothèque standard que nous avons utilisées précédemment, renvoie un , cette fois avec un type d'erreur fourni par la caisse. `replace` `Result` `regex`

`Regex::new` compile le regex fourni par l'utilisateur et peut échouer si une chaîne n'est pas valide. Comme dans le programme Mandelbrot, nous avons l'habitude de court-circuiter en cas d'échec, mais dans ce cas, la fonction renvoie un type d'erreur spécifique à la caisse. Une fois le regex compilé, sa méthode remplace toutes les correspondances par la chaîne de remplacement donnée. ? `Regex::new regex replace_all text`

Si trouve des correspondances, il renvoie un nouveau avec ces correspondances remplacées par le texte que nous lui avons donné. Sinon, renvoie un pointeur vers le texte d'origine, évitant ainsi l'allocation et la copie inutiles de mémoire. Dans ce cas, cependant, nous voulons toujours une copie indépendante, nous utilisons donc la méthode pour obtenir un dans les deux cas et renvoyer cette chaîne enveloppée dans , comme dans les autres

```
fonctions. replace_all String replace_all to_string String Res
ult::Ok
```

Maintenant, il est temps d'incorporer la nouvelle fonction dans notre code: `main`

```
fn main() {
    let args = parse_args();
```

```

let data = match fs::read_to_string(&args.filename) {
    Ok(v) => v,
    Err(e) => {
        eprintln!("{} failed to read from file '{}': {:?}", 
            "Error:".red().bold(), args.filename, e);
        std::process::exit(1);
    }
};

let replaced_data = match replace(&args.target, &args.replacement, &data) {
    Ok(v) => v,
    Err(e) => {
        eprintln!("{} failed to replace text: {:?}", 
            "Error:".red().bold(), e);
        std::process::exit(1);
    }
};

match fs::write(&args.output, &replaced_data) {
    Ok(v) => v,
    Err(e) => {
        eprintln!("{} failed to write to file '{}': {:?}", 
            "Error:".red().bold(), args.filename, e);
        std::process::exit(1);
    }
};
}

```

Avec cette touche finale, le programme est prêt et vous devriez pouvoir le tester:

```

$ echo "Hello, world" > test.txt
$ cargo run "world" "Rust" test.txt test-modified.txt
Compiling quickreplace v0.1.0 (/home/jimb/rust/quickreplace)
  Finished dev [unoptimized + debuginfo] target(s) in 0.88s
  Running `target/debug/quickreplace world test.txt test-modified.txt`

$ cat test-modified.txt
Hello, Rust

```

Et, bien sûr, la gestion des erreurs est également en place, signalant gracieusement les erreurs à l'utilisateur:

```

$ cargo run "[[a-z]]" "0" test.txt test-modified.txt
  Finished dev [unoptimized + debuginfo] target(s) in 0.01s
  Running `target/debug/quickreplace '[[a-z]]' 0 test.txt test-modified.txt`
Error: failed to replace text: Syntax(
-----
regex parse error:
  [[a-z]
^

```

```
error: unclosed character class
~~~~~
)
```

Il y a, bien sûr, de nombreuses fonctionnalités manquantes dans cette simple démonstration, mais les fondamentaux sont là. Vous avez vu comment lire et écrire des fichiers, propager et afficher des erreurs et coloriser la sortie pour améliorer l'expérience utilisateur dans le terminal.

Les prochains chapitres exploreront des techniques plus avancées pour le développement d'applications, des collections de données et de la programmation fonctionnelle avec des itérateurs aux techniques de programmation asynchrone pour une concurrence extrêmement efficace, mais d'abord, vous aurez besoin de la base solide du chapitre suivant dans les types de données fondamentaux de Rust.

- 1** La caisse fournit une fonction qui renvoie le nombre de CPU disponibles sur le système actuel. `num_cpus`

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 3. Types fondamentaux

Il y a beaucoup, beaucoup de types de livres dans le monde, ce qui est logique, parce qu'il y a beaucoup, beaucoup de types de gens, et tout le monde veut lire quelque chose de différent.

—Lemony Snicket

Dans une large mesure, le langage Rust est conçu autour de ses types. Sa prise en charge du code haute performance découle du fait qu'il permet aux développeurs de choisir la représentation des données qui convient le mieux à la situation, avec le bon équilibre entre simplicité et coût. Les garanties de sécurité de la mémoire et du filetage de Rust reposent également sur la solidité de son système de type, et la flexibilité de Rust découle de ses types et caractéristiques génériques.

Ce chapitre couvre les types fondamentaux de Rust pour représenter les valeurs. Ces types au niveau de la source ont des homologues concrets au niveau de la machine avec des coûts et des performances prévisibles. Bien que Rust ne promet pas qu'il représentera les choses exactement comme vous l'avez demandé, il prend soin de s'écartez de vos demandes uniquement lorsqu'il s'agit d'une amélioration fiable.

Comparé à un langage typé dynamiquement comme JavaScript ou Python, Rust nécessite plus de planification de votre part à l'avance. Vous devez épeler les types d'arguments de fonction et de valeurs de retour, les champs destruct et quelques autres constructions. Cependant, deux caractéristiques de Rust rendent cela moins difficile que vous ne le pensez:

- Compte tenu des types que vous épelez, *l'inférence de type* de Rust déterminera la plupart du reste pour vous. En pratique, il n'y a souvent qu'un seul type qui fonctionnera pour une variable ou une expression donnée; lorsque c'est le cas, Rust vous permet de laisser de côté, ou *d'éliminer*, le type. Par exemple, vous pouvez épeler chaque type d'une fonction, comme ceci :

```
fn build_vector() -> Vec<i16> {
    let mut v: Vec<i16> = Vec::new();
    v.push(10i16);
    v.push(20i16);
    v
}
```

Mais c'est encombré et répétitif. Étant donné le type de retour de la fonction, il est évident que doit être un , un vecteur d'entiers signés 16 bits ; aucun autre type ne fonctionnerait. Et il s'ensuit que chaque élément du vecteur doit être un . C'est exactement le genre de raisonnement que l'inférence de type rust applique, vous permettant d'écrire à la place: v Vec<i16> i16

```
fn build_vector() -> Vec<i16> {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v
}
```

Ces deux définitions sont exactement équivalentes, et Rust générera le même code machine dans les deux sens. L'inférence de type redonne une grande partie de la lisibilité des langages typés dynamiquement, tout en détectant les erreurs de type au moment de la compilation.

- Les fonctions peuvent être *génériques* : une seule fonction peut fonctionner sur des valeurs de nombreux types différents.

En Python et JavaScript, toutes les fonctions fonctionnent de cette façon naturellement : une fonction peut fonctionner sur n'importe quelle valeur qui possède les propriétés et les méthodes dont la fonction aura besoin. (C'est la caractéristique souvent appelée *typage de canard* : s'il charlatan comme un canard, c'est un canard.) Mais c'est précisément cette flexibilité qui rend si difficile pour ces langues de détecter les erreurs de type tôt; les tests sont souvent le seul moyen d'attraper de telles erreurs. Les fonctions génériques de Rust donnent au langage un degré de la même flexibilité, tout en détectant toutes les erreurs de type au moment de la compilation.

Malgré leur flexibilité, les fonctions génériques sont tout aussi efficaces que leurs homologues non génériques. Il n'y a aucun avantage inhérent en termes de performances à écrire, par exemple, une fonction spécifique pour chaque entier par rapport à l'écriture d'une fonction générique qui gère tous les entiers. Nous discuterons en détail des fonctions génériques au [chapitre 11](#).

Le reste de ce chapitre couvre les types de Rust de bas en haut, en commençant par des types numériques simples comme les entiers et les

valeurs à virgule flottante, puis en passant aux types qui contiennent plus de données : boîtes, tuples, tableaux et chaînes.

Voici un résumé des types que vous verrez dans Rust. [Le tableau 3-1](#) montre les types primitifs de Rust, certains types très courants de la bibliothèque standard et quelques exemples de types définis par l'utilisateur.

Tableau 3-1. Exemples de types dans Rust

Type	Description	Valeurs
i8 ,,, , ,,, i16 i 32 i64 il 28 u8 u16 u32 u64 u 128	Entiers signés et non signés, d'une largeur de bits donnée	42 , ,, , ,
		(littéral de l'octet) - 5i8 0x400u16 0o10 0i16 20_922_789_8 88_000u64 b'*' u8
isize, us ize	Entiers signés et non signés, de la même taille qu'une adresse sur la machine (32 ou 64 bits)	137 , , -0b0101_0010isi ze 0xffff_fc00usize
f32 , f64	Nombres à virgule flottante IEEE, simple et double précision	1.61803 , , 3.14f3 2 6.0221e23f64
bool	Booléen	true , false
char	Caractère Unicode, 32 bits de large	'*' , , , '\n' '字' '\x7f' '\u{CA0}'
(char, u 8, i32)	Tuple: types mixtes autorisés	('%', 0x7f, -1)
()	« Unité » (tuple vide)	()
struct S { x: f32 , y: f32 }	Structure de champ nommé	S { x: 120.0 , y: 209.0 }
struct T (i32, ch ar);	Structure de type Tuple	T(120 , 'x')

Type	Description	Valeurs
struct E;	Structure de type unité; n'a pas de champs	E
enum Att end { OnT ime, Late (u32) }	Énumération, type de données algébriques	Attend::Late(5), Attend::OnTime
Box<AttEnd>	Boîte : posséder un pointeur vers la valeur dans le tas	Box::new(Late(15))
&i32, &mut i32	Références partagées et modifiables : pointeurs non propriétaires qui ne doivent pas survivre à leur référent	&s.y, &mut v
String	Chaîne UTF-8, dimensionnée dynamiquement	"ラーメン: rame n".to_string()
&str	Référence à : pointeur non propriétaire vers le texte UTF-8 str	"そば: soba", &s[0..12]
[f64; 4], [u8; 256]	Tableau, longueur fixe; éléments tous du même type	[1.0, 0.0, 0.0, 1.0], [b' '; 256]
Vec<f64>	Vecteur, longueur variable; éléments tous du même type	vec![0.367, 2.718, 7.389]
&[u8], &mut [u8]	Référence à la tranche : référence à une partie d'un tableau ou d'un vecteur, comprenant le pointeur et la longueur	&v[10..20], &mut a[..]

Type	Description	Valeurs
Option<&str>	Valeur facultative : (absent) ou (présent, avec valeur Non nulle Some(v))	Some("Dr. ") , None
Result<u64, Error>	Résultat de l'opération susceptible d'échouer : soit une valeur de réussite, soit une erreur Ok(v) Err(e)	Ok(4096) , Err(Error::last_os_error())
&dyn Any, &mut dyn Read	Objet Trait : référence à toute valeur qui implémente un ensemble donné de méthodes	value as &dyn Any, &mut file as &mut dyn Read
fn(&str) -> bool	Pointeur vers la fonction	str::is_empty
(Les types de fermeture n'ont pas de forme écrite)	Fermeture	a, b { a*a + b*b }

La plupart de ces types sont traités dans le présent chapitre, à l'exception des éléments suivants :

- Nous donnons aux types leur propre chapitre, [le chapitre 9. struct](#)
- Nous donnons aux types énumérés leur propre chapitre, [le chapitre 10.](#)
- Nous décrivons les objets [traits au chapitre 11.](#)
- Nous décrivons l'essentiel de et ici, mais fournissons plus de détails au [chapitre 17. String &str](#)
- Nous couvrons les types de fonction et de fermeture au [chapitre 14.](#)

Types numériques à largeur fixe

La base du système de type de Rust est une collection de types numériques à largeur fixe, choisis pour correspondre aux types que

presque tous les processeurs modernes implémentent directement dans le matériel.

Les types numériques à largeur fixe peuvent déborder ou perdre en précision, mais ils sont adéquats pour la plupart des applications et peuvent être des milliers de fois plus rapides que les représentations telles que les entiers de précision arbitraire et les rationnels exacts. Si vous avez besoin de ce type de représentations numériques, elles sont prises en charge dans la caisse. `num`

Les noms des types numériques de Rust suivent un modèle régulier, épelant leur largeur en bits et la représentation qu'ils utilisent ([tableau 3-2](#)).

Tableau 3-2. Types numériques rust

Taille (bits)	Entier non signé	Entier signé	Virgule flottante
8	<code>u8</code>	<code>i8</code>	
16	<code>u16</code>	<code>i16</code>	
32	<code>u32</code>	<code>i32</code>	<code>f32</code>
64	<code>u64</code>	<code>i64</code>	<code>f64</code>
128	<code>u128</code>	<code>i128</code>	
Mot machine	<code>usize</code>	<code>isize</code>	

Ici, un *mot machine* est une valeur de la taille d'une adresse sur la machine sur laquelle le code s'exécute, 32 ou 64 bits.

Types d'entiers

Les types entiers non signés de Rust utilisent leur plage complète pour représenter des valeurs positives et nulles ([tableau 3-3](#)).

Tableau 3-3. Types entiers non signés Rust

Type	Gamme
u8	0 à 2^8-1 (0 à 255)
u16	0 à $2^{16}-1$ (0 à 65 535)
u32	0 à $2^{32}-1$ (0 à 4 294 967 295)
u64	0 à $2^{64}-1$ (0 à 18 446 744 073 709 551 615, soit 18 quintillions)
u128	0 à $2^{128}-1$ (0 à environ $3,4\times10^{38}$)
usiz	0 à $2^{32}-1$ ou $2^{64}-1$
e	

Les types entiers signés de Rust utilisent la représentation complémentaire des deux, en utilisant les mêmes modèles de bits que le type non signé correspondant pour couvrir une plage de valeurs positives et négatives ([tableau 3-4](#)).

Tableau 3-4. Types entiers signés Rust

Type	Gamme
i8	-2^7 à 2^7-1 (-128 à 127)
i16	-2^{15} à $2^{15}-1$ (-32 768 à 32 767)
i32	-2^{31} à $2^{31}-1$ (-2 147 483 648 à 2 147 483 647)
i64	-2^{63} à $2^{63}-1$ (-9 223 372 036 854 775 808 à 9 223 372 036 854 775 807)
i128	-2^{127} à $2^{127}-1$ (environ $-1,7\times10^{38}$ à $+1,7\times10^{38}$)
isiz	Soit -2^{31} à $2^{31}-1$ ou -2^{63} à $2^{63}-1$
e	

Rust utilise le type pour les valeurs d'octets. Par exemple, la lecture de données à partir d'un fichier binaire ou d'un socket génère un flux de valeurs. `u8` `u8`

Contrairement à C et C++, Rust traite les caractères comme distincts des types numériques : `a` n'est pas un `, ni un` (bien qu'il mesure 32 bits de long). Nous décrivons le type de Rust dans

[« Personnages ».](#) `char` `u8` `u32` `char`

Les types et sont analogues à et en C et C++. Leur précision correspond à la taille de l'espace d'adressage sur la machine cible : ils sont longs de 32 bits sur les architectures 32 bits et longs de 64 bits sur les architectures 64 bits. Rust exige que les indices de tableau soient des valeurs. Les valeurs représentant la taille des tableaux ou des vecteurs ou le nombre d'éléments dans une structure de données ont également généralement le type. `usize` `isize` `size_t` `ptrdiff_t` `usize` `usize`

Les littéraux entiers dans Rust peuvent prendre un suffixe indiquant leur type : `est` une valeur, `et` est un `.` Si un littéral entier n'a pas de suffixe de type, Rust reporte la détermination de son type jusqu'à ce qu'il trouve la valeur utilisée d'une manière qui l'épingle : stockée dans une variable d'un type particulier, transmise à une fonction qui attend un type particulier, par rapport à une autre valeur d'un type particulier, ou quelque chose comme ça. En fin de compte, si plusieurs types peuvent fonctionner, Rust par défaut si cela fait partie des possibilités. Sinon, Rust signale l'ambiguïté comme une erreur. `42u8` `u8` `1729isize` `isize` `i32`

Les préfixes `,` et désignent des littéraux hexadécimaux, octaux et binaires. `0x` `0o` `0b`

Pour rendre les nombres longs plus lisibles, vous pouvez insérer des traits de soulignement parmi les chiffres. Par exemple, vous pouvez écrire la plus grande valeur sous la forme `.` L'emplacement exact des traits de soulignement n'est pas significatif, vous pouvez donc diviser les nombres hexadécimaux ou binaires en groupes de quatre chiffres au lieu de trois, comme dans `,` ou définir le suffixe de type à partir des chiffres, comme dans `.` Quelques exemples de littéraux entiers sont illustrés dans [le tableau 3-5.](#) `u32` `4_294_967_295` `0xffff_ffff` `127_u8`

Tableau 3-5. Exemples de littéraux entiers

Littéral	Type	Valeur décimale
116i8	i8	116
0xcafeu32	u32	51966
0b0010_1010	Déduit	42
0o106	Déduit	70

Bien que les types numériques et le type soient distincts, Rust fournit des *littéraux d'octets*, des littéraux de type caractère pour les valeurs: représente le code ASCII pour le caractère , en tant que valeur. Par exemple, puisque le code ASCII pour est 65, les littéraux et sont exactement équivalents. Seuls les caractères ASCII peuvent apparaître dans les littéraux d'octets. `char u8 b'X' X u8 A b'A' 65u8`

Il y a quelques caractères que vous ne pouvez pas simplement placer après la seule citation, car ce serait soit syntaxiquement ambigu, soit difficile à lire. Les caractères [du tableau 3-6](#) ne peuvent être écrits qu'à l'aide d'une notation de remplacement, introduite par une barre oblique inverse.

Tableau 3-6. Caractères nécessitant une notation de remplacement

Personnage	Littéral octet	Équivalent numérique
Devis unique, '	b'\ ''	39u8
Backslash \	b'\\\'	92u8
Newline	b'\n'	10u8
Retour chariot	b'\r'	13u8
Onglet	b'\t'	9u8

Pour les caractères difficiles à écrire ou à lire, vous pouvez écrire leur code en hexadécimal à la place. Un littéral octet de la forme , où est tout nombre hexadécimal à deux chiffres, représente l'octet dont la valeur est

. Par exemple, vous pouvez écrire un littéral d'octet pour le caractère de contrôle ASCII « escape » comme , puisque le code ASCII pour « escape » est 27, ou 1B en hexadécimal. Étant donné que les littéraux d'octets ne sont qu'une autre notation pour les valeurs, demandez-vous si un littéral numérique simple pourrait être plus lisible : il est probablement logique de l'utiliser au lieu de simplement seulement lorsque vous voulez souligner que la valeur représente un code

```
ASCII.b'\xHH' HH HH b'\x1b' u8 b'\x1b' 27
```

Vous pouvez convertir d'un type entier à un autre à l'aide de l'opérateur. Nous expliquons comment fonctionnent les conversions dans [«Type Casts»](#), mais voici quelques exemples: as

```
assert_eq!( 10_i8 as u16,      10_u16); // in range
assert_eq!( 2525_u16 as i16,   2525_i16); // in range

assert_eq!(-1_i16 as i32,     -1_i32); // sign-extended
assert_eq!(65535_u16 as i32,  65535_i32); // zero-extended

// Conversions that are out of range for the destination
// produce values that are equivalent to the original modulo 2^N,
// where N is the width of the destination in bits. This
// is sometimes called "truncation."
assert_eq!( 1000_i16 as u8,    232_u8);
assert_eq!( 65535_u32 as i16, -1_i16);

assert_eq!(-1_i8 as u8,       255_u8);
assert_eq!( 255_u8 as i8,     -1_i8);
```

La bibliothèque standard fournit certaines opérations en tant que méthodes sur des entiers. Par exemple:

```
assert_eq!(2_u16.pow(4), 16);           // exponentiation
assert_eq!((-4_i32).abs(), 4);          // absolute value
assert_eq!(0b101101_u8.count_ones(), 4); // population count
```

Vous pouvez les trouver dans la documentation en ligne. Notez toutefois que la documentation contient des pages distinctes pour le type lui-même sous « (type primitif) », et pour le module dédié à ce type (recherchez « ».i32 std::i32

Dans le code réel, vous n'aurez généralement pas besoin d'écrire les suffixes de type comme nous l'avons fait ici, car le contexte déterminera le

type. Lorsque ce n'est pas le cas, cependant, les messages d'erreur peuvent être surprenants. Par exemple, les éléments suivants ne sont pas compilés :

```
println!("{}", (-4).abs());
```

Rust se plaint:

```
error: can't call method `abs` on ambiguous numeric type `{}integer`
```

Cela peut être un peu déroutant: tous les types d'entiers signés ont une méthode, alors quel est le problème? Pour des raisons techniques, Rust veut savoir exactement quel type entier a une valeur avant d'appeler les propres méthodes du type. La valeur par défaut de s'applique uniquement si le type est encore ambigu après que tous les appels de méthode ont été résolus, il est donc trop tard pour aider ici. La solution consiste à préciser le type que vous souhaitez, soit avec un suffixe, soit en utilisant la fonction d'un type spécifique : `abs i32`

```
println!("{}", (-4_i32).abs());  
println!("{}", i32::abs(-4));
```

Notez que les appels de méthode ont une priorité plus élevée que les opérateurs de préfixe unaire, alors soyez prudent lorsque vous appliquez des méthodes à des valeurs annulées. Sans les parenthèses dans la première instruction, appliquerait la méthode à la valeur positive , produisant positif , puis annulerait cela, produisant . -4_i32 - 4_i32.abs() abs 4 4 -4

Arithmétique vérifiée, enveloppante, saturée et débordante

Lorsqu'une opération arithmétique d'entier déborde, Rust panique, dans une version de débogage. Dans une version release, l'opération *s'enroule* : elle produit la valeur équivalente au résultat mathématiquement correct modulo la plage de la valeur. (Dans aucun des deux cas, le comportement de débordement n'est défini, comme c'est le cas en C et C++.)

Par exemple, le code suivant panique dans une version de débogage :

```
let mut i = 1;  
loop {
```

```

    i *= 10; // panic: attempt to multiply with overflow
              // (but only in debug builds!)
}

```

Dans une version release, cette multiplication s'enroule à un nombre négatif et la boucle s'exécute indéfiniment.

Lorsque ce comportement par défaut n'est pas ce dont vous avez besoin, les types entiers fournissent des méthodes qui vous permettent d'épeler exactement ce que vous voulez. Par exemple, les paniques suivantes dans n'importe quelle version :

```

let mut i: i32 = 1;
loop {
    // panic: multiplication overflowed (in any build)
    i = i.checked_mul(10).expect("multiplication overflowed");
}

```

Ces méthodes arithmétiques d'entiers se répartissent en quatre catégories générales :

- Les opérations *vérifiées* renvoient un résultat : si le résultat mathématiquement correct peut être représenté comme une valeur de ce type, ou s'il ne le peut pas. Par exemple: Option Some(v) None

```

// The sum of 10 and 20 can be represented as a u8.
assert_eq!(10_u8.checked_add(20), Some(30));

// Unfortunately, the sum of 100 and 200 cannot.
assert_eq!(100_u8.checked_add(200), None);

// Do the addition; panic if it overflows.
let sum = x.checked_add(y).unwrap();

// Oddly, signed division can overflow too, in one particular case.
// A signed n-bit type can represent  $-2^{n-1}$ , but not  $2^{n-1}$ .
assert_eq!((-128_i8).checked_div(-1), None);

```

- Les opérations *d'encapsulation* renvoient la valeur équivalente au résultat mathématiquement correct modulo la plage de la valeur :

```

// The first product can be represented as a u16;
// the second cannot, so we get 250000 modulo  $2^{16}$ .
assert_eq!(100_u16.wrapping_mul(200), 20000);

```

```

assert_eq!(500_u16.wrapping_mul(500), 53392);

// Operations on signed types may wrap to negative values.
assert_eq!(500_i16.wrapping_mul(500), -12144);

// In bitwise shift operations, the shift distance
// is wrapped to fall within the size of the value.
// So a shift of 17 bits in a 16-bit type is a shift
// of 1.
assert_eq!(5_i16.wrapping_shl(17), 10);

```

Comme expliqué, c'est ainsi que les opérateurs arithmétiques ordinaires se comportent dans les versions de version. L'avantage de ces méthodes est qu'elles se comportent de la même manière dans toutes les constructions.

- Les opérations *de saturation* renvoient la valeur représentable la plus proche du résultat mathématiquement correct. En d'autres termes, le résultat est « serré » aux valeurs maximales et minimales que le type peut représenter:

```

assert_eq!(32760_i16.saturating_add(10), 32767);
assert_eq!((-32760_i16).saturating_sub(10), -32768);

```

Il n'y a pas de méthodes de division saturée, de reste ou de décalage binaire.

- Les opérations *de débordement* renvoient un tuple , où est ce que la version d'encapsulation de la fonction renverrait, et indique si un débordement s'est produit : (result, overflowed) result overflowed bool

```

assert_eq!(255_u8.overflowing_sub(2), (253, false));
assert_eq!(255_u8.overflowing_add(2), (1, true));

```

overflowing_shl et s'écartent un peu du motif : ils ne retournent vrai que si la distance de décalage était aussi grande ou plus grande que la largeur de bit du type lui-même. Le décalage réel appliqué est le décalage demandé modulo la largeur de bit du type
:overflowing_shr overflowed

```

// A shift of 17 bits is too large for `u16`, and 17 modulo 16 is 1.
assert_eq!(5_u16.overflowing_shl(17), (10, true));

```

Les noms d'opération qui suivent le , , , ou préfixe sont indiqués dans [le tableau 3-7](#). `checked_` `wrapping_` `saturating_` `overflowing_`

Tableau 3-7. Noms des opérations

Opération	Suffixe du nom	Exemple
Addition	add	<code>100_i8.checked_add(27) == Some(127)</code>
Soustraction	sub	<code>10_u8.checked_sub(11) == None</code>
Multiplication	mul	<code>128_u8.saturating_mul(3) == 255</code>
Division	div	<code>64_u16.wrapping_div(8) == 8</code>
Reste	rem	<code>(-32768_i16).wrapping_rem(-1) == 0</code>
Négation	neg	<code>(-128_i8).checked_neg() = None</code>
Valeur absolue	abs	<code>(-32768_i16).wrapping_abs() == -32768</code>
Exponentiation	pow	<code>3_u8.checked_pow(4) == Some(81)</code>
Décalage bito-gauche	shl	<code>10_u32.wrapping_shl(34) = 40</code>
Décalage binaire vers la droite	shr	<code>40_u64.wrapping_shr(66) = 10</code>

Types à virgule flottante

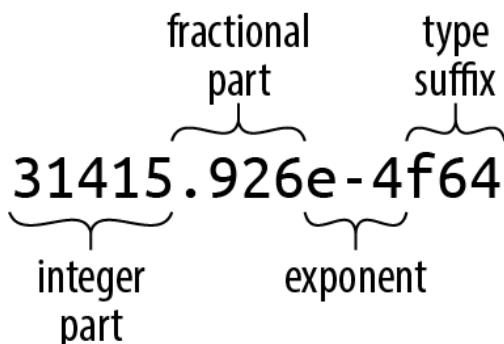
Rust fournit des types à virgule flottante IEEE à simple et double précision. Ces types comprennent des infinis positifs et négatifs, des valeurs nulles positives et négatives distinctes et *une valeur non numérique* ([tableau 3-8](#)).

Tableau 3-8. Types à virgule flottante IEEE simple et double précision

Type	Précision	Gamme
f32	Ieee précision unique (au moins 6 chiffres décimaux)	Environ $-3,4 \times 10^{38}$ à $+3,4 \times 10^{38}$
f64	Double précision IEEE (au moins 15 chiffres décimaux)	Environ $-1,8 \times 10^{308}$ à $+1,8 \times 10^{308}$

Rust et correspondent aux types et en C et C++ (dans les implémentations qui prennent en charge la virgule flottante IEEE) ainsi que Java (qui utilise toujours la virgule flottante IEEE). `f32 f64 float double`

Les littéraux à virgule flottante ont la forme générale schématisée à [la figure 3-1](#).



Graphique 3-1. Littéral à virgule flottante

Chaque partie d'un nombre à virgule flottante après la partie entière est facultative, mais au moins une partie fractionnaire, exposant ou suffixe de type doit être présente pour la distinguer d'un littéral entier. La partie fractionnaire peut être constituée d'une virgule décimale unique, de même qu'une constante à virgule flottante valide. 5 .

Si un littéral à virgule flottante n'a pas de suffixe de type, Rust vérifie le contexte pour voir comment les valeurs sont utilisées, tout comme il le fait pour les littéraux entiers. S'il trouve finalement que l'un ou l'autre type à virgule flottante pourrait convenir, il choisit par défaut. `f64`

Aux fins de l'inférence de type, Rust traite les littéraux entiers et les littéraux à virgule flottante comme des classes distinctes : il ne déduira ja-

mais un type à virgule flottante pour un littéral entier, ou vice versa. [Le tableau 3-9](#) présente quelques exemples de littéraux à virgule flottante.

Tableau 3-9. Exemples de littéraux à virgule flottante

Littéral	Type	Valeur mathématique
-1.5625	Déduit	$-(1 \frac{9}{16})$
2.	Déduit	2
0.25	Déduit	$\frac{1}{4}$
1e4	Déduit	10,000
40f32	f32	40
9.109_383_56e-31f64	f64	Environ $9,10938356 \times 10^{-31}$

Les types et ont des constantes associées pour les valeurs spéciales requises par l'IEEE telles que , (infini négatif), (la valeur non numérique), et et (les plus grandes et les plus petites valeurs finies): f32 f64 INFINITY NEG_INFINITY NAN MIN MAX

```
assert!((-1. / f32::INFINITY).is_sign_negative());
assert_eq!(-f32::MIN, f32::MAX);
```

Les types et fournissent un complément complet de méthodes pour les calculs mathématiques; par exemple, est la racine carrée à double précision de deux. Quelques exemples : f32 f64 2f64.sqrt()

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // exactly 5.0, per IEEE
assert_eq!((-1.01f64).floor(), -2.0);
```

Encore une fois, les appels de méthode ont une priorité plus élevée que les opérateurs de préfixe, alors veillez à mettre correctement entre parenthèses les appels de méthode sur les valeurs annulées.

Les modules et fournissent diverses constantes mathématiques couramment utilisées comme , , et la racine carrée de deux. std::f32::consts std::f64::consts E PI

Lorsque vous recherchez dans la documentation, rappelez-vous qu'il existe des pages pour les deux types eux-mêmes, nommées " (type primitif) » et « (type primitif) », et les modules pour chaque type, et

```
.f32 f64 std::f32 std::f64
```

Comme pour les entiers, vous n'aurez généralement pas besoin d'écrire des suffixes de type sur des littéraux à virgule flottante dans du code réel, mais lorsque vous le faites, il suffit de mettre un type sur le littéral ou la fonction:

```
println!("{}", (2.0_f64).sqrt());  
println!("{}", f64::sqrt(2.0));
```

Contrairement à C et C++, Rust n'effectue presque aucune conversion numérique implicitement. Si une fonction attend un argument, c'est une erreur de passer une valeur comme argument. En fait, Rust ne convertira même pas implicitement une valeur en valeur, même si chaque valeur est aussi une valeur. Mais vous pouvez toujours écrire des conversions *explicites* à l'aide de l'opérateur :, ou `.f64 i32 i16 i32 i16 i32 as i as f64 x as i32`

L'absence de conversions implicites rend parfois une expression Rust plus verbeuse que ne le serait le code C ou C++ analogue. Cependant, les conversions implicites d'entiers ont un historique bien établi de bogues et de failles de sécurité, en particulier lorsque les entiers en question représentent la taille de quelque chose en mémoire et qu'un débordement imprévu se produit. D'après notre expérience, l'acte d'écrire des conversions numériques dans Rust nous a alertés sur des problèmes que nous aurions autrement manqués.

Nous expliquons exactement comment les conversions se comportent dans [« Type Casts »](#).

Le type de bool

Le type booléen de Rust, , a les deux valeurs habituelles pour ces types, et . Les opérateurs de comparaison aiment et produisent des résultats: la valeur de est .bool true false == < bool 2 < 5 true

De nombreux langages sont indulgents quant à l'utilisation de valeurs d'autres types dans des contextes qui nécessitent une valeur booléenne : C et C++ convertissent implicitement des caractères, des entiers, des nom-

bres à virgule flottante et des pointeurs en valeurs booléennes, de sorte qu'ils peuvent être utilisés directement comme condition dans une instruction ou. Python autorise les chaînes, les listes, les dictionnaires et même les ensembles dans les contextes booléens, traitant ces valeurs comme true si elles ne sont pas vides. La rouille, cependant, est très stricte: les structures de contrôle aiment et exigent que leurs conditions soient des expressions, tout comme les opérateurs logiques de court-circuit et . Vous devez écrire , pas simplement

```
.if while if while bool && || if x != 0 { ... } if x { ... }
```

L'opérateur de Rust peut convertir des valeurs en types entiers : as bool

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

Cependant, ne convertira pas dans l'autre sens, des types numériques à .

Au lieu de cela, vous devez écrire une comparaison explicite comme

```
.as bool x != 0
```

Bien qu'un seul bit n'ait besoin que d'un seul bit pour le représenter, Rust utilise un octet entier pour une valeur en mémoire, de sorte que vous pouvez créer un pointeur vers celui-ci. bool bool

Caractères

Le type de caractère de Rust représente un seul caractère Unicode, sous la forme d'une valeur de 32 bits. char

Rust utilise le type pour les caractères uniques de manière isolée, mais utilise le codage UTF-8 pour les chaînes et les flux de texte. Ainsi, a représente son texte comme une séquence d'octets UTF-8, et non comme un tableau de caractères. char String

Les littéraux de caractères sont des caractères entre guillemets simples, comme ou . Vous pouvez utiliser toute l'étendue d'Unicode: est un littéral représentant le kanji japonais pour *sabi* (rouille). '8' '!' '錆' char

Comme pour les littéraux d'octets, des échappements de barre oblique inverse sont requis pour quelques caractères ([Tableau 3-10](#)).

Tableau 3-10. Caractères nécessitant des échappements de barre oblique inverse

Personnage	Caractère rouillé littéral
Devis unique, '	'\\ ''
Backslash \	'\\\\'
Newline	'\\n'
Retour chariot	'\\r'
Onglet	'\\t'

Si vous préférez, vous pouvez écrire le point de code Unicode d'un caractère en hexadécimal :

- Si le point de code du caractère se situe dans la plage U+0000 à U+007F (c'est-à-dire s'il est tiré du jeu de caractères ASCII), vous pouvez écrire le caractère sous la forme , où est un nombre hexadécimal à deux chiffres. Par exemple, les caractères littéraux et sont équivalents, car le point de code du caractère est 42, ou 2A en hexadécimal. '\xHH' HH '*' '\x2A' *
- Vous pouvez écrire n'importe quel caractère Unicode comme , où est un nombre hexadécimal d'une longueur maximale de six chiffres, avec des traits de soulignement autorisés pour le regroupement comme d'habitude. Par exemple, le caractère littéral représente le caractère « ಂ », un caractère Kannada utilisé dans l'aspect Unicode de désapprobation, « ಂ_ಂ ». Le même littéral pourrait également être simplement écrit comme . '\u{HHHHHH}' HHHHHH '\u{CA0}' 'ಂ'

A contient toujours un point de code Unicode dans la plage 0x0000 à 0xD7FF ou 0xE000 à 0x10FFFF. A n'est jamais une moitié de paire de substitution (c'est-à-dire un point de code dans la plage 0xD800 à 0xDFFF), ou une valeur en dehors de l'espace de code Unicode (c'est-à-dire supérieure à 0x10FFFF). Rust utilise le système de type et des contrôles dynamiques pour s'assurer que les valeurs sont toujours dans la plage autorisée. `char` `char` `char`

Rust ne convertit jamais implicitement entre et tout autre type. Vous pouvez utiliser l'opérateur de conversion pour convertir a en un type entier ;

pour les types inférieurs à 32 bits, les bits supérieurs de la valeur du caractère sont tronqués : `char as char`

```
assert_eq!('*' as i32, 42);
assert_eq!(‘Ø’ as u16, 0xca0);
assert_eq!(‘Ø’ as i8, -0x60); // U+0CA0 truncated to eight bits, signed
```

Aller dans l'autre sens, est le seul type que l'opérateur convertira en : Rust a l'intention que l'opérateur n'effectue que des conversions bon marché et infaillibles, mais chaque type d'entier autre que les valeurs qui ne sont pas autorisées points de code Unicode, de sorte que ces conversions nécessiteraient des vérifications d'exécution. Au lieu de cela, la fonction de bibliothèque standard prend n'importe quelle valeur et renvoie un : si le n'est pas un point de code Unicode autorisé, alors renvoie ; sinon, il renvoie , où est le

```
résultat.u8 as char as u8 std::char::from_u32 u32 Option<char> u32 from_u32 None Some(c) c char
```

La bibliothèque standard fournit des méthodes utiles sur les caractères, que vous pouvez rechercher dans la documentation en ligne sous « (type primitif) » et le module « ». Par exemple: `char std::char`

```
assert_eq!(‘*’.is_alphabetic(), false);
assert_eq!(‘Ø’.is_alphabetic(), true);
assert_eq!(‘8’.to_digit(10), Some(8));
assert_eq!(‘Ø’.len_utf8(), 3);
assert_eq!(std::char::from_digit(2, 10), Some('2'));
```

Naturellement, les caractères isolés ne sont pas aussi intéressants que les chaînes et les flux de texte. Nous décrirons le type standard de Rust et la gestion du texte en général dans [« Types de chaînes »](#). `String`

Tuples

Un *tuple* est une paire, ou triple, quadruple, quintuple, etc. (d'où *n-tuple*, ou *tuple*), de valeurs de types assortis. Vous pouvez écrire un tuple sous la forme d'une séquence d'éléments, séparés par des virgules et entourés de parenthèses. Par exemple, est un tuple dont le premier élément est une chaîne allouée statiquement et dont le second est un entier ; son type est . Étant donné une valeur tuple , vous pouvez accéder à ses éléments en

```
tant que , , et ainsi de suite. ("Brazil" , 1985) (&str,  
i32) t t.0 t.1
```

Dans une certaine mesure, les tuples ressemblent à des tableaux : les deux types représentent une séquence ordonnée de valeurs. De nombreux langages de programmation confondent ou combinent les deux concepts, mais dans Rust, ils sont complètement séparés. D'une part, chaque élément d'un tuple peut avoir un type différent, alors que les éléments d'un tableau doivent être tous du même type. De plus, les tuples n'autorisent que les constantes en tant qu'indices, comme . Vous ne pouvez pas écrire ou obtenir le ème élément. t.4 t.i t[i] i

Le code Rust utilise souvent des types de tuple pour renvoyer plusieurs valeurs à partir d'une fonction. Par exemple, la méthode sur les tranches de chaîne, qui divise une chaîne en deux moitiés et les renvoie toutes les deux, est déclarée comme suit : `split_at`

```
fn split_at(&self, mid: usize) -> (&str, &str);
```

Le type de retour est un tuple de deux tranches de chaîne. Vous pouvez utiliser la syntaxe de correspondance de modèle pour affecter chaque élément de la valeur renvoyée à une variable différente : `(&str, &str)`

```
let text = "I see the eigenvalue in thine eye";  
let (head, tail) = text.split_at(21);  
assert_eq!(head, "I see the eigenvalue ");  
assert_eq!(tail, "in thine eye");
```

C'est plus lisible que l'équivalent :

```
let text = "I see the eigenvalue in thine eye";  
let temp = text.split_at(21);  
let head = temp.0;  
let tail = temp.1;  
assert_eq!(head, "I see the eigenvalue ");  
assert_eq!(tail, "in thine eye");
```

Vous verrez également des tuples utilisés comme une sorte de type de structure à drame minimal. Par exemple, dans le programme Mandelbrot du [chapitre 2](#), nous devions passer la largeur et la hauteur de l'image aux fonctions qui la tracent et l'écrivent sur le disque. Nous pourrions déclarer une structure avec et membres, mais c'est une notation assez lourde

pour quelque chose d'aussi évident, alors nous avons juste utilisé un tuple: width height

```
// Write the buffer `pixels`, whose dimensions are given by `bounds`,  
// file named `filename`.  
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))  
    -> Result<(), std::io::Error>  
{ ... }
```

Le type du paramètre est , un tuple de deux valeurs. Certes, nous pourrions tout aussi bien écrire des paramètres séparés, et le code machine serait à peu près le même dans les deux cas. C'est une question de clarté. Nous considérons la taille comme une valeur, pas deux, et l'utilisation d'un tuple nous permet d'écrire ce que nous voulons
dire. bounds (usize, usize) usize width height

L'autre type de tuple couramment utilisé est le zéro-tuple . C'est ce qu'on appelle traditionnellement le *type d'unité* car il n'a qu'une seule valeur, également écrite . Rust utilise le type d'unité où il n'y a pas de valeur significative à transporter, mais le contexte nécessite néanmoins une sorte de type. () ()

Par exemple, une fonction qui ne renvoie aucune valeur a un type de retour de . La fonction de la bibliothèque standard n'a pas de valeur de retour significative ; il ne fait qu'échanger les valeurs de ses deux arguments. La déclaration pour se lit comme suit:

```
() std::mem::swap std::mem::swap
```

```
fn swap<T>(x: &mut T, y: &mut T);
```

Le moyen qui est *générique* : vous pouvez l'utiliser sur des références à des valeurs de n'importe quel type. Mais la signature omet complètement le type de retour de ', qui est un raccourci pour renvoyer le type d'unité:

```
<T> swap T swap
```

```
fn swap<T>(x: &mut T, y: &mut T) -> ();
```

De même, l'exemple que nous avons mentionné précédemment a un type de retour de , ce qui signifie que la fonction renvoie une valeur en cas de problème, mais ne renvoie aucune valeur en cas de réussite. write_image Result<(), std::io::Error> std::io::Error

Si vous le souhaitez, vous pouvez inclure une virgule après le dernier élément d'un tuple : les types et sont équivalents, tout comme les expressions et . Rust permet systématiquement une virgule de fin supplémentaire partout où des virgules sont utilisées : arguments de fonction, tableaux, définitions struct et enum, etc. Cela peut sembler étrange pour les lecteurs humains, mais cela peut rendre les diffs plus faciles à lire lorsque des entrées sont ajoutées et supprimées à la fin d'une liste.

```
(&str, i32,) (&str, i32) ("Brazil", 1985,) ("Brazil",  
1985)
```

Par souci de cohérence, il existe même des tuples qui contiennent une seule valeur. Le littéral est un tuple contenant une seule chaîne ; son type est . Ici, la virgule après la valeur est nécessaire pour distinguer le tuple singleton d'une simple expression entre parenthèses. ("lonely hearts",) (&str,)

Types de pointeurs

Rust a plusieurs types qui représentent des adresses mémoire.

C'est une grande différence entre Rust et la plupart des langues avec garbage collection. En Java, si contient un champ , alors est une référence à un autre objet créé séparément. Les objets ne contiennent jamais physiquement d'autres objets en Java. class Rectangle Vector2D
upperLeft; upperLeft Vector2D

La rouille est différente. Le langage est conçu pour aider à réduire les allocations au minimum. Les valeurs s'imbriquent par défaut. La valeur est stockée sous la forme de quatre entiers adjacents. Si vous la stockez dans une variable locale, vous avez une variable locale de quatre entiers de large. Rien n'est alloué dans le tas. ((0, 0), (1440, 900))

C'est excellent pour l'efficacité de la mémoire, mais par conséquent, lorsqu'un programme Rust a besoin de valeurs pour pointer vers d'autres valeurs, il doit utiliser explicitement des types de pointeur. La bonne nouvelle est que les types de pointeurs utilisés dans Rust sécurisé sont contraints d'éliminer les comportements non définis, de sorte que les pointeurs sont beaucoup plus faciles à utiliser correctement dans Rust qu'en C++.

Nous aborderons ici trois types de pointeurs : les références, les boîtes et les pointeurs dangereux.

Références

Une valeur de type (prononcé « ref String ») est une référence à une valeur, `a` est une référence à `un`, et ainsi de suite.

```
&String String &i32 i32
```

Il est plus facile de commencer en pensant aux références comme le type de pointeur de base de Rust. Au moment de l'exécution, une référence à `an` est un mot de machine unique contenant l'adresse du `x`, qui peut se trouver sur la pile ou dans le tas. L'expression produit une référence à `x`; dans la terminologie Rust, nous disons qu'il *emprunte une référence à `x`*. Étant donné une référence `x`, l'expression fait référence à la valeur pointée vers. Ceux-ci ressemblent beaucoup aux opérateurs et en C++ et C++. Et comme un pointeur C, une référence ne libère pas automatiquement de ressources lorsqu'elle sort de son champ d'application.

```
i32 i32 &x x r *r r & *
```

Contrairement aux pointeurs C, cependant, les références Rust ne sont jamais nulles : il n'y a tout simplement aucun moyen de produire une référence nulle dans Rust sûr. Et contrairement à C, Rust suit la propriété et la durée de vie des valeurs, de sorte que les erreurs telles que les pointeurs pendants, les doubles libres et l'invalidation du pointeur sont exclues au moment de la compilation.

Les références à la rouille se déclinent en deux saveurs :

`&T`

Une référence immuable et partagée. Vous pouvez avoir de nombreuses références partagées à une valeur donnée à la fois, mais elles sont en lecture seule : il est interdit de modifier la valeur vers laquelle elles pointent, comme dans C. `const T*`

`&mut T`

Une référence mutable et exclusive. Vous pouvez lire et modifier la valeur vers laquelle il pointe, comme avec `a` en C. Mais tant que la référence existe, vous n'avez peut-être pas d'autres références d'aucune sorte à cette valeur. En fait, la seule façon d'accéder à la valeur est via la référence modifiable. `T*`

Rust utilise cette dichotomie entre les références partagées et mutables pour appliquer une règle « un seul écrivain *ou* plusieurs lecteurs »: soit vous pouvez lire et écrire la valeur, soit elle peut être partagée par n'im-

porte quel nombre de lecteurs, mais jamais les deux en même temps. Cette séparation, appliquée par des contrôles au moment de la compilation, est au cœur des garanties de sécurité de Rust. [Le chapitre 5](#) explique les règles de Rust pour une utilisation de référence sûre.

Boîtes

La façon la plus simple d'allouer une valeur dans le tas est d'utiliser

`:Box::new`

```
let t = (12, "eggs");
let b = Box::new(t); // allocate a tuple in the heap
```

Le type de `t` est `(i32, &str)`, donc le type de `b` est `Box<(i32, &str)>`. L'appel à `Box::new` alloue suffisamment de mémoire pour contenir le tuple sur le tas. Lorsqu'elle sort de la portée, la mémoire est libérée immédiatement, sauf si elle a été *déplacée*, par exemple en la renvoyant. Les mouvements sont essentiels à la façon dont Rust gère les valeurs allouées au tas ; nous expliquons tout cela en détail au [chapitre 4](#).

`t (i32, &str) b Box<(i32, &str)> Box::new b b`

Pointeurs bruts

Rust a également les types de pointeurs bruts et `Box`. Les pointeurs bruts sont vraiment comme les pointeurs en C++. L'utilisation d'un pointeur brut n'est pas sûre, car Rust ne fait aucun effort pour suivre ce qu'il pointe. Par exemple, les pointeurs bruts peuvent être null ou pointer vers une mémoire qui a été libérée ou qui contient maintenant une valeur d'un type différent. Toutes les erreurs de pointeur classiques de C++ sont proposées pour votre plaisir.

*`mut` `T` *`const` `T`

Toutefois, vous ne pouvez déréférencer que les pointeurs bruts d'un bloc. Un bloc est le mécanisme d'opt-in de Rust pour les fonctionnalités linguistiques avancées dont la sécurité dépend de vous. Si votre code n'a pas de blocs (ou si ceux qu'il fait sont écrits correctement), alors les garanties de sécurité que nous soulignons tout au long de ce livre sont toujours valables. Pour plus de détails, voir [le chapitre 22](#).

Tableaux, vecteurs et tranches

Rust a trois types pour représenter une séquence de valeurs en mémoire :

- Le type représente un tableau de valeurs, chacune de type . La taille d'un tableau est une constante déterminée au moment de la compilation et fait partie du type ; vous ne pouvez pas ajouter de nouveaux éléments ou réduire un tableau. [T; N] N T
- Le type , appelé *vecteur de Ts*, est une séquence de valeurs de type cultivable allouée dynamiquement. Les éléments d'un vecteur vivent sur le tas, vous pouvez donc redimensionner les vecteurs à volonté : poussez-y de nouveaux éléments, ajoutez-y d'autres vecteurs, supprimez des éléments, etc. Vec<T> T
- Les types et , appelés *tranche partagée de Ts* et *tranche mutable de Ts*, sont des références à une série d'éléments qui font partie d'une autre valeur, comme un tableau ou un vecteur. Vous pouvez considérer une tranche comme un pointeur vers son premier élément, ainsi qu'un décompte du nombre d'éléments auxquels vous pouvez accéder à partir de ce point. Une tranche modifiable vous permet de lire et de modifier des éléments, mais ne peut pas être partagée ; une tranche partagée vous permet de partager l'accès entre plusieurs lecteurs, mais ne vous permet pas de modifier des éléments. &[T] &mut [T] &mut [T] &[T]

Étant donné une valeur de l'un de ces trois types, l'expression donne le nombre d'éléments dans , et se réfère au ème élément de . Le premier élément est , et le dernier élément est . Contrôles de rouille qui se situent toujours dans cette fourchette; si ce n'est pas le cas, l'expression panique. La longueur de peut être nulle, auquel cas toute tentative d'indexation paniquera. doit être une valeur; vous ne pouvez utiliser aucun autre type d'entier comme index. v v.len() v v[i] i v v[0] v[v.len() - 1] i v i usize

Tableaux

Il existe plusieurs façons d'écrire des valeurs de tableau. Le plus simple est d'écrire une série de valeurs entre crochets :

```
let lazy_caterer: [u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

Pour le cas courant d'un tableau long rempli d'une valeur, vous pouvez écrire , où est la valeur que chaque élément doit avoir, et est la longueur.

Par exemple, est un tableau de 10 000 éléments, tous définis sur : [v;

N] V N [true; 10000] bool true

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
        }
    }
}

assert!(sieve[211]);
assert!(!sieve[9876]);
```

Vous verrez cette syntaxe utilisée pour les tampons de taille fixe: peut être un tampon d'un kilo-octet, rempli de zéros. Rust n'a pas de notation pour un tableau non initialisé. (En général, Rust garantit que le code ne peut jamais accéder à aucune sorte de valeur non initialisée.) [0u8;
1024]

La longueur d'un tableau fait partie de son type et est fixée au moment de la compilation. Si est une variable, vous ne pouvez pas écrire pour obtenir un tableau d'éléments. Lorsque vous avez besoin d'un tableau dont la longueur varie au moment de l'exécution (et c'est généralement le cas), utilisez plutôt un vecteur. n [true; n] n

Les méthodes utiles que vous souhaitez voir sur les tableaux (itération sur les éléments, recherche, tri, remplissage, filtrage, etc.) sont toutes fournies sous forme de méthodes sur des tranches, et non de tableaux. Mais Rust convertit implicitement une référence à un tableau en tranche lors de la recherche de méthodes, de sorte que vous pouvez appeler directement n'importe quelle méthode de tranche sur un tableau :

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

Ici, la méthode est en fait définie sur des tranches, mais comme elle prend son opérande par référence, Rust produit implicitement une tranche se référant à l'ensemble du tableau et la transmet pour opérer. En fait, la

méthode que nous avons mentionnée précédemment est également une méthode de tranche. Nous couvrons les tranches plus en détail dans « [Tranches](#) ».

```
sort &mut [i32] sort len
```

Vecteurs

Un vecteur est un tableau redimensionnable d'éléments de type , alloués sur le tas.

```
Vec<T> T
```

Il existe plusieurs façons de créer des vecteurs. Le plus simple est d'utiliser la macro, ce qui nous donne une syntaxe pour les vecteurs qui ressemble beaucoup à un littéral de tableau: vec !

```
let mut primes = vec![2, 3, 5, 7];
assert_eq!(primes.iter().product::<i32>(), 210);
```

Mais bien sûr, il s'agit d'un vecteur, pas d'un tableau, nous pouvons donc y ajouter des éléments dynamiquement:

```
primes.push(11);
primes.push(13);
assert_eq!(primes.iter().product::<i32>(), 30030);
```

Vous pouvez également créer un vecteur en répétant une valeur donnée un certain nombre de fois, toujours en utilisant une syntaxe qui imite les littéraux de tableau :

```
fn new_pixel_buffer(rows: usize, cols: usize) -> Vec<u8> {
    vec![0; rows * cols]
}
```

La macro équivaut à appeler à créer un nouveau vecteur vide, puis à y pousser les éléments, ce qui est un autre idiome : vec! Vec::new

```
let mut pal = Vec::new();
pal.push("step");
pal.push("on");
pal.push("no");
pal.push("pets");
assert_eq!(pal, vec!["step", "on", "no", "pets"]);
```

Une autre possibilité est de construire un vecteur à partir des valeurs produites par un itérateur :

```
let v: Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

Vous devrez souvent fournir le type lors de l'utilisation (comme nous l'avons fait ici), car il peut créer de nombreux types de collections, pas seulement des vecteurs. En spécifiant le type de , nous avons rendu sans ambiguïté le type de collection que nous voulons. collect v

Comme pour les tableaux, vous pouvez utiliser des méthodes de tranche sur des vecteurs :

```
// A palindrome!
let mut palindrome = vec!["a man", "a plan", "a canal", "panama"];
palindrome.reverse();
// Reasonable yet disappointing:
assert_eq!(palindrome, vec!["panama", "a canal", "a plan", "a man"]);
```

Ici, la méthode est en fait définie sur des tranches, mais l'appel emprunte implicitement une tranche au vecteur et l'appelle. reverse &mut [&str] reverse

vec est un type essentiel à Rust - il est utilisé presque partout où l'on a besoin d'une liste de taille dynamique - il existe donc de nombreuses autres méthodes qui construisent de nouveaux vecteurs ou étendent ceux existants. Nous les couvrirons au [chapitre 16](#).

A se compose de trois valeurs : un pointeur vers le tampon alloué au tas pour les éléments, qui est créé et détenu par le ; le nombre d'éléments que la mémoire tampon a la capacité de stocker; et le nombre qu'il contient réellement maintenant (en d'autres termes, sa longueur). Lorsque le tampon a atteint sa capacité, l'ajout d'un autre élément au vecteur implique l'allocation d'un tampon plus grand, la copie du contenu actuel, la mise à jour du pointeur du vecteur et la capacité de décrire le nouveau tampon, et enfin libérer l'ancien. `Vec<T>`

Si vous connaissez le nombre d'éléments dont un vecteur aura besoin à l'avance, vous pouvez appeler pour créer un vecteur avec un tampon suffisamment grand pour les contenir tous, dès le début; ensuite, vous pouvez ajouter les éléments au vecteur un à la fois sans provoquer de réaffectation. La macro utilise une astuce comme celle-ci, car elle sait combien d'éléments le vecteur final aura. Notez que cela n'établit que la taille initiale du vecteur ; si vous dépassez votre estimation, le vecteur

agrandit simplement son stockage comme d'habitude. `Vec::new Vec::with_capacity vec!`

De nombreuses fonctions de bibliothèque recherchent la possibilité d'utiliser à la place de `.` Par exemple, dans l'exemple, l'itérateur sait à l'avance qu'il donnera cinq valeurs, et la fonction en profite pour pré-alouer le vecteur qu'il renvoie avec la capacité correcte. Nous verrons comment cela fonctionne au [chapitre](#)

[15.](#) `Vec::with_capacity Vec::new collect 0..5 collect`

Tout comme la méthode d'un vecteur renvoie le nombre d'éléments qu'elle contient maintenant, sa méthode renvoie le nombre d'éléments qu'elle pourrait contenir sans réaffectation : `len capacity`

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
// Typically prints "capacity is now 4":
println!("capacity is now {}", v.capacity());
```

La capacité imprimée à la fin n'est pas garantie d'être exactement 4, mais elle sera d'au moins 3, puisque le vecteur contient trois valeurs.

Vous pouvez insérer et supprimer des éléments où vous le souhaitez dans un vecteur, bien que ces opérations déplacent tous les éléments après la position affectée vers l'avant ou vers l'arrière, de sorte qu'ils peuvent être lents si le vecteur est long :

```
let mut v = vec![10, 20, 30, 40, 50];

// Make the element at index 3 be 35.
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// Remove the element at index 1.
```

```
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

Vous pouvez utiliser la méthode pour supprimer le dernier élément et le renvoyer. Plus précisément, faire apparaître une valeur à partir de `a` renvoie `None` si le vecteur était déjà vide, ou si son dernier élément avait été :
`pop` `Vec<T>` `Option<T>` `None` `Some(v)` `v`

```
let mut v = vec!["Snow Puff", "Glass Gem"];
assert_eq!(v.pop(), Some("Glass Gem"));
assert_eq!(v.pop(), Some("Snow Puff"));
assert_eq!(v.pop(), None);
```

Vous pouvez utiliser une boucle pour itérer sur un vecteur : `for`

```
// Get our command-line arguments as a vector of Strings.
let languages: Vec<String> = std::env::args().skip(1).collect();
for l in languages {
    println!("{}: {}", l,
        if l.len() % 2 == 0 {
            "functional"
        } else {
            "imperative"
    });
}
```

L'exécution de ce programme avec une liste de langages de programmation est éclairante:

```
$ cargo run Lisp Scheme C C++ Fortran
Compiling proglangs v0.1.0 (/home/jimb/rust/proglangs)
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
        Running `target/debug/proglangs Lisp Scheme C C++ Fortran`
Lisp: functional
Scheme: functional
C: imperative
C++: imperative
Fortran: imperative
$
```

Enfin, une définition satisfaisante du terme *langage fonctionnel*.

Malgré son rôle fondamental, est un type ordinaire défini dans Rust, non intégré dans le langage. Nous couvrirons les techniques nécessaires à la

mise en œuvre de tels types au [chapitre 22](#). Vec

Tranches

Une tranche, écrite sans spécifier la longueur, est une région d'un tableau ou d'un vecteur. Étant donné qu'une tranche peut être de n'importe quelle longueur, les tranches ne peuvent pas être stockées directement dans des variables ou passées en tant qu'arguments de fonction. Les tranches sont toujours transmises par référence. [T]

Une référence à une tranche est un *pointeur gras* : une valeur de deux mots comprenant un pointeur vers le premier élément de la tranche et le nombre d'éléments dans la tranche.

Supposons que vous exécutez le code suivant :

```
let v: Vec<f64> = vec![0.0, 0.707, 1.0, 0.707];
let a: [f64; 4] = [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa: &[f64] = &a;
```

Dans les deux dernières lignes, Rust convertit automatiquement la référence et la référence en références de tranche qui pointent directement vers les données. `&Vec<f64> &[f64; 4]`

À la fin, la mémoire ressemble à [la Figure 3-2](#).

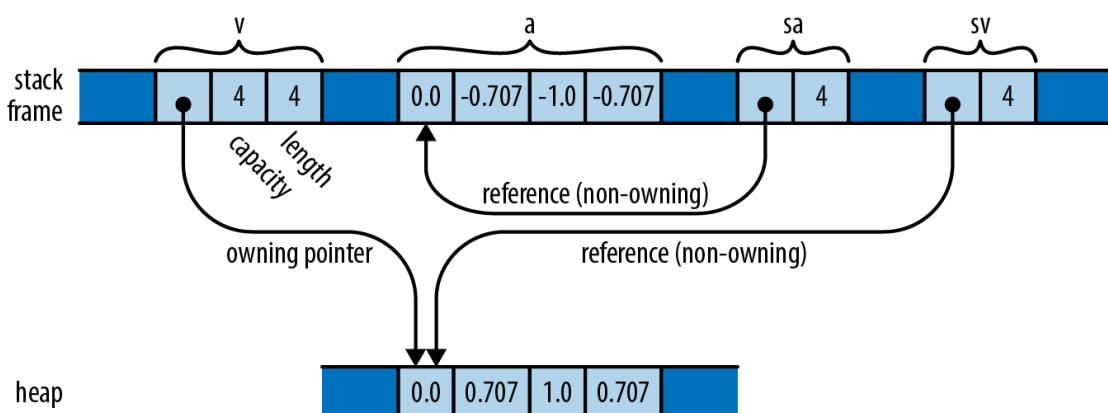


Figure 3-2. Un vecteur et un tableau en mémoire, avec des tranches et se référant à chacun `v` `a` `sa` `sv`

Alors qu'une référence ordinaire est un pointeur non propriétaire vers une seule valeur, une référence à une tranche est un pointeur non propriétaire vers une plage de valeurs consécutives en mémoire. Cela fait des références de tranche un bon choix lorsque vous souhaitez écrire une

fonction qui fonctionne sur un tableau ou un vecteur. Par exemple, voici une fonction qui imprime une tranche de nombres, un par ligne :

```
fn print(n: &[f64]) {
    for elt in n {
        println!("{}", elt);
    }
}

print(&a); // works on arrays
print(&v); // works on vectors
```

Étant donné que cette fonction prend une référence de tranche comme argument, vous pouvez l'appliquer à un vecteur ou à un tableau, comme illustré. En fait, de nombreuses méthodes que vous pourriez considérer comme appartenant à des vecteurs ou à des tableaux sont des méthodes définies sur des tranches : par exemple, les méthodes `et`, qui trient ou inversent une séquence d'éléments en place, sont en fait des méthodes sur le type de tranche `.sort reverse [T]`

Vous pouvez obtenir une référence à une tranche d'un tableau ou d'un vecteur, ou à une tranche d'une tranche existante, en l'indexant avec une plage :

```
print(&v[0..2]); // print the first two elements of v
print(&a[2..]); // print elements of a starting with a[2]
print(&sv[1..3]); // print v[1] and v[2]
```

Comme pour les accès aux tableaux ordinaires, Rust vérifie que les index sont valides. Essayer d'emprunter une tranche qui s'étend au-delà de la fin des données entraîne une panique.

Étant donné que les tranches apparaissent presque toujours derrière les références, nous nous référons souvent simplement à des types comme ou comme des « tranches », en utilisant le nom plus court pour le concept le plus commun. `&[T]` `&str`

Types de chaînes

Les programmeurs familiers avec C++ se souviendront qu'il existe deux types de chaînes dans le langage. Les littéraux de chaîne ont le type de pointeur `. La bibliothèque standard propose également une classe, , pour`

créer dynamiquement des chaînes au moment de l'exécution. const

```
char * std::string
```

Rust a un design similaire. Dans cette section, nous allons montrer toutes les façons d'écrire des littéraux de chaîne, puis présenter les deux types de chaînes de Rust. Nous fournissons plus de détails sur les chaînes et la gestion du texte au [chapitre 17](#).

Littéraux de chaîne

Les littéraux de chaîne sont placés entre guillemets doubles. Ils utilisent les mêmes séquences d'échappement de barre oblique inverse que les littéraux : char

```
let speech = "\"Ouch!\" said the well.\n";
```

Dans les littéraux de chaîne, contrairement aux littéraux, les guillemets simples n'ont pas besoin d'une barre oblique inverse, contrairement aux guillemets doubles. char

Une chaîne peut s'étendre sur plusieurs lignes :

```
println!("In the room the women come and go,
Singing of Mount Abora");
```

Le caractère de nouvelle ligne dans ce littéral de chaîne est inclus dans la chaîne et donc dans la sortie. Il en va de même pour les espaces au début de la deuxième ligne.

Si une ligne d'une chaîne se termine par une barre oblique inverse, le caractère de nouvelle ligne et l'espace blanc de début de la ligne suivante sont supprimés :

```
println!("It was a bright, cold day in April, and \
there were four of us--\
more or less.");
```

Cela imprime une seule ligne de texte. La chaîne contient un seul espace entre « et » et « là » car il y a un espace avant la barre oblique inverse dans le programme, et aucun espace entre le tiret em et « more ».

Dans quelques cas, la nécessité de doubler chaque barre oblique inverse d'une chaîne est une nuisance. (Les exemples classiques sont les expres-

sions régulières et les chemins d'accès Windows.) Pour ces cas, Rust propose des *cordes brutes*. Une chaîne brute est marquée avec la lettre minuscule `r`. Toutes les barres obliques inverses et les espaces blancs à l'intérieur d'une chaîne brute sont inclus textuellement dans la chaîne. Aucune séquence d'échappement n'est reconnue : `r`

```
let default_win_install_path = r"C:\Program Files\Gorillas";  
  
let pattern = Regex::new(r"\d+(\.\d+)*");
```

Vous ne pouvez pas inclure un caractère à guillemets doubles dans une chaîne brute simplement en plaçant une barre oblique inverse devant elle – rappelez-vous, nous avons dit qu'aucune séquence d'échappement n'est reconnue. Cependant, il existe également un remède pour cela. Le début et la fin d'une chaîne brute peuvent être marqués par des signes dièse :

```
println!(r###"  
    This raw string started with 'r###'.  
    Therefore it does not end until we reach a quote mark ('"')  
    followed immediately by three pound signs ('###'):###);
```

Vous pouvez ajouter aussi peu ou autant de signes dièse que nécessaire pour indiquer clairement où se termine la chaîne brute.

Chaînes d'octets

Un littéral de chaîne avec le préfixe est une *chaîne d'octets*. Une telle chaîne est une tranche de valeurs, c'est-à-dire des octets, plutôt que du texte Unicode : `b u8`

```
let method = b"GET";  
assert_eq!(method, &[b'G', b'E', b'T']);
```

Le type de est : c'est une référence à un tableau de trois octets. Il n'a aucune des méthodes de chaîne dont nous discuterons dans une minute. La chose la plus semblable à une chaîne à ce sujet est la syntaxe que nous avons utilisée pour l'écrire. `method & [u8; 3]`

Les chaînes d'octets peuvent utiliser toutes les autres syntaxes de chaîne que nous avons montrées : elles peuvent s'étendre sur plusieurs lignes,

utiliser des séquences d'échappement et utiliser des barres obliques inverses pour joindre des lignes. Les chaînes d'octets brutes commencent par `.br`"

Les chaînes d'octets ne peuvent pas contenir de caractères Unicode arbitraires. Ils doivent se contenter de séquences ASCII et d'échappement. `\xHH`

Chaînes en mémoire

Les chaînes Rust sont des séquences de caractères Unicode, mais elles ne sont pas stockées en mémoire sous forme de tableaux de `s`. Au lieu de cela, ils sont stockés en UTF-8, un codage à largeur variable. Chaque caractère ASCII d'une chaîne est stocké dans un octet. Les autres caractères occupent plusieurs octets. `char`

La figure 3-3 montre les valeurs créées par le code suivant : `String &str`

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ø_ø";
```

A a un tampon redimensionnable contenant du texte UTF-8. La mémoire tampon est allouée sur le tas, de sorte qu'il peut redimensionner sa mémoire tampon selon les besoins ou la demande. Dans l'exemple, est un qui possède un tampon de huit octets, dont sept sont utilisés. Vous pouvez considérer un comme un qui est garanti de contenir UTF-8 bien formé; en fait, c'est ainsi que l'on met en œuvre.

```
String noodles String String Vec<u8> String
```

A (prononcé « stir » ou « string slice ») est une référence à une série de texte UTF-8 appartenant à quelqu'un d'autre: il « emprunte » le texte. Dans l'exemple, est une référence aux six derniers octets du texte appartenant à , il représente donc le texte « oodles ». Comme d'autres références de tranche, a est un pointeur gras, contenant à la fois l'adresse des données réelles et leur longueur. Vous pouvez penser à un comme n'étant rien de plus qu'un qui est garanti de contenir UTF-8 bien formé.

```
&str oodles &str noodles &str &str &[u8]
```

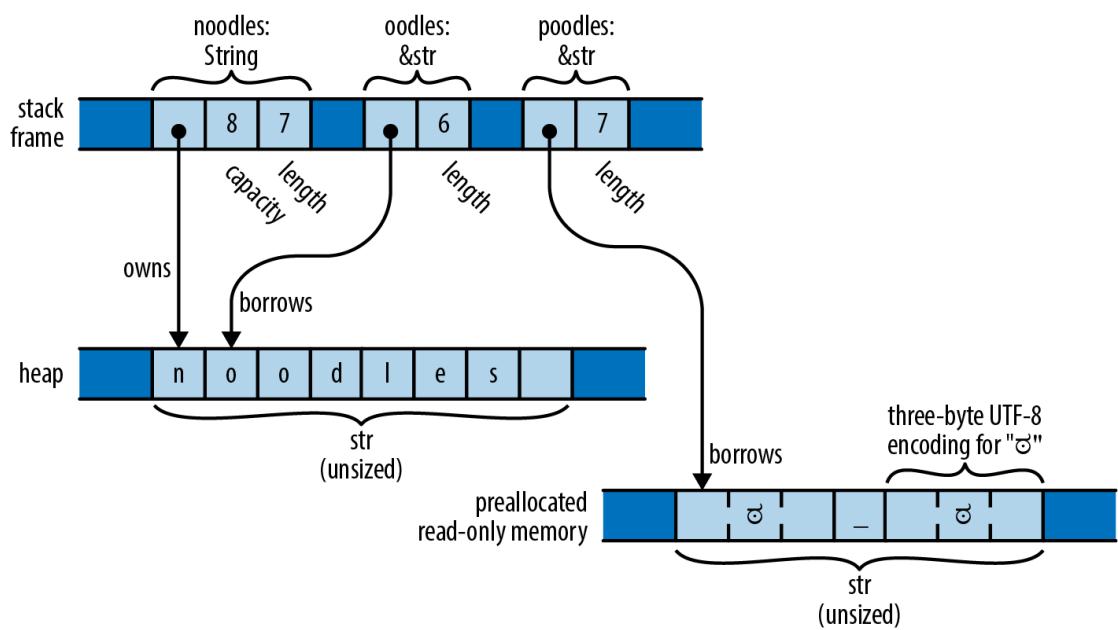


Figure 3-3., et String &str str

Un littéral de chaîne est un qui fait référence au texte préalloué, généralement stocké dans la mémoire en lecture seule avec le code machine du programme. Dans l'exemple précédent, est un littéral de chaîne, pointant vers sept octets qui sont créés lorsque le programme commence l'exécution et qui durent jusqu'à ce qu'il se ferme. `&str poodles`

La méthode de A ou renvoie sa longueur. La longueur est mesurée en octets, et non en caractères : `String &str .len()`

```
assert_eq!("o_o_".len(), 7);
assert_eq!("o_o_".chars().count(), 3);
```

Il est impossible de modifier un : `&str`

```
let mut s = "hello";
s[0] = 'c';      // error: `&str` cannot be modified, and other reasons
s.push('\n');   // error: no method named `push` found for reference `&s`
```

Pour créer de nouvelles chaînes au moment de l'exécution, utilisez

`.String`

Le type existe, mais il n'est pas très utile, car presque toutes les opérations sur UTF-8 peuvent modifier sa longueur totale d'octets et une tranche ne peut pas réaffecter son référent. En fait, les seules opérations disponibles sur sont et , qui modifient le texte en place et n'affectent que les caractères codés sur un octet, par définition. `&mut str &mut str make_ascii_uppercase make_ascii_lowercase`

Corde

`&str` ressemble beaucoup à : un gros pointeur vers certaines données.
est analogue à , tel que décrit dans [le tableau 3-11](#). & [T] String Vec<T>

Tableau 3-11. et comparaison vec <T> string

	Vec<T>	Corde
Libère automatiquement les tampons	Oui	Oui
Cultivable	Oui	Oui
<code>::new()</code> et fonctions associées au type <code>::with_capacity()</code>	Oui	Oui
<code>.reserve()</code> et méthodes <code>.capacity</code> ()	Oui	Oui
<code>.push()</code> et méthodes <code>.pop()</code>	Oui	Oui
Syntaxe de plage <code>v[start..stop]</code>	Oui, retours & [T]	Oui, retours &s tr
Conversion automatique	<code>&Vec<T></code> À <code>&[T]</code>	<code>&String</code> À <code>&str</code>
Hérite des méthodes	De <code>&[T]</code>	De <code>&str</code>

Comme un , chacun a son propre tampon alloué au tas qui n'est partagé avec aucun autre . Lorsqu'une variable sort de la portée, la mémoire tampon est automatiquement libérée, sauf si elle a été déplacée. Vec String String String String

Il existe plusieurs façons de créer s : string

- La méthode convertit a en . Cela copie la chaîne
`: .to_string() &str String`

```
let error_message = "too many pets".to_string();
```

La méthode fait la même chose, et vous pouvez la voir utilisée de la même manière. Cela fonctionne également pour d'autres types, comme

nous en discuterons au [chapitre 13](#). `.to_owned()`

- La macro fonctionne comme , sauf qu'elle renvoie un nouveau texte au lieu d'écrire du texte dans stdout, et elle n'ajoute pas automatiquement une nouvelle ligne à la fin : `format!() println!() String`

```
assert_eq!(format!("{}{:02}{:02}"N, 24, 5, 23),  
          "2405'23"N".to_string());
```

- Les tableaux, les tranches et les vecteurs de chaînes ont deux méthodes, `: .concat() .join(sep) String`, qui forment une nouvelle à partir de nombreuses chaînes

```
let bits = vec![ "veni", "vidi", "vici"];  
assert_eq!(bits.concat(), "venividivici");  
assert_eq!(bits.join(", "), "veni, vidi, vici");
```

Le choix se pose parfois du type à utiliser : ou . [Le chapitre 5](#) aborde cette question en détail. Pour l'instant, il suffira de souligner que a peut faire référence à n'importe quelle tranche de n'importe quelle chaîne, qu'il s'agisse d'un littéral de chaîne (stocké dans l'exécutable) ou d'un (alloué et libéré au moment de l'exécution). Cela signifie que cela est plus approprié pour les arguments de fonction lorsque l'appelant doit être autorisé à passer l'un ou l'autre type de chaîne. `&str String &str String &str`

Utilisation de chaînes

Les chaînes prennent en charge les opérateurs et. Deux chaînes sont égales si elles contiennent les mêmes caractères dans le même ordre (qu'elles pointent ou non vers le même emplacement en mémoire)

`: == !=`

```
assert!("ONE".to_lowercase() == "one");
```

Les chaînes prennent également en charge les opérateurs de comparaison , , et , ainsi que de nombreuses méthodes et fonctions utiles que vous pouvez trouver dans la documentation en ligne sous « (type primitif) » ou le module « » (ou simplement basculer vers le [chapitre 17](#)). En voici quelques exemples : `< <= > >= str std::str`

```
assert!("peanut".contains("nut"));  
assert_eq!("σ_σ".replace("σ", "█"), "█_█");  
assert_eq!("    clean\n".trim(), "clean");
```

```
for word in "veni, vidi, vici".split(", ") {  
    assert!(word.starts_with("v"));  
}
```

Gardez à l'esprit que, compte tenu de la nature d'Unicode, une simple comparaison ne donne *pas* toujours les réponses attendues. Par exemple, les chaînes Rust et sont toutes deux des représentations Unicode valides pour thé, le mot Français pour thé. Unicode dit qu'ils devraient tous deux être affichés et traités de la même manière, mais Rust les traite comme deux chaînes complètement distinctes. De même, les opérateurs d'ordre de Rust utilisent un ordre lexicographique simple basé sur des valeurs de point de code de caractère. Cet ordre ne ressemble que parfois à l'ordre utilisé pour le texte dans la langue et la culture de l'utilisateur. Nous abordons ces questions plus en détail au [chapitre 17](#).

```
17. char char "th\u{e9}" "the\u{301}" <
```

Autres types de chaînes

Rust garantit que les chaînes sont valides UTF-8. Parfois, un programme a vraiment besoin de pouvoir traiter des chaînes qui *ne sont pas* des Unicode valides. Cela se produit généralement lorsqu'un programme Rust doit interagir avec un autre système qui n'applique pas de telles règles. Par exemple, dans la plupart des systèmes d'exploitation, il est facile de créer un fichier avec un nom de fichier qui n'est pas valide Unicode. Que devrait-il se passer lorsqu'un programme Rust rencontre ce type de nom de fichier?

La solution de Rust est d'offrir quelques types de chaînes pour ces situations:

- S'en tenir à et pour le texte Unicode. `String &str`
- Lorsque vous travaillez avec des noms de fichiers, utilisez et à la place. `std::path::PathBuf &Path`
- Lorsque vous travaillez avec des données binaires qui ne sont pas du tout codées en UTF-8, utilisez et `.Vec<u8> &[u8]`
- Lorsque vous travaillez avec des noms de variables d'environnement et des arguments de ligne de commande sous la forme native présentée par le système d'exploitation, utilisez et `.OsString &OsStr`
- Lors de l'interopérabilité avec des bibliothèques C qui utilisent des chaînes terminées par une valeur NULL, utilisez et `.std::ffi::CString &CStr`

Tapez des alias

Le mot-clé peut être utilisé comme en C++ pour déclarer un nouveau nom pour un type existant : `type typedef`

```
type Bytes = Vec<u8>;
```

Le type que nous déclarons ici est un raccourci pour ce type particulier de `:Bytes Vec`

```
fn decode(data: &Bytes) {  
    ...  
}
```

Au-delà des bases

Les types sont une partie centrale de Rust. Nous continuerons à parler des types et à en introduire de nouveaux tout au long du livre. En particulier, les types définis par l'utilisateur de Rust donnent au langage une grande partie de sa saveur, car c'est là que les méthodes sont définies. Il existe trois types de types définis par l'utilisateur, et nous les couvrirons dans trois chapitres successifs : les structs du [chapitre 9](#), les enums du [chapitre 10](#) et les traits du [chapitre 11](#).

Les fonctions et les fermetures ont leurs propres types, couverts au [chapitre 14](#). Et les types qui composent la bibliothèque standard sont couverts tout au long du livre. Par exemple, [le chapitre 16](#) présente les types de collection standard.

Tout cela devra attendre, cependant. Avant de passer à autre chose, il est temps de s'attaquer aux concepts qui sont au cœur des règles de sécurité de Rust.

Chapitre 4. Propriété et déménagements

En ce qui concerne la gestion de la mémoire, il y a deux caractéristiques que nous aimerais de nos langages de programmation :

- Nous aimerais que la mémoire soit libérée rapidement, au moment de notre choix. Cela nous donne le contrôle sur la consommation de mémoire du programme.
- Nous ne voulons jamais utiliser un pointeur vers un objet après sa libération. Il s'agirait d'un comportement indéfini, conduisant à des accidents et à des failles de sécurité.

Mais ceux-ci semblent s'exclure mutuellement : libérer une valeur alors que des pointeurs existent vers elle laisse nécessairement ces pointeurs pendants. Presque tous les principaux langages de programmation tombent dans l'un des deux camps, selon laquelle des deux qualités ils abandonnent:

- Le camp « La sécurité d'abord » utilise le ramassage des ordures pour gérer la mémoire, libérant automatiquement les objets lorsque tous les pointeurs accessibles vers eux ont disparu. Cela élimine les pointeurs pendants en gardant simplement les objets autour jusqu'à ce qu'il n'y ait plus de pointeurs vers eux à pendre. Presque tous les langages modernes tombent dans ce camp, de Python, JavaScript et Ruby à Java, C # et Haskell.

Mais s'appuyer sur le ramassage des ordures signifie abandonner le contrôle sur le moment exact où les objets sont libérés au collecteur. En général, les éboueurs sont des bêtes surprenantes, et comprendre pourquoi la mémoire n'a pas été libérée alors que vous vous y attendiez peut être un défi.

- Le camp « Control First » vous laisse en charge de libérer la mémoire. La consommation de mémoire de votre programme est entièrement entre vos mains, mais éviter les pointeurs pendants devient également entièrement votre préoccupation. C et C++ sont les seuls langages courants dans ce camp.

C'est génial si vous ne faites jamais d'erreurs, mais les preuves suggèrent que vous finirez par le faire. L'utilisation abusive du pointeur est un coupable courant des problèmes de sécurité signalés depuis que ces données ont été collectées.

Rust vise à être à la fois sûr et performant, donc aucun de ces compromis n'est acceptable. Mais si la réconciliation était facile, quelqu'un l'aurait fait bien avant maintenant. Quelque chose de fondamental doit changer.

Rust brise l'impasse d'une manière surprenante: en limitant la façon dont vos programmes peuvent utiliser les pointeurs. Ce chapitre et le suivant sont consacrés à expliquer exactement ce que sont ces restrictions et pourquoi elles fonctionnent. Pour l'instant, il suffit de dire que certaines structures courantes que vous avez l'habitude d'utiliser peuvent ne pas correspondre aux règles, et vous devrez chercher des alternatives. Mais l'effet net de ces restrictions est d'apporter juste assez d'ordre au chaos pour permettre aux vérifications au moment de la compilation de Rust de vérifier que votre programme est exempt d'erreurs de sécurité de la mémoire: pointeurs pendants, doubles libres, utilisation de mémoire non initialisée, etc. Au moment de l'exécution, vos pointeurs sont de simples adresses en mémoire, tout comme ils le seraient en C et C++. La différence est que votre code a été prouvé pour les utiliser en toute sécurité.

Ces mêmes règles constituent également la base de la prise en charge par Rust pour une programmation simultanée sécurisée. En utilisant les primitives de threading soigneusement conçues par Rust, les règles qui garantissent que votre code utilise correctement la mémoire servent également à prouver qu'il est exempt de courses de données. Un bogue dans un programme Rust ne peut pas amener un thread à corrompre les données d'un autre, introduisant des échecs difficiles à reproduire dans des parties non liées du système. Le comportement non déterministe inhérent au code multithread est isolé aux fonctionnalités conçues pour le gérer (mutex, canaux de message, valeurs atomiques, etc.) plutôt que d'apparaître dans des références de mémoire ordinaires. Le code multi-thread en C et C ++ a gagné sa réputation laide, mais Rust le réhabilite assez bien.

Le pari radical de Rust, la revendication sur laquelle il mise son succès et qui constitue la racine du langage, est que même avec ces restrictions en place, vous trouverez le langage plus que suffisamment flexible pour presque toutes les tâches et que les avantages – l'élimination de larges classes de bogues de gestion de la mémoire et de concurrence – justifient les adaptations que vous devrez apporter à votre style. Les auteurs de ce livre sont optimistes sur Rust précisément en raison de notre vaste expérience avec C et C ++. Pour nous, l'accord de Rust est une évidence.

Les règles de Rust sont probablement différentes de ce que vous avez vu dans d'autres langages de programmation. Apprendre à travailler avec

eux et à les tourner à votre avantage est, à notre avis, le défi central de l'apprentissage de Rust. Dans ce chapitre, nous allons d'abord donner un aperçu de la logique et de l'intention derrière les règles de Rust en montrant comment les mêmes problèmes sous-jacents se déroulent dans d'autres langues. Ensuite, nous expliquerons en détail les règles de Rust, en examinant ce que signifie la propriété au niveau conceptuel et mécanique, comment les changements de propriété sont suivis dans divers scénarios et les types qui plient ou enfreignent certaines de ces règles afin d'offrir plus de flexibilité.

Propriété

Si vous avez lu beaucoup de code C ou C++, vous avez probablement rencontré un commentaire disant qu'une instance d'une classe *possède* un autre objet vers lequel elle pointe. Cela signifie généralement que l'objet propriétaire décide quand libérer l'objet possédé: lorsque le propriétaire est détruit, il détruit ses biens avec lui.

Par exemple, supposons que vous écriviez le code C++ suivant :

```
std::string s = "frayed knot";
```

La chaîne est généralement représentée en mémoire, comme illustré à [la figure 4-1](#). `s`

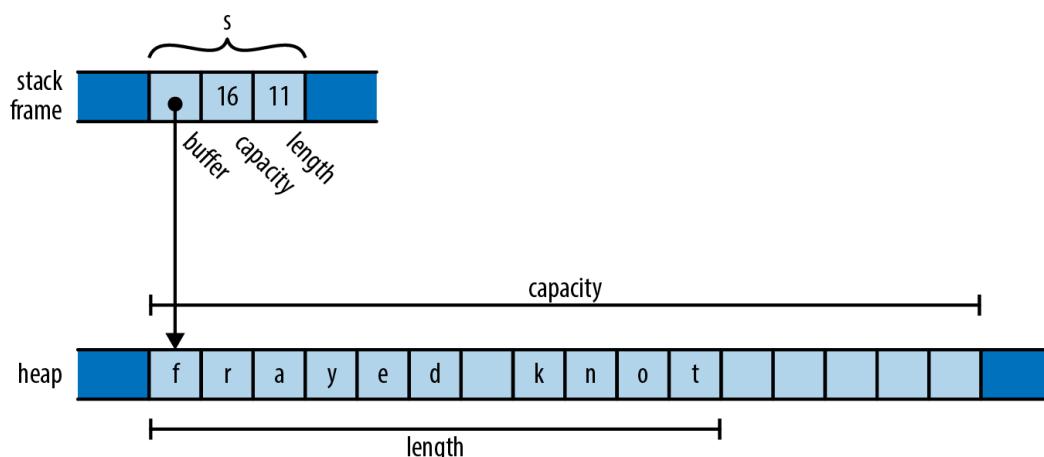


Figure 4-1. Valeur C++ sur la pile, pointant vers sa mémoire tampon allouée au tas `std::string`

Ici, l'objet lui-même est toujours exactement de trois mots, comprenant un pointeur vers un tampon alloué au tas, la capacité globale du tampon (c'est-à-dire la taille du texte peut croître avant que la chaîne ne doive allouer un tampon plus grand pour le contenir) et la longueur du texte qu'il contient maintenant. Il s'agit de champs privés à la classe, non accessibles aux utilisateurs de la chaîne. `std::string`

A possède son tampon : lorsque le programme détruit la chaîne, le destructeur de la chaîne libère le tampon. Dans le passé, certaines bibliothèques C++ partageaient un seul tampon entre plusieurs valeurs, en utilisant un nombre de références pour décider quand le tampon devait être libéré. Les versions plus récentes de la spécification C++ excluent effectivement cette représentation ; toutes les bibliothèques C++ modernes utilisent l'approche présentée ici. `std::string std::string`

Dans ces situations, il est généralement entendu que, bien qu'il soit acceptable pour un autre code de créer des pointeurs temporaires vers la mémoire possédée, il incombe à ce code de s'assurer que ses pointeurs ont disparu avant que le propriétaire ne décide de détruire l'objet possédé. Vous pouvez créer un pointeur vers un caractère vivant dans la mémoire tampon d'un tampon, mais lorsque la chaîne est détruite, votre pointeur devient invalide et c'est à vous de vous assurer que vous ne l'utilisez plus. Le propriétaire détermine la durée de vie du propriétaire, et tous les autres doivent respecter ses décisions. `std::string`

Nous avons utilisé ici comme exemple de ce à quoi ressemble la propriété en C++ : c'est juste une convention que la bibliothèque standard suit généralement, et bien que le langage vous encourage à suivre des pratiques similaires, la façon dont vous concevez vos propres types dépend en fin de compte de vous. `std::string`

Dans Rust, cependant, le concept de propriété est intégré dans le langage lui-même et appliqué par des contrôles au moment de la compilation. Chaque valeur a un seul propriétaire qui détermine sa durée de vie. Lorsque le propriétaire est libéré – *abandonné*, dans la terminologie Rust – la valeur possédée est également abandonnée. Ces règles sont destinées à vous permettre de trouver facilement la durée de vie d'une valeur donnée simplement en inspectant le code, ce qui vous donne le contrôle sur sa durée de vie qu'un langage système devrait fournir.

Une variable possède sa valeur. Lorsque le contrôle quitte le bloc dans lequel la variable est déclarée, la variable est supprimée, de sorte que sa valeur est supprimée avec elle. Par exemple :

```
fn print_padovan() {
    let mut padovan = vec![1, 1, 1]; // allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
} // dropped here
```

Le type de la variable est , un vecteur d'entiers 32 bits. En mémoire, la valeur finale de ressemblera à [la figure 4-2](#). padovan Vec<i32> padovan

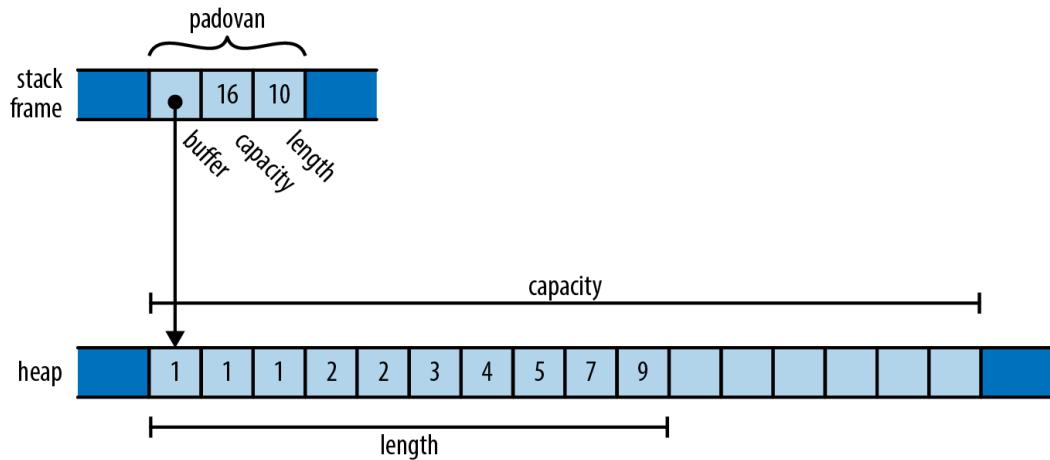


Figure 4-2. A sur la pile, pointant vers son tampon dans le tas Vec<i32>

Ceci est très similaire au C ++ que nous avons montré précédemment, sauf que les éléments dans la mémoire tampon sont des valeurs 32 bits, pas des caractères. Notez que les mots qui maintiennent le pointeur, la capacité et la longueur de la fonction vivent directement dans le cadre de la pile de la fonction; seul le tampon du vecteur est alloué sur le tas. std::string padovan print_padovan

Comme pour la chaîne précédente, le vecteur possède le tampon contenant ses éléments. Lorsque la variable sort de la portée à la fin de la fonction, le programme laisse tomber le vecteur. Et puisque le vecteur possède son tampon, le tampon va avec. s padovan

Le type de rouille sert d'autre exemple de propriété. A est un pointeur vers une valeur de type stockée sur le tas. L'appel alloue de l'espace de tas, y déplace la valeur et renvoie un pointage vers l'espace de tas. Puisqu'un possède l'espace vers lequel il pointe, lorsque le est lâché, il libère également l'espace. Box Box<T> T Box::new(v) v Box Box Box

Par exemple, vous pouvez allouer un tuple dans le tas comme suit :

```
{
    let point = Box::new((0.625, 0.5)); // point allocated here
    let label = format!("{}:", point); // label allocated here
    assert_eq!(label, "(0.625, 0.5)");
} // both dropped here
```

Lorsque le programme appelle , il alloue de l'espace pour un tuple de deux valeurs sur le tas, déplace son argument dans cet espace et lui renvoie un pointeur. Au moment où le contrôle atteint l'appel à , le cadre de la pile ressemble à [la figure 4-3](#). Box::new f64 (0.625, 0.5) assert_eq!

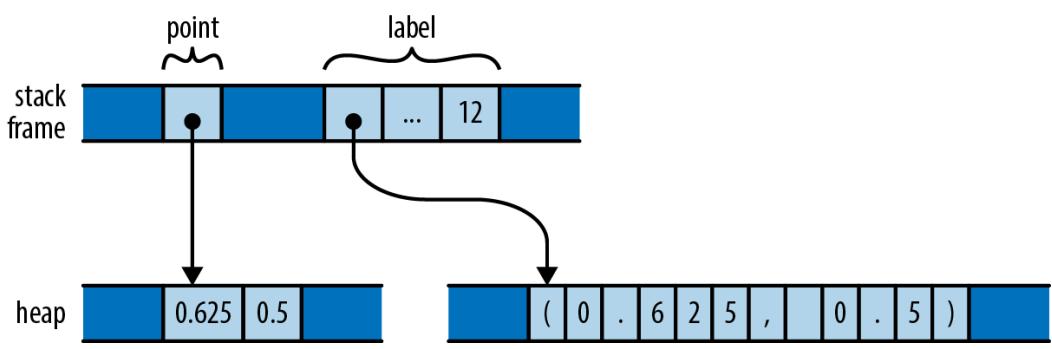


Figure 4-3. Deux variables locales, chacune possédant de la mémoire dans le tas

Le cadre de pile lui-même contient les variables et , chacune d'entre elles faisant référence à une allocation de tas qu'elle possède. Lorsqu'ils sont abandonnés, les allocations qu'ils possèdent sont libérées avec eux. `point label`

Tout comme les variables possèdent leurs valeurs, les structures possèdent leurs champs et les tuples, les tableaux et les vecteurs possèdent leurs éléments :

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{} , born {}", composer.name, composer.birth);
}
```

Ici, est un , un vecteur de structs, dont chacun contient une chaîne et un nombre. En mémoire, la valeur finale de ressemble à [la figure 4-4.](#)

`composers Vec<Person> composers`

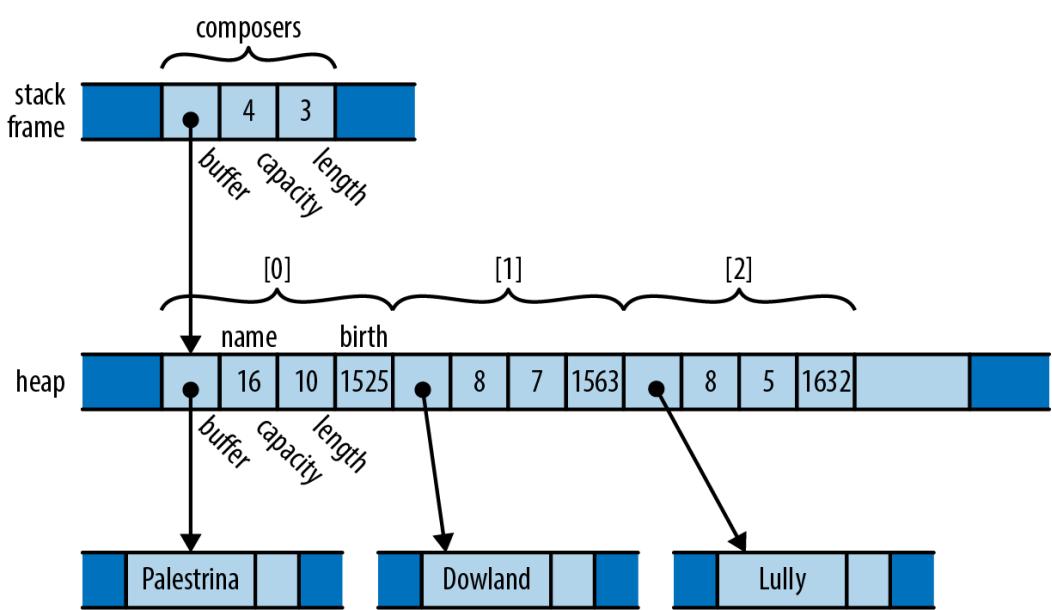


Figure 4-4. Un arbre de propriété plus complexe

Il existe de nombreuses relations de propriété ici, mais chacune est assez simple: possède un vecteur; le vecteur possède ses éléments, dont chacun est une structure; chaque structure possède ses champs; et le champ de chaîne possède son texte. Lorsque le contrôle quitte la portée dans laquelle est déclaré, le programme abandonne sa valeur et emporte l'ensemble de l'arrangement avec lui. S'il y avait d'autres types de collections dans l'image – une, peut-être, ou une – l'histoire serait la même. `composers Person composers HashMap BTreeSet`

À ce stade, prenez du recul et considérez les conséquences des relations de propriété que nous avons présentées jusqu'à présent. Chaque valeur a un seul propriétaire, ce qui facilite la décision de la laisser tomber. Mais une seule valeur peut posséder beaucoup d'autres valeurs : par exemple, le vecteur possède tous ses éléments. Et ces valeurs peuvent posséder d'autres valeurs à tour de rôle: chaque élément de possède une chaîne, qui possède son texte. `composers composers`

Il s'ensuit que les propriétaires et leurs valeurs propres forment *des arbres*: votre propriétaire est votre parent, et les valeurs que vous possédez sont vos enfants. Et à la racine ultime de chaque arbre se trouve une variable; lorsque cette variable sort de la portée, tout l'arbre l'accompagne. On peut voir un tel arbre de propriété dans le diagramme pour : ce n'est pas un « arbre » au sens d'une structure de données d'arborescence de recherche, ou un document HTML fait à partir d'éléments DOM. Nous avons plutôt un arbre construit à partir d'un mélange de types, avec la règle du propriétaire unique de Rust interdisant toute réunification de structure qui pourrait rendre l'arrangement plus complexe qu'un arbre. Chaque valeur d'un programme Rust est membre d'un arbre, enraciné dans une variable. `composers`

Les programmes Rust ne suppriment généralement pas explicitement les valeurs, de la même manière que les programmes C et C ++ utiliseraient et . La façon de supprimer une valeur dans Rust est de la supprimer de l'arbre de propriété d'une manière ou d'une autre: en quittant la portée d'une variable, ou en supprimant un élément d'un vecteur, ou quelque chose de ce genre. À ce stade, Rust s'assure que la valeur est correctement abandonnée, ainsi que tout ce qu'elle possède. `free` `delete`

Dans un certain sens, Rust est moins puissant que d'autres langages : tous les autres langages de programmation pratiques vous permettent de construire des graphiques arbitraires d'objets qui pointent les uns vers les autres de la manière que vous jugez appropriée. Mais c'est précisément parce que Rust est moins puissant que les analyses que le langage peut effectuer sur vos programmes peuvent être plus puissantes. Les garanties de sécurité de Rust sont possibles précisément parce que les relations qu'il peut rencontrer dans votre code sont plus faciles à gérer. Cela fait partie du « pari radical » de Rust que nous avons mentionné plus tôt: dans la pratique, affirme Rust, il y a généralement plus qu'assez de flexibilité dans la façon dont on résout un problème pour s'assurer qu'au moins quelques solutions parfaitement fines entrent dans les restrictions imposées par le langage.

Cela dit, le concept de propriété tel que nous l'avons expliqué jusqu'à présent est encore beaucoup trop rigide pour être utile. Rust étend cette idée simple de plusieurs façons:

- Vous pouvez déplacer des valeurs d'un propriétaire à un autre. Cela vous permet de construire, de réorganiser et de démolir l'arbre.
- Les types très simples comme les entiers, les nombres à virgule flottante et les caractères sont exemptés des règles de propriété. C'est ce qu'on appelle les types. `Copy`
- La bibliothèque standard fournit les types de pointeurs comptés par référence et , qui permettent aux valeurs d'avoir plusieurs propriétaires, sous certaines restrictions. `Rc` `Arc`
- Vous pouvez « emprunter une référence » à une valeur; les références sont des pointeurs non propriétaires, avec des durées de vie limitées.

Chacune de ces stratégies apporte de la flexibilité au modèle de propriété, tout en respectant les promesses de Rust. Nous expliquerons chacun d'eux à tour de rôle, avec des références couvertes dans le chapitre suivant.

Se déplace

Dans Rust, pour la plupart des types, les opérations telles que l'affectation d'une valeur à une variable, son passage à une fonction ou son renvoi à partir d'une fonction ne copient pas la valeur : elles la *déplacent*. La source cède la propriété de la valeur à la destination et devient non initialisée; la destination contrôle désormais la durée de vie de la valeur. Les programmes de rouille construisent et démolissent des structures complexes une valeur à la fois, un mouvement à la fois.

Vous serez peut-être surpris que Rust change le sens de ces opérations fondamentales; L'affectation est sûrement quelque chose qui devrait être assez bien cloué à ce stade de l'histoire. Cependant, si vous regardez de près comment différentes langues ont choisi de gérer les devoirs, vous verrez qu'il y a en fait une variation significative d'une école à l'autre. La comparaison rend également la signification et les conséquences du choix de Rust plus faciles à voir.

Considérez le code Python suivant :

```
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

Chaque objet Python porte un nombre de références, suivant le nombre de valeurs qui y font actuellement référence. Ainsi, après l'affectation à , l'état du programme ressemble à la [figure 4-5](#) (notez que certains champs sont omis). s

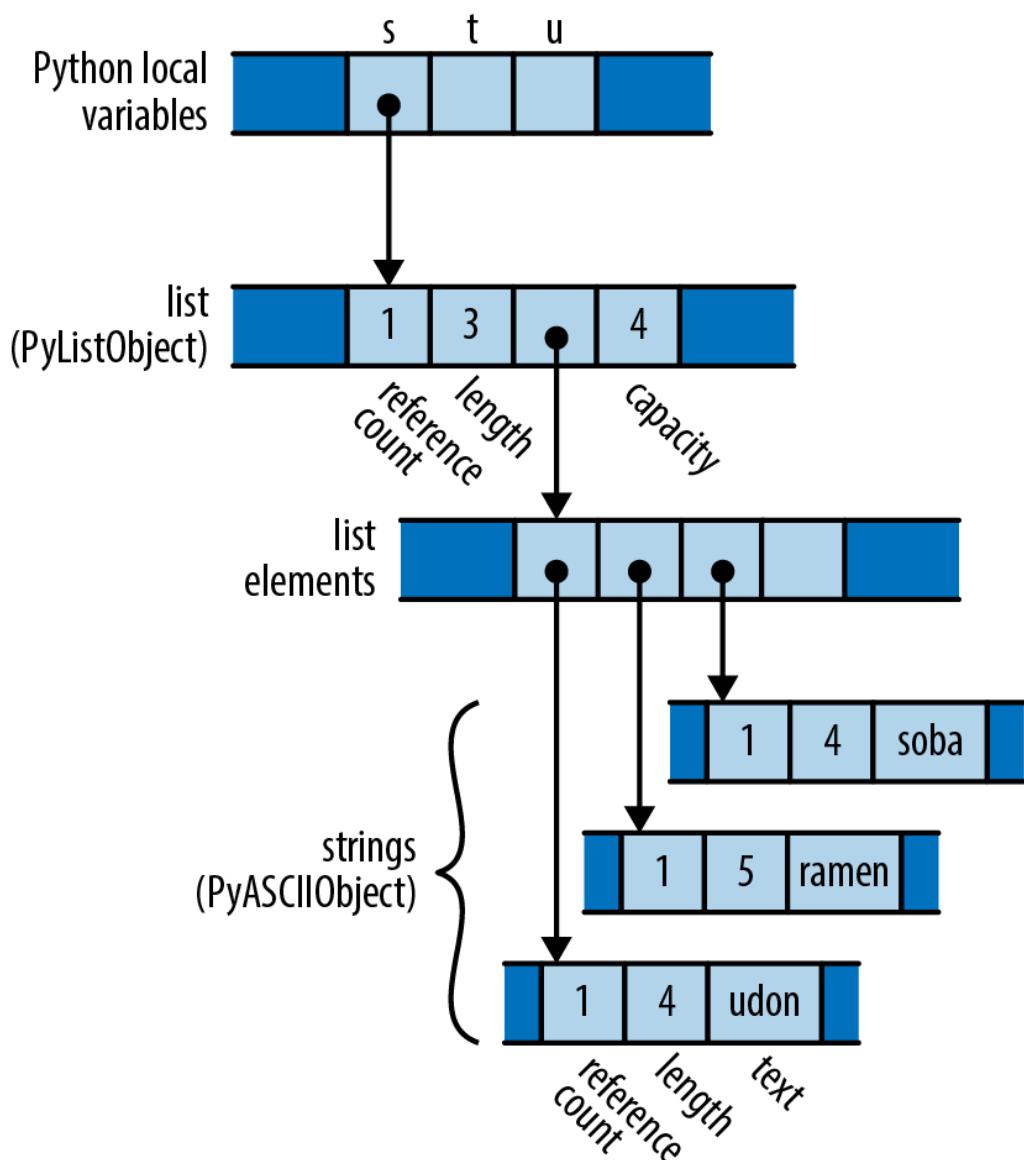


Figure 4-5. Comment Python représente une liste de chaînes en mémoire

Étant donné qu'il ne pointe que vers la liste, le nombre de références de la liste est de 1; et puisque la liste est le seul objet pointant vers les chaînes, chacun de leurs nombres de références est également égal à 1. s

Que se passe-t-il lorsque le programme exécute les affectations à `t` et `u` ?

Python implémente l'affectation simplement en faisant pointer le point de destination vers le même objet que la source et en incrémentant le nombre de références de l'objet. Donc, l'état final du programme est quelque chose comme [la figure 4-6](#). t u

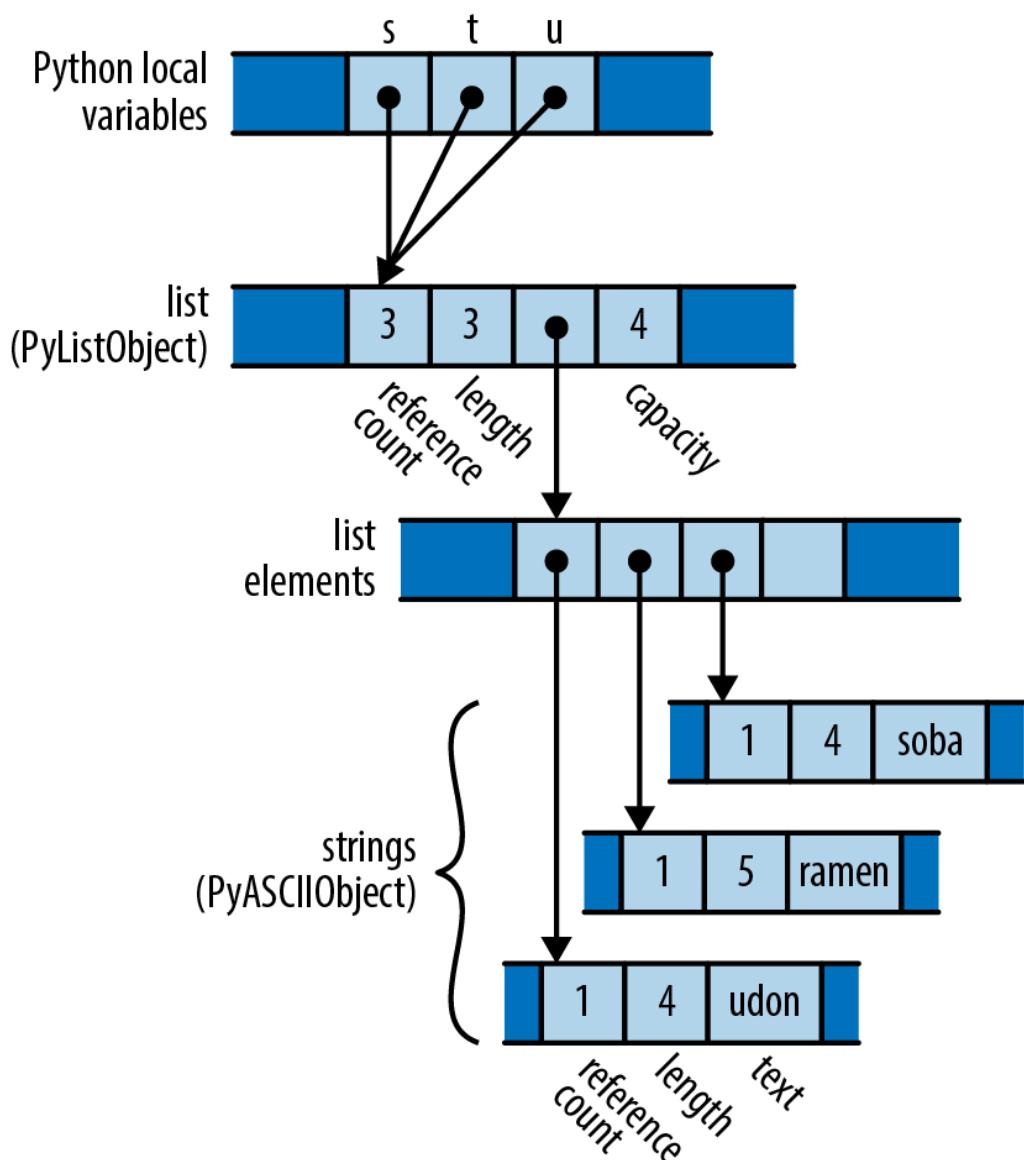


Figure 4-6. Le résultat de l'affectation aux deux et en Python *s* *t* *u*

Python a copié le pointeur de l'intérieur et mis à jour le nombre de références de la liste à 3. L'affectation en Python est bon marché, mais comme elle crée une nouvelle référence à l'objet, nous devons maintenir le nombre de références pour savoir quand nous pouvons libérer la valeur. *s* *t* *u*

Considérons maintenant le code C++ analogue :

```

using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
  
```

La valeur d'origine de ressemble à [la Figure 4-7](#) en mémoire. *s*

Que se passe-t-il lorsque le programme affecte à *t* et *u*? L'affectation d'un produit une copie du vecteur en C++ ; se comporte de la même manière. Ainsi, au moment où le programme atteint la fin de ce code, il a en fait al-

loué trois vecteurs et neuf chaînes (Figure 4-8).

8). s t u std::vector std::string

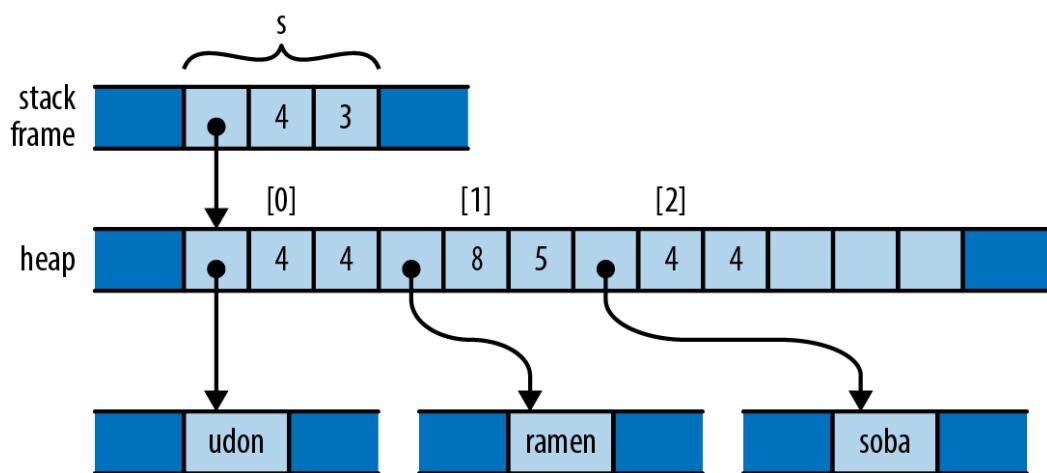


Figure 4-7. Comment C++ représente un vecteur de chaînes en mémoire

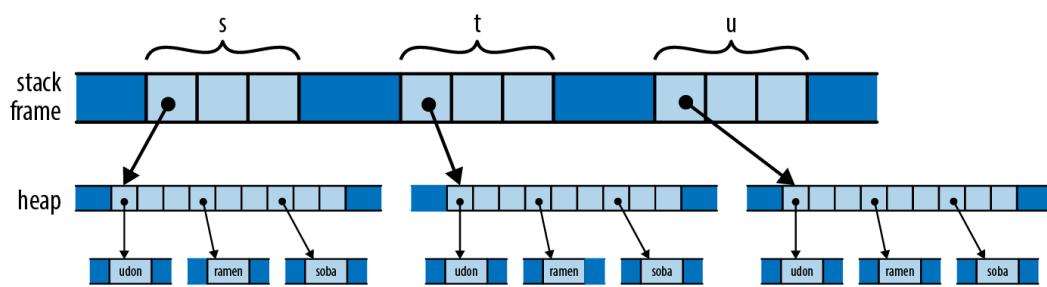


Figure 4-8. Résultat de l'affectation aux deux et en C++ s = t = u

Selon les valeurs impliquées, l'affectation en C++ peut consommer des quantités illimitées de mémoire et de temps processeur. L'avantage, cependant, est qu'il est facile pour le programme de décider quand libérer toute cette mémoire: lorsque les variables sortent du champ d'application, tout ce qui est alloué ici est nettoyé automatiquement.

Dans un sens, C++ et Python ont choisi des compromis opposés : Python rend l'affectation bon marché, au détriment du comptage des références (et dans le cas général, du garbage collection). C++ maintient la propriété de toute la mémoire claire, au détriment de l'affectation effectuer une copie profonde de l'objet. Les programmeurs C++ sont souvent moins qu'enthousiastes à propos de ce choix : les copies profondes peuvent être coûteuses, et il existe généralement des alternatives plus pratiques.

Alors, que ferait le programme analogue dans Rust? Voici le code :

```
let s = vec![ "udon".to_string(), "ramen".to_string(), "soba".to_string() ]
let t = s;
let u = s;
```

Comme C et C++, Rust met des littéraux de chaîne simples comme dans la mémoire en lecture seule, donc pour une comparaison plus claire avec les

exemples C ++ et Python, nous appelons ici pour obtenir des valeurs allouées au tas. "udon" `to_string` String

Après avoir effectué l'initialisation de , puisque Rust et C++ utilisent des représentations similaires pour les vecteurs et les chaînes, la situation ressemble à ce qu'elle était en C++ ([Figure 4-9](#)). s

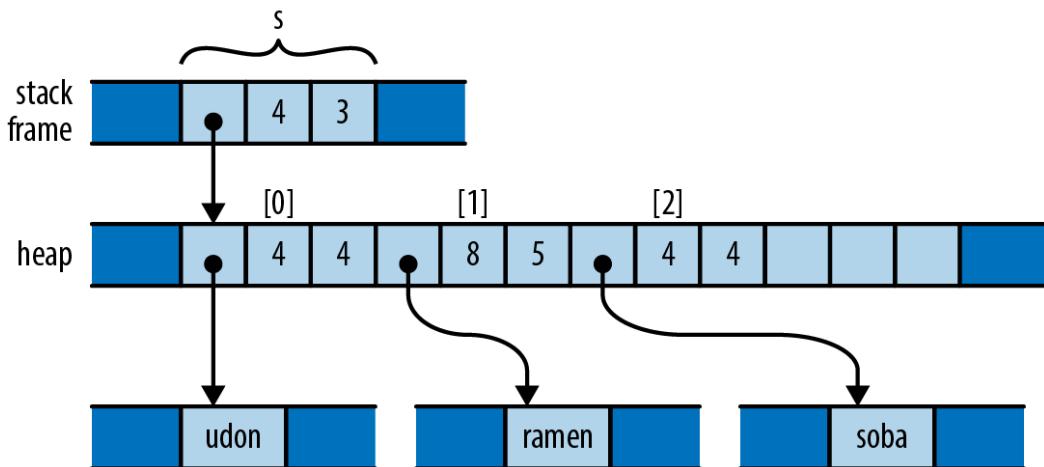


Figure 4-9. Comment Rust représente un vecteur de chaînes en mémoire

Mais rappelez-vous que, dans Rust, les affectations de la plupart des types déplacent la valeur de la source vers la destination, laissant la source non initialisée. Ainsi, après l'initialisation , la mémoire du programme ressemble à [la Figure 4-10](#). t

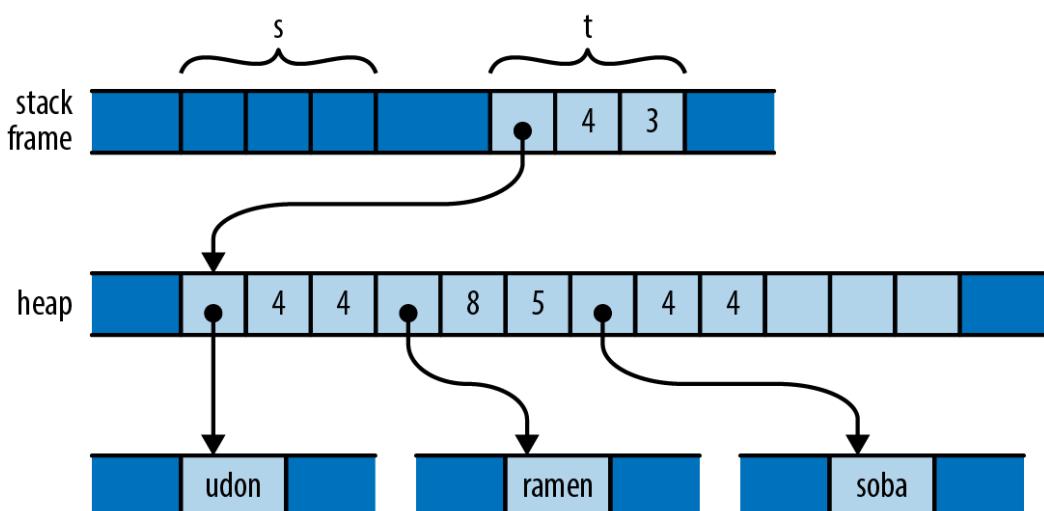


Figure 4-10. Le résultat de l'affectation à dans Rust `s = t`

Que s'est-il passé ici? L'initialisation déplaçait les trois champs d'en-tête du vecteur de à ; possède maintenant le vecteur. Les éléments du vecteur sont restés là où ils étaient, et rien n'est arrivé aux cordes non plus.

Chaque valeur a toujours un seul propriétaire, bien que l'on ait changé de mains. Il n'y avait pas de dénombrement de référence à ajuster. Et le compilateur considère maintenant comme non initialisé. `let t = s; s = t`

Alors que se passe-t-il lorsque nous atteignons l'initialisation ? Cela affecterait la valeur non initialisée à . Rust interdit prudemment l'utilisation

tion de valeurs non initialisées, de sorte que le compilateur rejette ce code avec l'erreur suivante : `let u = s; s u`

```
error: use of moved value: `s`  
|  
7 |     let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()  
|         - move occurs because `s` has type `Vec<String>`,  
|             which does not implement the `Copy` trait  
8 |     let t = s;  
|         - value moved here  
9 |     let u = s;  
|         ^ value used here after move
```

Considérez les conséquences de l'utilisation d'un déménagement par Rust ici. Comme Python, l'affectation est bon marché: le programme déplace simplement l'en-tête de trois mots du vecteur d'un endroit à un autre. Mais comme C++, la propriété est toujours claire : le programme n'a pas besoin de comptage de références ou de garbage collection pour savoir quand libérer les éléments vectoriels et le contenu des chaînes.

Le prix que vous payez est que vous devez demander explicitement des copies quand vous le souhaitez. Si vous voulez vous retrouver dans le même état que le programme C++, chaque variable contenant une copie indépendante de la structure, vous devez appeler la méthode du vecteur, qui effectue une copie profonde du vecteur et de ses éléments : `clone`

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()];  
let t = s.clone();  
let u = s.clone();
```

Vous pouvez également recréer le comportement de Python en utilisant les types de pointeurs comptés par référence de Rust ; nous en discuterons sous peu dans [« Rc et Arc: Shared Ownership »](#).

Plus d'opérations qui se déplacent

Dans les exemples jusqu'à présent, nous avons montré des initialisations, fournissant des valeurs pour les variables au fur et à mesure qu'elles entrent dans la portée d'une instruction. L'affectation à une variable est légèrement différente, en ce sens que si vous déplacez une valeur dans une variable déjà initialisée, Rust supprime la valeur antérieure de la variable. Par exemple: `let`

```
let mut s = "Govinda".to_string();  
s = "Siddhartha".to_string(); // value "Govinda" dropped here
```

Dans ce code, lorsque le programme affecte la chaîne à , sa valeur antérieure est supprimée en premier. Mais considérez ce qui suit: "Siddhartha" s "Govinda"

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // nothing is dropped here
```

Cette fois, a pris possession de la chaîne d'origine de , de sorte qu'au moment où nous l'affectons à , elle n'est pas initialisée. Dans ce scénario, aucune chaîne n'est supprimée. t s s

Nous avons utilisé des initialisations et des affectations dans les exemples ici parce qu'elles sont simples, mais Rust applique la sémantique de déplacement à presque toutes les utilisations d'une valeur. Le passage d'arguments aux fonctions déplace la propriété vers les paramètres de la fonction ; le renvoi d'une valeur d'une fonction déplace la propriété vers l'appelant. La construction d'un tuple déplace les valeurs dans le tuple. Et ainsi de suite.

Vous avez peut-être maintenant un meilleur aperçu de ce qui se passe réellement dans les exemples que nous avons proposés dans la section précédente. Par exemple, lorsque nous construisions notre vecteur de compositeurs, nous écrivions :

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

Ce code indique plusieurs endroits où les déplacements se produisent, au-delà de l'initialisation et de l'affectation :

Renvoi de valeurs à partir d'une fonction

L'appel construit un nouveau vecteur et renvoie, non pas un pointeur vers le vecteur, mais le vecteur lui-même : sa propriété se déplace de vers la variable . De même, l'appel renvoie une nouvelle instance. Vec::new() Vec::new composers to_string String

Construire de nouvelles valeurs

Le champ de la nouvelle structure est initialisé avec la valeur de retour de . La structure prend possession de la chaîne. name Person to_string

La structure entière, et non un pointeur vers elle, est transmise à la méthode du vecteur, qui la déplace à l'extrémité de la structure. Le vecteur prend possession du et devient ainsi le propriétaire indirect du nom. Person push Person String

Déplacer des valeurs comme celle-ci peut sembler inefficace, mais il y a deux choses à garder à l'esprit. Tout d'abord, les mouvements s'appliquent toujours à la valeur proprement dite, et non au stockage en tas qu'ils possèdent. Pour les vecteurs et les chaînes, la *valeur propre* est l'en-tête de trois mots seul ; les tableaux d'éléments et les tampons de texte potentiellement volumineux se trouvent à l'endroit où ils se trouvent dans le tas. Deuxièmement, la génération de code du compilateur Rust est bonne pour « voir à travers » tous ces mouvements; en pratique, le code machine stocke souvent la valeur directement là où elle appartient.

Déplacements et contrôle du flux

Les exemples précédents ont tous un flux de contrôle très simple; Comment les mouvements interagissent-ils avec un code plus compliqué ? Le principe général est que, s'il est possible qu'une variable ait vu sa valeur déplacée et qu'elle n'ait pas définitivement reçu une nouvelle valeur depuis, elle est considérée comme non initialisée. Par exemple, si une variable a encore une valeur après avoir évalué la condition d'une expression, nous pouvons l'utiliser dans les deux branches : if

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... ok to move from x here
} else {
    g(x); // ... and ok to also move from x here
}
h(x); // bad: x is uninitialized here if either path uses it
```

Pour des raisons similaires, il est interdit de passer d'une variable dans une boucle :

```
let x = vec![10, 20, 30];
while f() {
    g(x); // bad: x would be moved in first iteration,
           // uninitialized in second
}
```

C'est-à-dire, à moins que nous ne lui ayons définitivement donné une nouvelle valeur d'ici la prochaine itération:

```

let mut x = vec![10, 20, 30];
while f() {
    g(x);           // move from x
    x = h();        // give x a fresh value
}
e(x);

```

Déplacements et contenu indexé

Nous avons mentionné qu'un déménagement laisse sa source non initialisée, car la destination prend possession de la valeur. Mais tous les types de propriétaires de valeur ne sont pas prêts à devenir non initialisés. Par exemple, considérez le code suivant :

```

// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
let third = v[2]; // error: Cannot move out of index of Vec
let fifth = v[4]; // here too

```

Pour que cela fonctionne, Rust devrait en quelque sorte se rappeler que les troisième et cinquième éléments du vecteur sont devenus non initialisés et suivre cette information jusqu'à ce que le vecteur soit abandonné. Dans le cas le plus général, les vecteurs devraient emporter avec eux des informations supplémentaires pour indiquer quels éléments sont vivants et lesquels sont devenus non initialisés. Ce n'est clairement pas le bon comportement pour un langage de programmation de systèmes; un vecteur ne doit être rien d'autre qu'un vecteur. En fait, Rust rejette le code précédent avec l'erreur suivante :

```

error: cannot move out of index of `Vec<String>`
|
14 |     let third = v[2];
|          ^
|          |
|          move occurs because value has type `String`,
|          which does not implement the `Copy` trait
|          help: consider borrowing here: `&v[2]`

```

Il fait également une plainte similaire au sujet du déménagement à . Dans le message d'erreur, Rust suggère d'utiliser une référence, au cas où vous

voudriez accéder à l'élément sans le déplacer. C'est souvent ce que vous voulez. Mais que se passe-t-il si vous voulez vraiment déplacer un élément hors d'un vecteur ? Vous devez trouver une méthode qui le fait d'une manière qui respecte les limites du type. Voici trois possibilités : `fifth`

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 1. Pop a value off the end of the vector:
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");

// 2. Move a value out of a given index in the vector,
// and move the last element into its spot:
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. Swap in another value for the one we're taking out:
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// Let's see what's left of our vector.
assert_eq!(v, vec!["101", "104", "substitute"]);
```

Chacune de ces méthodes déplace un élément hors du vecteur, mais le fait d'une manière qui laisse le vecteur dans un état entièrement peuplé, même s'il est peut-être plus petit.

Les types de collection comme proposent également généralement des méthodes pour consommer tous leurs éléments dans une boucle: `vec`

```
let v = vec![
    "liberté".to_string(),
    "égalité".to_string(),
    "fraternité".to_string()];
for mut s in v {
    s.push('!');
    println!("{}!", s);
}
```

Lorsque nous passons directement le vecteur à la boucle, comme dans , cela *déplace* le vecteur hors de , laissant non initialisé. La machinerie interne de la boucle s'approprie le vecteur et le dissèque en ses éléments. À

chaque itération, la boucle déplace un autre élément vers la variable . Puisque maintenant possède la chaîne, nous sommes en mesure de la modifier dans le corps de la boucle avant de l'imprimer. Et puisque le vecteur lui-même n'est plus visible par le code, rien ne peut l'observer en boucle dans un état partiellement vidé. `for ... in v v v for s s`

Si vous avez besoin de déplacer une valeur d'un propriétaire que le compilateur ne peut pas suivre, vous pouvez envisager de changer le type du propriétaire en quelque chose qui peut suivre dynamiquement s'il a une valeur ou non. Par exemple, voici une variante de l'exemple précédent :

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth: 1525 });
```

Vous ne pouvez pas faire ça:

```
let first_name = composers[0].name;
```

Cela suscitera simplement la même erreur « ne peut pas sortir de l'index » montrée précédemment. Mais parce que vous avez changé le type du champ de à , cela signifie qu'il s'agit d'une valeur légitime pour le champ à conserver, donc cela fonctionne

```
: name String Option<String> None
```

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

L'appel déplace la valeur de , laissant à sa place et transmet la propriété de la valeur d'origine à son appelant. En fait, l'utilisation de cette méthode est suffisamment courante pour que le type fournisse une méthode à cette fin. Vous pouvez écrire la manipulation précédente de manière plus lisible comme suit : `replace composers[0].name None Option take`

```
let first_name = composers[0].name.take();
```

Cet appel à a le même effet que l'appel précédent à . `take replace`

Types de copie : exception aux déplacements

Les exemples que nous avons montrés jusqu'à présent de valeurs déplacées impliquent des vecteurs, des chaînes et d'autres types qui pourraient potentiellement utiliser beaucoup de mémoire et être coûteux à copier. Les déménagements gardent la propriété de ces types claire et l'affection bon marché. Mais pour les types plus simples comme les entiers ou les caractères, ce genre de manipulation prudente n'est vraiment pas nécessaire.

Comparez ce qui se passe en mémoire lorsque nous attribuons un avec ce qui se passe lorsque nous attribuons une valeur : `String i32`

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

Après avoir exécuté ce code, la mémoire ressemble à [la Figure 4-11](#).

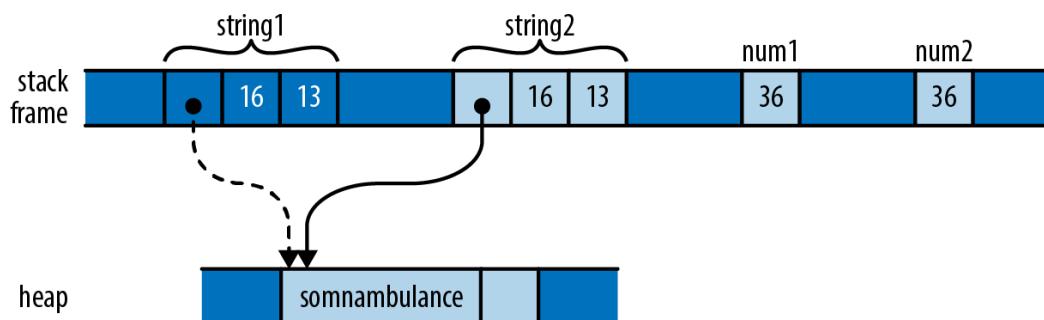


Figure 4-11. L'affectation d'un déplace la valeur, tandis que l'affectation d'un copie `String i32`

Comme pour les vecteurs précédents, l'affectation *se déplace vers* afin que nous ne nous retrouvions pas avec deux chaînes responsables de la libération du même tampon. Cependant, la situation avec est différente. An est simplement un modèle de bits en mémoire; il ne possède aucune ressource de tas ou ne dépend vraiment de rien d'autre que des octets qu'il comprend. Au moment où nous avons déplacé ses bits vers , nous avons fait une copie complètement indépendante de

`.string1 string2 num1 num2 i32 num2 num1`

Le déplacement d'une valeur laisse la source du déplacement non initialisée. Mais alors qu'il est essentiel de traiter comme sans valeur, traiter de cette façon est inutile; aucun dommage ne pourrait résulter de la poursuite de son utilisation. Les avantages d'un déménagement ne s'appliquent pas ici, et c'est gênant. `string1 num1`

Plus tôt, nous avons pris soin de dire que *la plupart des types* sont déplacés; nous en sommes maintenant arrivés aux exceptions, les types que Rust désigne comme *types de copie*. L'affectation d'une valeur d'un

type copie la valeur plutôt que de la déplacer. La source de l'affectation reste initialisée et utilisable, avec la même valeur qu'auparavant. Le passage de types aux fonctions et aux constructeurs se comporte de la même manière. `Copy`

Les types standard incluent tous les types numériques entiers et à virgule flottante de la machine, les types et quelques autres. Un tuple ou un tableau de types de taille fixe est lui-même un type. `Copy`

Seuls les types pour lesquels une simple copie bit pour bit suffit peuvent être. Comme nous l'avons déjà expliqué, n'est pas un type, car il possède un tampon alloué au tas. Pour des raisons similaires, n'est pas ; il est propriétaire de son référent attribué en tas. Le type, représentant un descripteur de fichier du système d'exploitation, n'est pas ; la duplication d'une telle valeur impliquerait de demander au système d'exploitation un autre descripteur de fichier. De même, le type, représentant un mutex verrouillé, ne l'est pas : ce type n'a aucun sens à copier, car un seul thread peut contenir un mutex à la

fois. `Copy String Copy Box<T> Copy File Copy MutexGuard Copy`

En règle générale, tout type qui a besoin de faire quelque chose de spécial lorsqu'une valeur est supprimée ne peut pas être : un besoin de libérer ses éléments, un doit fermer son descripteur de fichier, un doit déverrouiller son mutex, et ainsi de suite. La duplication bit pour bit de ces types ne permettrait pas de savoir quelle valeur était maintenant responsable des ressources de l'original. `Copy Vec File MutexGuard`

Qu'en est-il des types que vous définissez vous-même? Par défaut, et les types ne sont pas : `struct enum Copy`

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }

let l = Label { number: 3 };
print(l);
println!("My label number is: {}", l.number);
```

Cela ne compilera pas; Rust se plaint:

```
error: borrow of moved value: `l`
|
10 |     let l = Label { number: 3 };
|         - move occurs because `l` has type `main::Label`,
|             which does not implement the `Copy` trait
```

```

11 |     print(l);
|         - value moved here
12 |     println!("My label number is: {}", l.number);
|                                         ^
|
|         value borrowed here after move

```

Puisque n'est pas , en le passant pour déplacer la propriété de la valeur vers la fonction, qui l'a ensuite abandonnée avant de revenir. Mais c'est idiot; a n'est rien d'autre qu'un avec des prétentions. Il n'y a aucune raison de passer à devrait déplacer la

```
valeur. Label Copy print print Label u32 l print
```

Mais les types définis par l'utilisateur étant non- n'est que la valeur par défaut. Si tous les champs de votre structure sont eux-mêmes , alors vous pouvez également faire le type en plaçant l'attribut au-dessus de la définition, comme suit: Copy Copy Copy #[derive(Copy, Clone)]

```

#[derive(Copy, Clone)]
struct Label { number: u32 }

```

Avec cette modification, le code précédent se compile sans plainte. Cependant, si nous essayons cela sur un type dont les champs ne sont pas tous , cela ne fonctionne pas. Supposons que nous compilions le code suivant : Copy

```

#[derive(Copy, Clone)]
struct StringLabel { name: String }

```

Il déclenche cette erreur :

```

error: the trait `Copy` may not be implemented for this type
|
7 | #[derive(Copy, Clone)]
|      ^
|
8 | struct StringLabel { name: String }
|           ----- this field does not implement `Copy`

```

Pourquoi les types définis par l'utilisateur ne sont-ils pas automatiquement définis par l'utilisateur, en supposant qu'ils sont éligibles ? Qu'un type soit ou non a un effet important sur la façon dont le code est autorisé à l'utiliser : les types sont plus flexibles, car l'affectation et les opérations associées ne laissent pas l'original non initialisé. Mais pour l'implémenteur d'un type, c'est l'inverse qui est vrai : les types sont très limités dans les types qu'ils peuvent contenir, tandis que les non-types peuvent utiliser l'allocation de tas et posséder d'autres types de ressources. Donc,

faire un type représente un engagement sérieux de la part de l'implémenteur: s'il est nécessaire de le changer en non-plus tard, une grande partie du code qui l'utilise devra probablement être adaptée. Copy Copy Copy Copy Copy Copy

Alors que C++ vous permet de surcharger les opérateurs d'affectation et de définir des constructeurs de copie et de déplacement spécialisés, Rust ne permet pas ce type de personnalisation. Dans Rust, chaque mouvement est une copie superficielle octet pour octet qui laisse la source non initialisée. Les copies sont les mêmes, sauf que la source reste initialisée. Cela signifie que les classes C++ peuvent fournir des interfaces pratiques que les types Rust ne peuvent pas, où le code d'apparence ordinaire ajuste implicitement le nombre de références, reporte les copies coûteuses pour plus tard ou utilise d'autres astuces d'implémentation sophistiquées.

Mais l'effet de cette flexibilité sur C++ en tant que langage est de rendre les opérations de base telles que l'affectation, la transmission de paramètres et le retour de valeurs à partir de fonctions moins prévisibles. Par exemple, plus haut dans ce chapitre, nous avons montré comment l'affectation d'une variable à une autre en C++ peut nécessiter des quantités arbitraires de mémoire et de temps processeur. L'un des principes de Rust est que les coûts doivent être apparents pour le programmeur. Les opérations de base doivent rester simples. Les opérations potentiellement coûteuses doivent être explicites, comme les appels à dans l'exemple précédent qui font des copies profondes des vecteurs et des chaînes qu'ils contiennent. `clone`

Dans cette section, nous avons parlé en termes vagues des caractéristiques qu'un type pourrait avoir. Ce sont en fait des *exemples de traits*, la facilité ouverte de Rust pour catégoriser les types en fonction de ce que vous pouvez en faire. Nous décrivons les traits en général au [chapitre 11](#), et en particulier au [chapitre 13](#). Copy Clone Copy Clone

Rc et Arc : propriété partagée

Bien que la plupart des valeurs aient des propriétaires uniques dans le code Rust typique, dans certains cas, il est difficile de trouver chaque valeur d'un seul propriétaire qui a la durée de vie dont vous avez besoin; vous aimeriez que la valeur vive simplement jusqu'à ce que tout le monde ait fini de l'utiliser. Pour ces cas, Rust fournit les types de pointeur comptés par référence et . Comme on peut s'y attendre de Rust, ceux-ci sont entièrement sûrs à utiliser : vous ne pouvez pas oublier d'ajuster le nombre de références, de créer d'autres pointeurs vers le référent que

Rust ne remarque pas, ou de trébucher sur l'un des autres types de problèmes qui accompagnent les types de pointeurs comptés par référence en C++. `Rc` `Arc`

Les deux types sont très similaires; la seule différence entre eux est qu'il est sûr de partager directement entre les threads (le nom est l'abréviation de *atomic reference count*), tandis qu'un simple utilise un code non thread-safe plus rapide pour mettre à jour son nombre de références. Si vous n'avez pas besoin de partager les pointeurs entre les threads, il n'y a aucune raison de payer la pénalité de performance d'un , vous devez donc utiliser ; La rouille vous empêchera d'en passer accidentellement un à travers une limite de filetage. Les deux types sont par ailleurs équivalents, donc pour le reste de cette section, nous ne parlerons que de

`.Rc` `Arc` `Arc` `Rc` `Arc` `Rc` `Rc`

Plus tôt, nous avons montré comment Python utilise le nombre de références pour gérer la durée de vie de ses valeurs. Vous pouvez utiliser pour obtenir un effet similaire dans Rust. Considérez le code suivant : `Rc`

```
use std::rc::Rc;

// Rust can infer all these types; written out for clarity
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

Pour tout type , une valeur est un pointeur vers un tas alloué auquel un nombre de références a été apposé. Le clonage d'une valeur ne copie pas le ; au lieu de cela, il crée simplement un autre pointeur vers celui-ci et incrémenté le nombre de références. Ainsi, le code précédent produit la situation illustrée à [la figure 4-12](#) en mémoire. T `Rc<T>` T `Rc<T>` T

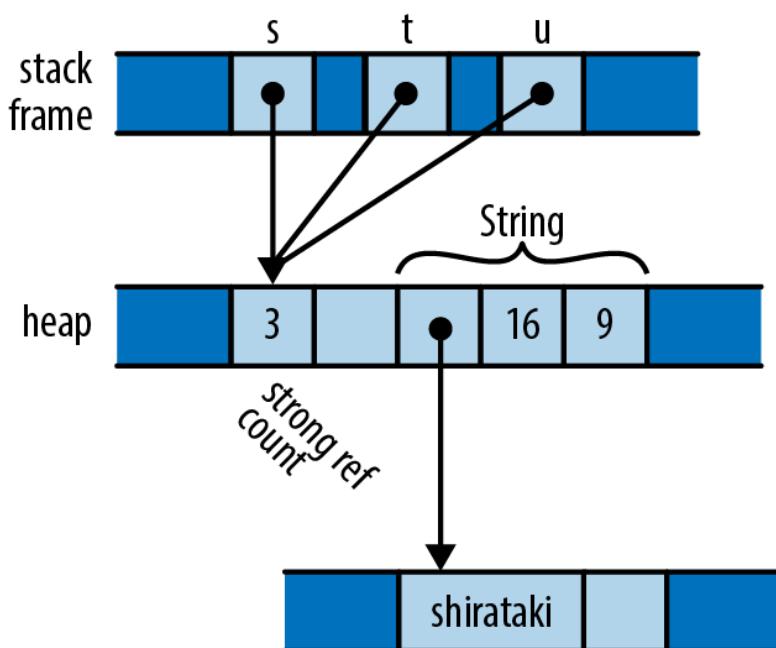


Figure 4-12. Chaîne comptée par références avec trois références

Chacun des trois pointeurs fait référence au même bloc de mémoire, qui contient un nombre de références et un espace pour le . Les règles de propriété habituelles s'appliquent aux pointeurs eux-mêmes, et lorsque le dernier existant est abandonné, Rust le laisse également tomber.
`Rc<String> String Rc Rc String`

Vous pouvez utiliser n'importe laquelle des méthodes habituelles directement sur un :
`String Rc<String>`

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

Une valeur appartenant à un pointeur est immuable. Supposons que vous essayiez d'ajouter du texte à la fin de la chaîne :
`Rc`

```
s.push_str(" noodles");
```

La rouille diminuera :

```
error: cannot borrow data in an `Rc` as mutable
|
13 |     s.push_str(" noodles");
|     ^ cannot borrow as mutable
|
```

Les garanties de sécurité de la mémoire et du thread de Rust dépendent de la garantie qu'aucune valeur n'est jamais partagée et mutable simultanément. Rust suppose que le référent d'un pointeur peut en général

être partagé, il ne doit donc pas être mutable. Nous expliquons pourquoi cette restriction est importante au [chapitre 5](#). Rc

Un problème bien connu avec l'utilisation du nombre de références pour gérer la mémoire est que, si jamais il y a deux valeurs comptées par référence qui pointent l'une vers l'autre, chacune maintiendra le nombre de références de l'autre au-dessus de zéro, de sorte que les valeurs ne seront jamais libérées ([Figure 4-13](#)).

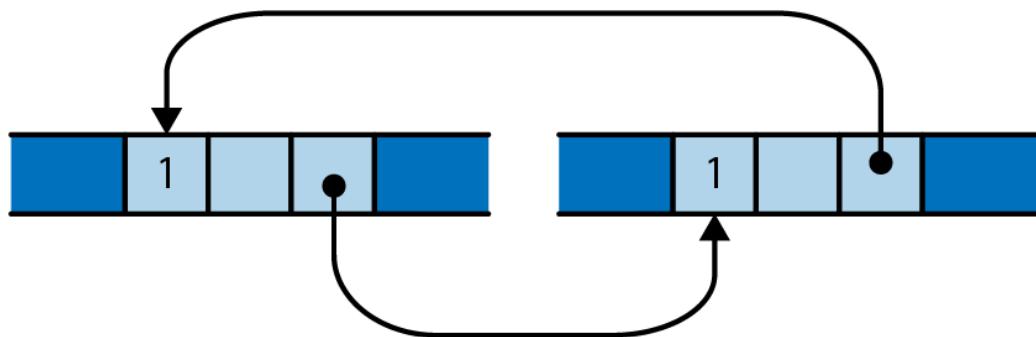


Figure 4-13. Une boucle de comptage de références; ces objets ne seront pas libérés

Il est possible de divulguer des valeurs dans Rust de cette façon, mais de telles situations sont rares. Vous ne pouvez pas créer un cycle sans, à un moment donné, faire pointer une valeur plus ancienne vers une valeur plus récente. Cela nécessite évidemment que l'ancienne valeur soit mutable. Étant donné que les pointeurs maintiennent leurs référents immuables, il n'est normalement pas possible de créer un cycle. Cependant, Rust fournit des moyens de créer des parties mutables de valeurs autrement immuables; c'est ce *qu'on appelle la mutabilité intérieure*, et nous la couvrons dans [« Mutabilité intérieure »](#). Si vous combinez ces techniques avec des pointeurs, vous pouvez créer un cycle et une mémoire de fuite. Rc Rc

Vous pouvez parfois éviter de créer des cycles de pointeurs en utilisant des *pointeurs faibles*, pour certains des liens à la place. Cependant, nous ne les couvrirons pas dans ce livre; consultez la documentation de la bibliothèque standard pour plus de détails. Rc `std::rc::Weak`

Les mouvements et les pointeurs comptés par référence sont deux façons de détendre la rigidité de l'arbre de propriété. Dans le chapitre suivant, nous examinerons une troisième voie : emprunter des références à des valeurs. Une fois que vous êtes à l'aise avec la propriété et l'emprunt, vous aurez gravi la partie la plus raide de la courbe d'apprentissage de Rust, et vous serez prêt à tirer parti des forces uniques de Rust.

Chapitre 5. Références

Les bibliothèques ne peuvent pas fournir de nouvelles incapacités.

—Mark Miller

Tous les types de pointeurs que nous avons vus jusqu'à présent (le pointeur de tas simple et les pointeurs internes à et valeurs) possèdent des pointeurs : lorsque le propriétaire est supprimé, le référent l'accompagne. Rust a également des types de pointeurs non *propriétaires* appelés *références*, qui n'ont aucun effet sur la durée de vie de leurs référents. Box<T> String Vec

En fait, c'est plutôt le contraire : les références ne doivent jamais survivre à leurs référents. Vous devez indiquer clairement dans votre code qu'aucune référence ne peut survivre à la valeur vers laquelle elle pointe. Pour souligner cela, Rust se réfère à la création d'une référence à une certaine valeur comme *emprunt de la valeur*: ce que vous avez emprunté, vous devez éventuellement retourner à son propriétaire.

Si vous avez ressenti un moment de scepticisme en lisant la phrase « Vous devez le faire apparaître dans votre code », vous êtes en excellente compagnie. Les références elles-mêmes n'ont rien de spécial – sous le capot, ce ne sont que des adresses. Mais les règles qui les gardent en sécurité sont nouvelles pour Rust; en dehors des langages de recherche, vous n'aurez jamais rien vu de tel auparavant. Et bien que ces règles soient la partie de Rust qui nécessite le plus d'efforts pour maîtriser, l'ampleur des bugs classiques et absolument quotidiens qu'elles empêchent est surprenante, et leur effet sur la programmation multithread est libérateur. C'est le pari radical de Rust, encore une fois.

Dans ce chapitre, nous allons passer en revue le fonctionnement des références dans Rust; montrer comment les références, les fonctions et les types définis par l'utilisateur intègrent tous les informations de durée de vie pour s'assurer qu'ils sont utilisés en toute sécurité ; et illustrer certaines catégories courantes de bogues que ces efforts empêchent, au moment de la compilation et sans pénalités de performance au moment de l'exécution.

Références aux valeurs

À titre d'exemple, supposons que nous allons construire un tableau d'artistes meurtriers de la Renaissance et des œuvres pour lesquelles ils sont connus. La bibliothèque standard de Rust comprend un type de table de hachage, de sorte que nous pouvons définir notre type comme ceci:

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

type Table = HashMap<String, Vec<String>>;
```

En d'autres termes, il s'agit d'une table de hachage qui mappe les valeurs aux valeurs, en prenant le nom d'un artiste à une liste des noms de ses œuvres. Vous pouvez itérer sur les entrées d'un avec une boucle, de sorte que nous pouvons écrire une fonction pour imprimer un

```
: String Vec<String> HashMap for Table
```

```
fn show(table: Table) {
    for (artist, works) in table {
        println!("works by {}: ", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

La construction et l'impression de la table sont simples :

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                 vec![ "many madrigals".to_string(),
                        "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                 vec![ "The Musicians".to_string(),
                        "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                 vec![ "Perseus with the head of Medusa".to_string(),
                        "a salt cellar".to_string()]);

    show(table);
}
```

Et tout fonctionne bien:

```

$ cargo run
    Running `/home/jimb/rust/book/fragments/target/debug/fragments`
works by Gesualdo:
many madrigals
Tenebrae Responsoria
works by Cellini:
Perseus with the head of Medusa
a salt cellar
works by Caravaggio:
The Musicians
The Calling of St. Matthew
$
```

Mais si vous avez lu la section du chapitre précédent sur les déménagements, cette définition devrait soulever quelques questions. En particulier, n'est pas - il ne peut pas l'être, car il possède une table allouée dynamiquement. Ainsi, lorsque le programme appelle , toute la structure est déplacée vers la fonction, laissant la variable non initialisée. (Il itére également sur son contenu sans ordre spécifique, donc si vous avez obtenu un ordre différent, ne vous inquiétez pas.) Si le code d'appel essaie d'être utilisé maintenant, il rencontrera des problèmes

```
: show HashMap Copy show(table) table table
```

```

...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust se plaint qu'il n'est plus disponible: table

```

error: borrow of moved value: `table`
|
20 |     let mut table = Table::new();
|     ----- move occurs because `table` has type
|           `HashMap<String, Vec<String>>`,
|           which does not implement the `Copy` trait
...
31 |     show(table);
|     ----- value moved here
32 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
|           ^^^^^^ value borrowed here after move
```

En fait, si nous examinons la définition de , la boucle externe prend possession de la table de hachage et la consomme entièrement; et la boucle

interne fait de même pour chacun des vecteurs. (Nous avons vu ce comportement plus tôt, dans l'exemple « liberté, égalité, fraternité ».) En raison de la sémantique de déplacement, nous avons complètement détruit toute la structure simplement en essayant de l'imprimer. Merci,

Rust! show for for

La bonne façon de gérer cela est d'utiliser des références. Une référence vous permet d'accéder à une valeur sans affecter sa propriété. Les références se déclinent en deux sortes :

- Une *référence partagée* vous permet de lire mais pas de modifier son référent. Cependant, vous pouvez avoir autant de références partagées à une valeur particulière à la fois que vous le souhaitez. L'expression donne une référence partagée à la valeur de ' ; si a le type , alors a le type , prononcé « ref ». Les références partagées sont . &e e e T &e &T T Copy
- Si vous avez une *référence modifiable* à une valeur, vous pouvez à la fois lire et modifier la valeur. Cependant, il se peut que vous n'ayez aucune autre référence d'aucune sorte à cette valeur active en même temps. L'expression donne une référence mutable à la valeur de ' ; vous écrivez son type comme , qui se prononce « ref mute ». Les références modifiables ne sont pas . &mut e e &mut T T Copy

Vous pouvez considérer la distinction entre les références partagées et mutables comme un moyen d'appliquer une règle de *plusieurs lecteurs ou d'un seul rédacteur* au moment de la compilation. En fait, cette règle ne s'applique pas seulement aux références; il couvre également le propriétaire de la valeur empruntée. Tant qu'il y a des références partagées à une valeur, même son propriétaire ne peut pas la modifier; la valeur est verrouillée. Personne ne peut modifier pendant qu'il travaille avec. De même, s'il existe une référence modifiable à une valeur, elle a un accès exclusif à la valeur ; vous ne pouvez pas utiliser le propriétaire du tout, jusqu'à ce que la référence mutable disparaisse. Garder le partage et la mutation complètement séparés s'avère essentiel à la sécurité de la mémoire, pour des raisons que nous aborderons plus loin dans le chapitre. table show

La fonction d'impression de notre exemple n'a pas besoin de modifier le tableau, il suffit de lire son contenu. Ainsi, l'appelant devrait être en mesure de lui transmettre une référence partagée à la table, comme suit :

```
show(&table);
```

Les références sont des pointeurs non propriétaires, de sorte que la variable reste propriétaire de l'ensemble de la structure ; vient de l'empêtrer un peu. Naturellement, nous devrons ajuster la définition de pour correspondre, mais vous devrez regarder de près pour voir la différence:

```
table show show
```

```
fn show(table: &Table) {  
    for (artist, works) in table {  
        println!("works by {}: ", artist);  
        for work in works {  
            println!("    {}", work);  
        }  
    }  
}
```

Le type de paramètre de 's est passé de à : au lieu de passer la table par valeur (et donc de déplacer la propriété dans la fonction), nous passons maintenant une référence partagée. C'est le seul changement textuel. Mais comment cela se passe-t-il lorsque nous travaillons à travers le corps?

```
show table Table &Table
```

Alors que notre boucle extérieure d'origine s'en est emparée et l'a consommée, dans notre nouvelle version, elle reçoit une référence partagée au . L'itération sur une référence partagée à a est définie pour produire des références partagées à la clé et à la valeur de chaque entrée : est passée de a à a , et de a à a

```
. for HashMap<String> artist String &String works Vec<String> &Vec<String>
```

La boucle interne est modifiée de la même manière. L'itération sur une référence partagée à un vecteur est définie pour produire des références partagées à ses éléments, de sorte qu'un . Aucun propriétaire ne change de mains dans cette fonction; c'est juste passer autour de références non propriétaires.

```
work &String
```

Maintenant, si nous voulions écrire une fonction pour alphabétiser les œuvres de chaque artiste, une référence partagée ne suffit pas, car les références partagées ne permettent pas la modification. Au lieu de cela, la fonction de tri doit prendre une référence modifiable à la table :

```
fn sort_works(table: &mut Table) {
    for (_artist, works) in table {
        works.sort();
    }
}
```

Et nous devons l'adopter un:

```
sort_works(&mut table);
```

Cet emprunt mutable donne la possibilité de lire et de modifier notre structure, comme l'exige la méthode des vecteurs. `sort_works` sort

Lorsque nous passons une valeur à une fonction d'une manière qui déplace la propriété de la valeur vers la fonction, nous disons que nous l'avons passée *par valeur*. Si nous passons plutôt à la fonction une référence à la valeur, nous disons que nous avons passé la valeur *par référence*. Par exemple, nous avons corrigé notre fonction en la modifiant pour accepter la table par référence, plutôt que par valeur. De nombreuses langues établissent cette distinction, mais elle est particulièrement importante dans Rust, car elle explique comment la propriété est affectée. `show`

Utilisation des références

L'exemple précédent montre une utilisation assez typique des références : permettre aux fonctions d'accéder à une structure ou de la manipuler sans s'en approprier. Mais les références sont plus flexibles que cela, alors regardons quelques exemples pour avoir une vue plus détaillée de ce qui se passe.

Références Rust versus références C++

Si vous êtes familier avec les références en C++, elles ont quelque chose en commun avec les références Rust. Plus important encore, ce ne sont que des adresses au niveau de la machine. Mais dans la pratique, les références de Rust ont une sensation très différente.

En C++, les références sont créées implicitement par conversion, et déréférencées implicitement :

```
// C++ code!
int x = 10;
int &r = x;           // initialization creates reference implicitly
assert(r == 10);     // implicitly dereference r to see x's value
r = 20;              // stores 20 in x, r itself still points to x
```

Dans Rust, les références sont créées explicitement avec l'opérateur et déréférencées explicitement avec l'opérateur : & *

```
// Back to Rust code from this point onward.
let x = 10;
let r = &x;           // &x is a shared reference to x
assert!(*r == 10);   // explicitly dereference r
```

Pour créer une référence modifiable, utilisez l'opérateur : &mut

```
let mut y = 32;
let m = &mut y;       // &mut y is a mutable reference to y
*m += 32;            // explicitly dereference m to set y's value
assert!(*m == 64);   // and to see y's new value
```

Mais vous vous souviendrez peut-être que, lorsque nous avons fixé la fonction de prendre la table des artistes par référence plutôt que par valeur, nous n'avons jamais eu à utiliser l'opérateur. Pourquoi? show *

Étant donné que les références sont si largement utilisées dans Rust, l'opérateur déréférence implicitement son opérande gauche, si nécessaire
..

```
struct Anime { name: &'static str, bechdel_pass: bool }
let aria = Anime { name: "Aria: The Animation", bechdel_pass: true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// Equivalent to the above, but with the dereference written out:
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

La macro utilisée dans la fonction s'étend au code qui utilise l'opérateur, de sorte qu'elle tire également parti de cette déréférence implicite. `println! show .`

L'opérateur peut également emprunter implicitement une référence à son opérande gauche, si nécessaire pour un appel de méthode. Par exem-

ple, la méthode de ' prend une référence mutable au vecteur, de sorte que ces deux appels sont équivalents : . Vec sort

```
let mut v = vec![1973, 1968];
v.sort();           // implicitly borrows a mutable reference to v
(&mut v).sort();  // equivalent, but more verbose
```

En un mot, alors que C++ convertit implicitement entre les références et les valeurs l (c'est-à-dire les expressions faisant référence à des emplacements en mémoire), ces conversions apparaissent partout où elles sont nécessaires, dans Rust, vous utilisez les opérateurs and pour créer et suivre des références, à l'exception de l'opérateur, qui emprunte et déréférence implicitement. & * .

Affectation de références

L'affectation d'une référence à une variable rend ce point de variable quelque chose de nouveau :

```
let x = 10;
let y = 20;
let mut r = &x;

if b { r = &y; }

assert!(*r == 10 || *r == 20);
```

La référence pointe d'abord vers . Mais si c'est vrai, le code le pointe plutôt, comme illustré à [la figure 5-1](#). r x b y

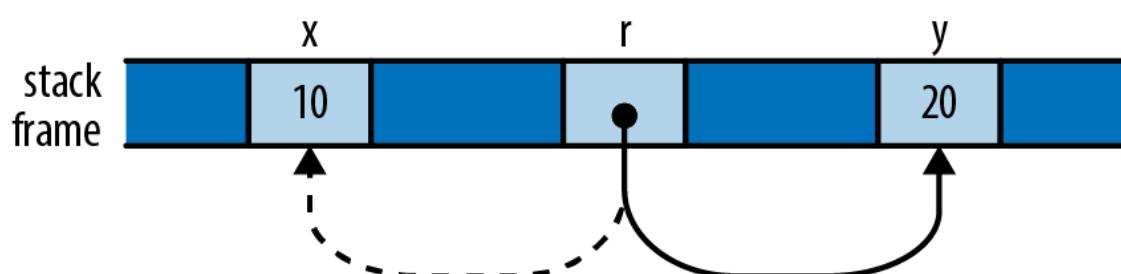


Figure 5-1. La référence , pointant maintenant vers au lieu de r y x

Ce comportement peut sembler trop évident pour mériter d'être mentionné: bien sûr, cela pointe maintenant vers , puisque nous y avons stocké. Mais nous le soulignons parce que les références C++ se comportent très différemment : comme indiqué précédemment, l'attribution d'une valeur à une référence en C++ stocke la valeur dans son référent. Une fois

qu'une référence C++ a été initialisée, il n'y a aucun moyen de la faire pointer vers autre chose. `r` `y` `&y`

Références aux références

La rouille autorise les références aux références :

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr: &&&Point = &rr;
```

(Nous avons écrit les types de référence pour plus de clarté, mais vous pouvez les omettre; il n'y a rien ici que Rust ne puisse pas déduire pour lui-même.) L'opérateur suit autant de références qu'il en faut pour trouver sa cible : .

```
assert_eq!(rrr.y, 729);
```

En mémoire, les références sont disposées comme illustré à [la figure 5-2](#).

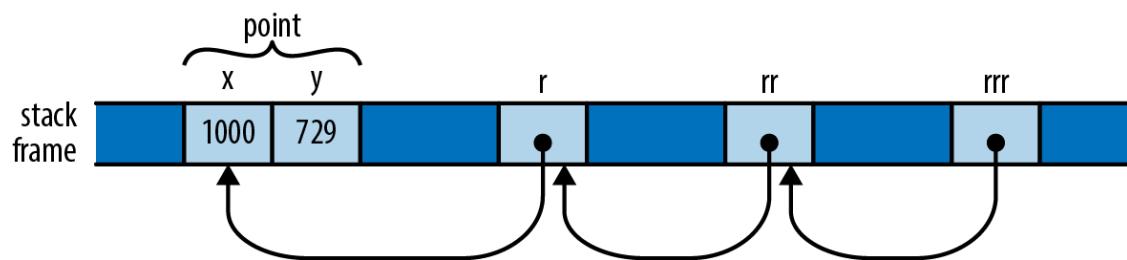


Figure 5-2. Une chaîne de références aux références

Ainsi, l'expression , guidée par le type de , traverse en fait trois références pour arriver à l'avant d'aller chercher son champ. `rrr.y` `rrr` `Point` `y`

Comparaison des références

Comme l'opérateur , les opérateurs de comparaison de Rust « voient à travers » n'importe quel nombre de références: .

```
let x = 10;
let y = 10;

let rx = &x;
let ry = &y;
```

```

let rrx = &rx;
let rry = &ry;

assert!(rrx <= rry);
assert!(rrx == rry);

```

L'assertion finale ici réussit, même si et pointent vers des valeurs différentes (à savoir, et), parce que l'opérateur suit toutes les références et effectue la comparaison sur leurs cibles finales, et . C'est presque toujours le comportement que vous souhaitez, en particulier lors de l'écriture de fonctions génériques. Si vous voulez réellement savoir si deux références pointent vers la même mémoire, vous pouvez utiliser , qui les compare en tant qu'adresses : rrx rry rx ry == x y std::ptr::eq

```

assert!(rx == ry);           // their referents are equal
assert!(!std::ptr::eq(rx, ry)); // but occupy different addresses

```

Notez que les opérandes d'une comparaison doivent avoir exactement le même type, y compris les références :

```

assert!(rx == rrx);      // error: type mismatch: `&i32` vs `*&i32`
assert!(rx == *rrx);     // this is okay

```

Les références ne sont jamais nulles

Les références Rust ne sont jamais nulles. Il n'y a pas d'analogie aux C ou C++. Il n'y a pas de valeur initiale par défaut pour une référence (vous ne pouvez utiliser aucune variable tant qu'elle n'a pas été initialisée, quel que soit son type) et Rust ne convertira pas les entiers en références (en dehors du code), vous ne pouvez donc pas convertir zéro en référence. `NULL` `nullptr` `unsafe`

Le code C et C++ utilise souvent un pointeur NULL pour indiquer l'absence de valeur : par exemple, la fonction renvoie soit un pointeur vers un nouveau bloc de mémoire, soit s'il n'y a pas assez de mémoire disponible pour satisfaire la demande. Dans Rust, si vous avez besoin d'une valeur qui est une référence à quelque chose ou non, utilisez le type . Au niveau de la machine, Rust représente comme un pointeur null et , où est une valeur, comme l'adresse non nulle, est donc tout aussi efficace qu'un pointeur nullable en C ou C ++, même s'il est plus sûr: son type vous oblige à vérifier si c'est avant de pouvoir

```
l'utiliser. malloc nullptr Option<&T> None Some(r) r &T Option<&
T> None
```

Emprunt de références à des expressions arbitraires

Alors que C et C++ ne vous permettent d'appliquer l'opérateur qu'à certains types d'expressions, Rust vous permet d'emprunter une référence à la valeur de tout type d'expression : &

```
fn factorial(n: usize) -> usize {
    (1..n+1).product()
}
let r = &factorial(6);
// Arithmetic operators can see through one level of references.
assert_eq!(r + &1009, 1729);
```

Dans des situations comme celle-ci, Rust crée simplement une variable anonyme pour conserver la valeur de l'expression et en fait référence. La durée de vie de cette variable anonyme dépend de ce que vous faites avec la référence :

- Si vous affectez immédiatement la référence à une variable dans une instruction (ou si vous l'intégrez à une structure ou à un tableau qui est immédiatement affecté), Rust fait vivre la variable anonyme aussi longtemps que la variable initialise. Dans l'exemple précédent, Rust le ferait pour le référent de .`let r`
- Sinon, la variable anonyme vit jusqu'à la fin de l'instruction en-globante. Dans notre exemple, la variable anonyme créée pour tenir ne dure que jusqu'à la fin de l'instruction `1009 assert_eq!`

Si vous êtes habitué à C ou C++, cela peut sembler sujet aux erreurs. Mais rappelez-vous que Rust ne vous laissera jamais écrire du code qui produirait une référence pendante. Si jamais la référence pouvait être utilisée au-delà de la durée de vie de la variable anonyme, Rust vous signalera toujours le problème au moment de la compilation. Vous pouvez ensuite corriger votre code pour conserver le référent dans une variable nommée avec une durée de vie appropriée.

Références aux tranches et aux objets traits

Les références que nous avons montrées jusqu'à présent sont toutes des adresses simples. Cependant, Rust comprend également deux types de *pointeurs de graisse*, *des valeurs* de deux mots portant l'adresse d'une certaine valeur, ainsi que des informations supplémentaires nécessaires pour utiliser la valeur.

Une référence à une tranche est un pointeur de graisse, portant l'adresse de départ de la tranche et sa longueur. Nous avons décrit les tranches en détail au [chapitre 3](#).

L'autre type de pointeur de graisse de Rust est un *objet de trait*, une référence à une valeur qui implémente un certain trait. Un objet trait porte l'adresse d'une valeur et un pointeur vers l'implémentation du trait appropriée à cette valeur, pour appeler les méthodes du trait. Nous couvrirons les objets traits en détail dans [« Objets traits »](#).

En plus de transporter ces données supplémentaires, les références d'objets de tranches et de traits se comportent comme les autres types de références que nous avons montrées jusqu'à présent dans ce chapitre : elles ne possèdent pas leurs référents, elles ne sont pas autorisées à survivre à leurs référents, elles peuvent être mutables ou partagées, etc.

Sécurité de référence

Comme nous les avons présentés jusqu'à présent, les références ressemblent à peu près à des pointeurs ordinaires en C ou C++. Mais ceux-ci ne sont pas sûrs; Comment Rust garde-t-il ses références sous contrôle ? Peut-être que la meilleure façon de voir les règles en action est d'essayer de les enfreindre.

Pour transmettre les idées fondamentales, nous commencerons par les cas les plus simples, en montrant comment Rust s'assure que les références sont utilisées correctement dans un seul corps de fonction. Ensuite, nous examinerons le passage de références entre les fonctions et leur stockage dans des structures de données. Cela implique de donner à ces fonctions et types *de données des paramètres de durée de vie*, que nous allons expliquer. Enfin, nous présenterons quelques raccourcis fournis par Rust pour simplifier les modèles d'utilisation courants. Tout au long, nous montrerons comment Rust pointe le code cassé et suggère souvent des solutions.

Emprunt d'une variable locale

Voici un cas assez évident. Vous ne pouvez pas emprunter une référence à une variable locale et la retirer de la portée de la variable :

```
{  
    let r;  
    {  
        let x = 1;  
        r = &x;  
    }  
    assert_eq!(*r, 1); // bad: reads memory `x` used to occupy  
}
```

Le compilateur Rust rejette ce programme, avec un message d'erreur détaillé :

```
error: `x` does not live long enough  
|  
7 |         r = &x;  
|             ^^^ borrowed value does not live long enough  
8 |     }  
|     - `x` dropped here while still borrowed  
9 |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy  
10 | }
```

La plainte de Rust est qu'elle ne vit que jusqu'à la fin du bloc intérieur, alors que la référence reste vivante jusqu'à la fin du bloc extérieur, ce qui en fait un pointeur pendant, qui est verboten. x

Bien qu'il soit évident pour un lecteur humain que ce programme est cassé, il vaut la peine de regarder comment Rust lui-même est arrivé à cette conclusion. Même cet exemple simple montre les outils logiques que Rust utilise pour vérifier un code beaucoup plus complexe.

Rust essaie d'attribuer à chaque type de référence de votre programme une *durée de vie* qui répond aux contraintes imposées par son utilisation. Une durée de vie est un tronçon de votre programme pour lequel une référence pourrait être utilisée en toute sécurité : une instruction, une expression, la portée d'une variable ou autre. Les durées de vie sont entièrement le fruit de l'imagination de Rust au moment de la compilation. Au moment de l'exécution, une référence n'est rien d'autre qu'une adresse ;

sa durée de vie fait partie de son type et n'a pas de représentation au moment de l'exécution.

Dans cet exemple, il y a trois vies dont nous devons établir les relations. Les variables et les deux ont une durée de vie, s'étendant du point où elles sont initialisées jusqu'au moment où le compilateur peut prouver qu'elles ne sont plus utilisées. La troisième durée de vie est celle d'un type de référence : le type de la référence à laquelle nous empruntons et stockons dans . r x x r

Voici une contrainte qui devrait sembler assez évidente : si vous avez une variable , alors une référence à ne doit pas survivre d'elle-même, comme le montre [la figure 5-3](#). x x x

Au-delà du point où elle sort du champ d'application, la référence serait un pointeur pendant. Nous disons que la durée de vie de la variable doit contenir ou enfermer celle de la référence qui lui est *empruntée*. x

```
{  
    let r;  
    {  
        let x = 1;  
        ...  
        r = &x;  
        ...  
    }  
    assert_eq!(*r, 1);  
}
```

lifetime of &x must not exceed this range

Figure 5-3. Durées de vie admissibles pour &x

Voici un autre type de contrainte : si vous stockez une référence dans une variable , le type de référence doit être valable pendant toute la durée de vie de la variable, de son initialisation jusqu'à sa dernière utilisation, comme le montre [la figure 5-4](#). r

Si la référence ne peut pas vivre au moins aussi longtemps que la variable, alors à un moment donné sera un pointeur pendant. Nous disons que la durée de vie de la référence doit contenir ou entourer celle de la variable. r

```

{
    let r;
{
    let x = 1;
    ...
    r = &x;
    ...
}
assert_eq!(*r, 1);
}

```

lifetime of anything stored in
r must cover at least this range

Figure 5-4. Durées de vie admissibles pour la référence stockée dans r

Le premier type de contrainte limite la taille de la durée de vie d'une référence, tandis que le second type limite sa petite taille. Rust essaie simplement de trouver une durée de vie pour chaque référence qui réponde à toutes ces contraintes. Dans notre exemple, cependant, il n'y a pas une telle durée de vie, comme le montre [la figure 5-5](#).

```

{
    let r;
{
    let x = 1;
    ...
    r = &x;
    ...
}
assert_eq!(*r, 1);
}

```

There is no lifetime that lies
entirely within this range...

...but also fully encloses this range.

Figure 5-5. Une référence avec des contraintes contradictoires sur sa durée de vie

Considérons maintenant un exemple différent où les choses fonctionnent. Nous avons les mêmes types de contraintes : la durée de vie de la référence doit être contenue par 's, mais entièrement fermée 's. Mais parce que la durée de vie de est plus petite maintenant, il y a une durée de vie qui répond aux contraintes, comme le montre [la figure 5-6](#).

```

{
    let x = 1;
    {
        let r = &x;
        ...
        assert_eq! (*r, 1);
        ...
    }
}

```

The inner lifetime covers the lifetime of `r`, but is fully enclosed by the lifetime of `x`.

Figure 5-6. Une référence avec une durée de vie englobant la portée de `'`, mais dans la portée de `"r x"`

Ces règles s'appliquent de manière naturelle lorsque vous empruntez une référence à une partie d'une structure de données plus vaste, comme un élément d'un vecteur :

```

let v = vec![1, 2, 3];
let r = &v[1];

```

Puisque possède le vecteur, qui possède ses éléments, la durée de vie de doit englober celle du type de référence de . De même, si vous stockez une référence dans une structure de données, sa durée de vie doit inclure celle de la structure de données. Par exemple, si vous construisez un vecteur de références, toutes doivent avoir des durées de vie englobant celle de la variable propriétaire du vecteur. `v v &v[1]`

C'est l'essence du processus que Rust utilise pour tout le code. L'intégration de plus de fonctionnalités de langage dans l'image , par exemple, les structures de données et les appels de fonctions, introduit de nouveaux types de contraintes, mais le principe reste le même : premièrement, comprendre les contraintes découlant de la façon dont le programme utilise les références ; ensuite, trouvez des durées de vie qui les satisfont. Ce n'est pas si différent du processus que les programmeurs C et C++ s'imposent à eux-mêmes; la différence est que Rust connaît les règles et les applique.

Réception de références en tant qu'arguments de fonction

Lorsque nous passons une référence à une fonction, comment Rust s'assure-t-elle que la fonction l'utilise en toute sécurité ? Supposons que nous ayons une fonction qui prend une référence et la stocke dans une vari-

able globale. Nous devrons apporter quelques révisions à cela, mais voici une première coupe: f

```
// This code has several problems, and doesn't compile.  
static mut STASH: &i32;  
fn f(p: &i32) { STASH = p; }
```

L'équivalent de Rust d'une variable globale est appelé *statique*: c'est une valeur qui est créée au démarrage du programme et qui dure jusqu'à sa fin. (Comme toute autre déclaration, le système de modules de Rust contrôle où les statiques sont visibles, de sorte qu'elles ne sont que « globales » au cours de leur vie, pas leur visibilité.) Nous couvrons la statique dans [le chapitre 8](#), mais pour l'instant, nous allons simplement appeler quelques règles que le code qui vient d'être affiché ne suit pas:

- Chaque statique doit être initialisé.
- Les statiques mutables ne sont pas intrinsèquement thread-safe (après tout, n'importe quel thread peut accéder à un static à tout moment), et même dans les programmes monothread, ils peuvent être la proie d'autres types de problèmes de réentrance. Pour ces raisons, vous pouvez accéder à une statique modifiable uniquement dans un bloc. Dans cet exemple, nous ne sommes pas concernés par ces problèmes particuliers, nous allons donc simplement jeter un bloc et passer à autre chose. `unsafe unsafe`

Une fois ces révisions effectuées, nous avons maintenant ce qui suit :

```
static mut STASH: &i32 = &128;  
fn f(p: &i32) { // still not good enough  
    unsafe {  
        STASH = p;  
    }  
}
```

Nous avons presque terminé. Pour voir le problème restant, nous devons écrire quelques choses que Rust nous laisse utilement omettre. La signature de tel qu'écrit ici est en fait un raccourci pour ce qui suit: f

```
fn f<'a>(p: &'a i32) { ... }
```

Ici, la durée de vie (prononcée « tick A ») est un *paramètre de durée de vie* de . Vous pouvez lire comme « pour n'importe quelle vie » donc quand

nous écrivons , nous définissons une fonction qui prend une référence à une avec une durée de vie donnée . ' a f <'a> ' a fn f<'a>(p: &'a i32) i32 ' a

Puisque nous devons permettre d'être n'importe quelle vie, les choses feraient mieux de fonctionner si c'est la plus petite vie possible: une seule en enfermant simplement l'appel à . Cette mission devient alors un point de discorde : ' a f

```
STASH = p;
```

Puisque vit pour l'ensemble de l'exécution du programme, le type de référence qu'il détient doit avoir une durée de vie de la même durée; Rust appelle cela la « *durée de vie statique* ». Mais la durée de vie de la référence de est certaine, ce qui pourrait être n'importe quoi, tant qu'elle enferme l'appel à . Ainsi, Rust rejette notre code: STASH p ' a f

```
error: explicit lifetime required in the type of `p`  
|  
5 |     STASH = p;  
|           ^ lifetime `static` required
```

À ce stade, il est clair que notre fonction ne peut pas accepter n'importe quelle référence comme argument. Mais comme le souligne Rust, il devrait être en mesure d'accepter une référence qui a une durée de vie: stocker une telle référence dans ne peut pas créer un pointeur pendant. Et en effet, le code suivant se compile très bien: 'static STASH

```
static mut STASH: &i32 = &10;  
  
fn f(p: &'static i32) {  
    unsafe {  
        STASH = p;  
    }  
}
```

Cette fois, la signature de 's indique qu'il doit s'agir d'une référence avec une durée de vie , il n'y a donc plus de problème à la stocker dans . Nous ne pouvons appliquer qu'aux références à d'autres statiques, mais c'est la seule chose qui est certaine de ne pas laisser pendre de toute façon. Nous pouvons donc écrire : f p ' static STASH f STASH

```
static WORTH_POINTING_AT: i32 = 1000;
f(&WORTH_POINTING_AT);
```

Puisque est un statique, le type de est , qui est sûr de passer à . WORTH_POINTING_AT &WORTH_POINTING_AT &'static i32 f

Prenez un peu de recul, cependant, et remarquez ce qui est arrivé à la signature de ', alors que nous modifiions notre façon de corriger: l'original a fini par être . En d'autres termes, nous n'avons pas pu écrire une fonction qui cachait une référence dans une variable globale sans refléter cette intention dans la signature de la fonction. Dans Rust, la signature d'une fonction expose toujours le comportement du corps. f f(p: &i32) f(p: &'static i32)

Inversement, si nous voyons une fonction avec une signature comme (ou avec les durées de vie écrites,), nous pouvons dire qu'elle *ne* cache pas son argument nulle part qui survivra à l'appel. Il n'est pas nécessaire de se pencher sur la définition de ' ; la signature seule nous dit ce qui peut et ne peut pas faire avec son argument. Ce fait finit par être très utile lorsque vous essayez d'établir la sécurité d'un appel à la fonction. g(p: &i32) g<'a>(p: &'a i32) p g g

Transmission de références à des fonctions

Maintenant que nous avons montré comment la signature d'une fonction est liée à son corps, examinons comment elle se rapporte aux appelants de la fonction. Supposons que vous ayez le code suivant :

```
// This could be written more briefly: fn g(p: &i32),
// but let's write out the lifetimes for now.
fn g<'a>(p: &'a i32) { ... }

let x = 10;
g(&x);
```

Rien que par sa signature, Rust sait qu'il n'économisera aucun endroit qui pourrait survivre à l'appel: toute durée de vie qui entoure l'appel doit fonctionner pendant . Rust choisit donc la plus petite durée de vie possible pour : celle de l'appel à . Cela répond à toutes les contraintes : il ne survit pas, et il enferme l'ensemble de l'appel à . Donc, ce code passe le rassemblement. g p 'a &x g x g

Notez que bien que prend un paramètre de durée de vie , nous n'avons pas eu besoin de le mentionner lors de l'appel . Vous n'avez qu'à vous soucier des paramètres de durée de vie lors de la définition des fonctions et des types; lors de leur utilisation, Rust déduit les durées de vie pour vous. g 'a g

Et si nous essayions de passer à notre fonction d'avant qui stocke son argument dans un statique ? &x f

```
fn f(p: &'static i32) { ... }

let x = 10;
f(&x);
```

Cela ne se compile pas : la référence ne doit pas survivre, mais en la transmettant à , on la contraint à vivre au moins aussi longtemps que . Il n'y a aucun moyen de satisfaire tout le monde ici, alors Rust rejette le code. &x x f 'static

Renvoi de références

Il est courant qu'une fonction prenne une référence à une structure de données, puis renvoie une référence dans une partie de cette structure. Par exemple, voici une fonction qui renvoie une référence au plus petit élément d'une tranche :

```
// v should have at least one element.
fn smallest(v: &[i32]) -> &i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

Nous avons omis les durées de vie de la signature de cette fonction de la manière habituelle. Lorsqu'une fonction prend une seule référence comme argument et renvoie une seule référence, Rust suppose que les deux doivent avoir la même durée de vie. Écrire ceci explicitement nous donnerait:

```
fn smallest<'a>(v: &'a [i32]) -> &'a i32 { ... }
```

Supposons que nous appelions comme ceci: `smallest`

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // bad: points to element of dropped array
```

À partir de la signature de `smallest`, nous pouvons voir que son argument et sa valeur de retour doivent avoir la même durée de vie. Dans notre appel, l'argument ne doit pas survivre à lui-même, mais la valeur de retour de `smallest` doit vivre au moins aussi longtemps que `parabola`. Il n'y a pas de durée de vie possible qui peut satisfaire les deux contraintes, donc Rust rejette le code

```
:smallest 'a &parabola parabola smallest s 'a
```

```
error: `parabola` does not live long enough
|
11 |         s = smallest(&parabola);
|                         ----- borrow occurs here
12 |
|         }
|         ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // bad: points to element of dropped array
|                         - borrowed value needs to live until here
14 | }
```

Se déplacer pour que sa durée de vie soit clairement contenue dans `s` résout le problème: `s = parabola`

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}
```

Les durées de vie dans les signatures de fonction permettent à Rust d'évaluer les relations entre les références que vous transmettez à la fonction et celles renvoyées par la fonction, et elles garantissent qu'elles sont utilisées en toute sécurité.

Structures contenant des références

Comment Rust gère-t-il les références stockées dans les structures de données ? Voici le même programme erroné que nous avons examiné précédemment, sauf que nous avons mis la référence à l'intérieur d'une structure :

```
// This does not compile.

struct S {
    r: &i32
}

let s;
{
    let x = 10;
    s = S { r: &x };
}
assert_eq!(*s.r, 10); // bad: reads from dropped `x`
```

Les contraintes de sécurité que Rust impose aux références ne peuvent pas disparaître comme par magie simplement parce que nous avons caché la référence à l'intérieur d'une structure. D'une manière ou d'une autre, ces contraintes doivent finir par s'appliquer également. En effet, Rust est sceptique : s

```
error: missing lifetime specifier
|
7 |     r: &i32
|           ^ expected lifetime parameter
```

Chaque fois qu'un type de référence apparaît dans la définition d'un autre type, vous devez écrire sa durée de vie. Vous pouvez écrire ceci :

```
struct S {
    r: &'static i32
}
```

Cela dit que cela ne peut se référer qu'à des valeurs qui dureront pendant toute la durée de vie du programme, ce qui est plutôt limitatif. L'alternative consiste à donner au type un paramètre de durée de vie et à l'utiliser pour : r i32 'a r

```
struct S<'a> {
    r: &'a i32
```

```
}
```

Maintenant, le type a une durée de vie, tout comme les types de référence. Chaque valeur que vous créez de type obtient une nouvelle durée de vie, qui devient limitée par la façon dont vous utilisez la valeur. La durée de vie de toute référence dans laquelle vous stockez devrait être incluse, et doit durer plus longtemps que la durée de vie de l'endroit où vous stockez le . s s 'a r 'a 'a s

En revenant au code précédent, l'expression crée une nouvelle valeur avec une certaine durée de vie . Lorsque vous stockez sur le terrain, vous vous contrainez à mentir entièrement dans la vie de . s { r: &x } s 'a &x r 'a x

L'affectation stocke cela dans une variable dont la durée de vie s'étend jusqu'à la fin de l'exemple, ce qui constraint de durer plus longtemps que la durée de vie de . Et maintenant Rust est arrivé aux mêmes contraintes contradictoires qu'avant: ne doit pas survivre , mais doit vivre au moins aussi longtemps que . Aucune durée de vie satisfaisante n'existe et Rust rejette le code. Désastre évité ! s = s { ... } s 'a s 'a x s

Comment un type avec un paramètre de durée de vie se comporte-t-il lorsqu'il est placé dans un autre type ?

```
struct D {
    s: S // not adequate
}
```

Rust est sceptique, tout comme c'était le cas lorsque nous avons essayé de placer une référence sans spécifier sa durée de vie: s

```
error: missing lifetime specifier
|
8 |     s: S // not adequate
|         ^ expected named lifetime parameter
|
```

Nous ne pouvons pas laisser de côté le paramètre de durée de vie de Rust ici: Rust a besoin de savoir comment la durée de vie de la référence dans son afin d'appliquer les mêmes contrôles qu'il fait pour et les références simples. s D S D S

Nous pourrions donner la vie. Cela fonctionne : s 'static

```
struct D {
    s: S<'static>
}
```

Avec cette définition, le champ ne peut emprunter que des valeurs qui vivent pour l'ensemble de l'exécution du programme. C'est quelque peu restrictif, mais cela signifie qu'il est impossible d'emprunter une variable locale; il n'y a pas de contraintes particulières sur la durée de vie de

```
.s D D
```

Le message d'erreur de Rust suggère en fait une autre approche, qui est plus générale:

```
help: consider introducing a named lifetime parameter
|
7 | struct D<'a> {
8 |     s: S<'a>
|
```

Ici, nous donnons son propre paramètre de durée de vie et le transmettons à : D s

```
struct D<'a> {
    s: S<'a>
}
```

En prenant un paramètre de durée de vie et en l'utilisant dans le type de „, nous avons permis à Rust de relier la durée de vie de la valeur à celle de la référence qu'il contient. 'a s D S

Nous avons montré plus tôt comment la signature d'une fonction expose ce qu'elle fait avec les références que nous lui transmettons. Maintenant, nous avons montré quelque chose de similaire à propos des types: les paramètres de durée de vie d'un type révèlent toujours s'il contient des références avec des durées de vie intéressantes (c'est-à-dire non) et ce que ces durées de vie peuvent être. 'static

Par exemple, supposons que nous ayons une fonction d'analyse qui prend une tranche d'octets et renvoie une structure contenant les résultats de l'analyse :

```
fn parse_record<'i>(input: &'i [u8]) -> Record<'i> { ... }
```

Sans examiner du tout la définition du type, nous pouvons dire que, si nous recevons un de , toutes les références qu'il contient doivent pointer dans le tampon d'entrée que nous avons passé, et nulle part ailleurs (sauf peut-être à des valeurs). Record Record parse_record 'static

En fait, cette exposition du comportement interne est la raison pour laquelle Rust exige des types qui contiennent des références pour prendre des paramètres de durée de vie explicites. Il n'y a aucune raison pour que Rust ne puisse pas simplement inventer une durée de vie distincte pour chaque référence dans la structure et vous épargner la peine de les écrire. Les premières versions de Rust se comportaient en fait de cette façon, mais les développeurs ont trouvé cela déroutant: il est utile de savoir quand une valeur emprunte quelque chose à une autre valeur, en particulier lorsque vous travaillez sur des erreurs.

Il n'y a pas que les références et les types comme ça qui ont des durées de vie. Chaque type dans Rust a une durée de vie, y compris et . La plupart sont simplement , ce qui signifie que les valeurs de ces types peuvent vivre aussi longtemps que vous le souhaitez; par exemple, a est autonome et n'a pas besoin d'être abandonné avant qu'une variable particulière ne sorte de son champ d'application. Mais un type comme a une durée de vie qui doit être fermée par : il doit être abandonné tant que ses référents sont encore vivants. S i32 String 'static Vec<i32> Vec<&'a i32> 'a

Paramètres de durée de vie distincts

Supposons que vous ayez défini une structure contenant deux références comme celle-ci :

```
struct S<'a> {
    x: &'a i32,
    y: &'a i32
}
```

Les deux références utilisent la même durée de vie. Cela pourrait être un problème si votre code veut faire quelque chose comme ceci: 'a

```

let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
println!("{}" , r);

```

Ce code ne crée pas de pointeurs pendants. La référence à reste dans , qui sort du champ d'application avant. La référence à finit dans , qui ne survit pas à `x.y` `s.y` `x.r`

Si vous essayez de compiler cela, cependant, Rust se plaindra de ne pas vivre assez longtemps, même si c'est clairement le cas. Pourquoi Rust est-il inquiet? Si vous parcourez le code avec soin, vous pouvez suivre son raisonnement :

- Les deux champs de sont des références avec la même durée de vie, donc Rust doit trouver une seule vie qui fonctionne pour les deux et `.S'a` `s.x` `s.y`
- Nous attribuons , nécessitant de joindre la durée de vie de `.r = s.x` `'a r`
- Nous avons initialisé avec , ne nécessitant pas plus de la durée de vie de `.s.y` `&y` `'a y`

Ces contraintes sont impossibles à satisfaire : aucune durée de vie n'est plus courte que la portée de ' mais plus longue que celle de 's. Rust rechigne.

Le problème se pose car les deux références dans ont la même durée de vie. Changer la définition de pour permettre à chaque référence d'avoir une durée de vie distincte corrige tout: `S'a` `S'b`

```

struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}

```

Avec cette définition, et avoir des durées de vie indépendantes. Ce que nous faisons avec n'a aucun effet sur ce que nous stockons dans , il est

donc facile de satisfaire les contraintes maintenant: peut simplement être la durée de vie de ', et peut être celle de '. (la durée de vie fonctionnerait aussi pour , mais Rust essaie de choisir la plus petite durée de vie qui fonctionne.) Tout finit bien. `s.x s.y s.x s.y 'a r 'b s y 'b`

Les signatures de fonction peuvent avoir des effets similaires. Supposons que nous ayons une fonction comme celle-ci :

```
fn f<'a>(r: &'a i32, s: &'a i32) -> &'a i32 { r } // perhaps too tight
```

Ici, les deux paramètres de référence utilisent la même durée de vie, ce qui peut contraindre inutilement l'appelant de la même manière que nous l'avons montré précédemment. Si c'est un problème, vous pouvez laisser la durée de vie des paramètres varier indépendamment : 'a

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) -> &'a i32 { r } // looser
```

L'inconvénient est que l'ajout de durées de vie peut rendre les types et les signatures de fonction plus difficiles à lire. Vos auteurs ont tendance à essayer d'abord la définition la plus simple possible, puis à assouplir les restrictions jusqu'à ce que le code se compile. Étant donné que Rust ne permet pas au code de s'exécuter à moins qu'il ne soit sûr, le simple fait d'attendre d'être informé lorsqu'il y a un problème est une tactique parfaitement acceptable.

Omission des paramètres de durée de vie

Nous avons montré beaucoup de fonctions jusqu'à présent dans ce livre qui renvoient des références ou les prennent comme paramètres, mais nous n'avons généralement pas besoin d'expliquer quelle durée de vie est laquelle. Les durées de vie sont là; Rust nous permet simplement de les omettre alors qu'il est raisonnablement évident ce qu'ils devraient être.

Dans les cas les plus simples, vous n'aurez peut-être jamais besoin d'écrire des durées de vie pour vos paramètres. Rust attribue simplement une durée de vie distincte à chaque endroit qui en a besoin. Par exemple:

```
struct S<'a, 'b> {
    x: &'a i32,
    y: &'b i32
}
```

```
fn sum_r_xy(r: &i32, s: S) -> i32 {
    r + s.x + s.y
}
```

La signature de cette fonction est l'abréviation de :

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) -> i32
```

Si vous renvoyez des références ou d'autres types avec des paramètres de durée de vie, Rust essaie toujours de faciliter les cas sans ambiguïté. S'il n'y a qu'une seule durée de vie qui apparaît parmi les paramètres de votre fonction, Rust suppose que toutes les durées de vie de votre valeur de retour doivent être celles-ci :

```
fn first_third(point: &[i32; 3]) -> (&i32, &i32) {
    (&point[0], &point[2])
}
```

Avec toutes les durées de vie écrites, l'équivalent serait:

```
fn first_third<'a>(point: &'a [i32; 3]) -> (&'a i32, &'a i32)
```

S'il y a plusieurs durées de vie parmi vos paramètres, il n'y a aucune raison naturelle de préférer l'une à l'autre pour la valeur de retour, et Rust vous fait expliquer ce qui se passe.

Si votre fonction est une méthode sur un type et prend son paramètre par référence, alors cela brise le lien: Rust suppose que la durée de vie de est celle qui donne tout dans votre valeur de retour. (Un paramètre fait référence à la valeur que la méthode est appelée sur l'équivalent de Rust en C++, Java ou JavaScript, ou en Python. Nous couvrirons les méthodes dans [« Définition des méthodes avec impl »](#).) self self self this self

Par exemple, vous pouvez écrire ce qui suit :

```
struct StringTable {
    elements: Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
```

```

        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}

```

La signature de la méthode est un raccourci pour : `find_by_prefix`

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) -> Option<&'a Str
```

Rust suppose que quoi que vous empruntiez, vous empruntez à `.self`

Encore une fois, ce ne sont que des abréviations, destinées à être utiles sans introduire de surprises. Quand ils ne sont pas ce que vous voulez, vous pouvez toujours écrire les durées de vie explicitement.

Partage versus mutation

Jusqu'à présent, nous avons discuté de la façon dont Rust garantit qu'aucune référence ne pointera jamais vers une variable qui est sortie de son champ d'application. Mais il existe d'autres façons d'introduire des pointeurs pendants. Voici un cas simple:

```

let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v; // move vector to aside
r[0];          // bad: uses `v`, which is now uninitialized

```

Affectation pour déplacer le vecteur, en laissant non initialisé, et se transforme en pointeur pendant, comme illustré à [la figure 5-7](#). `aside v r`

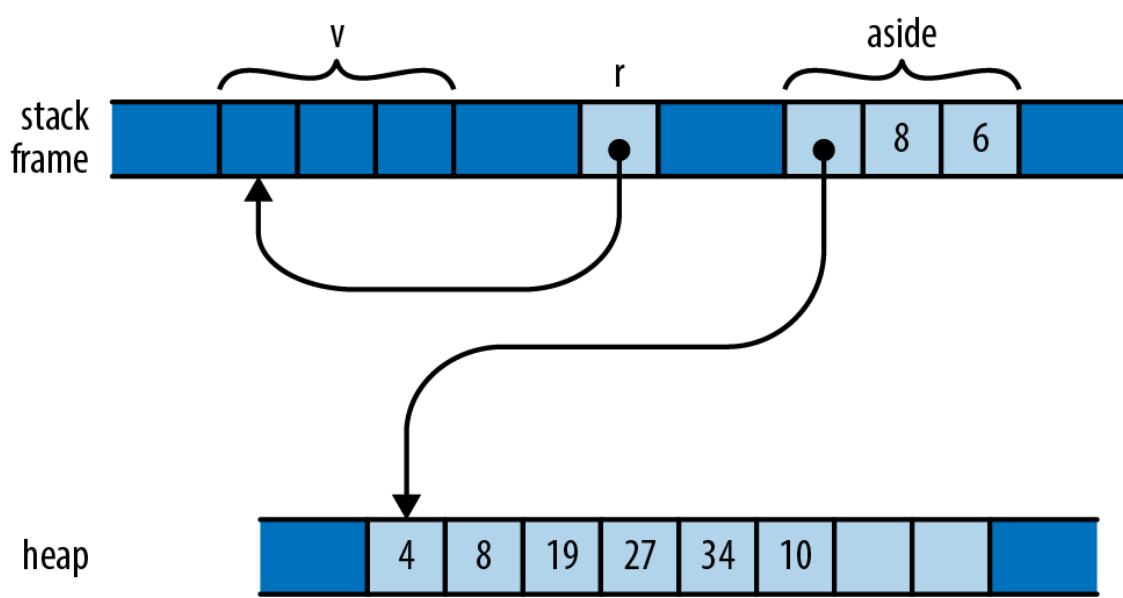


Figure 5-7. Référence à un vecteur qui a été déplacé

Bien qu'elle reste dans la portée de toute la vie, le problème ici est que la valeur de 's'déplace ailleurs, laissant non initialisée tout en s'y référant. Naturellement, Rust attrape l'erreur: `v r v v r`

```
error: cannot move out of `v` because it is borrowed
|
9 |     let r = &v;
|             - borrow of `v` occurs here
10 |     let aside = v; // move vector to aside
|             ^^^^^^ move out of `v` occurs here
```

Tout au long de sa durée de vie, une référence partagée rend son référent en lecture seule : vous ne pouvez pas l'affecter au référent ou déplacer sa valeur ailleurs. Dans ce code, la durée de vie de contient la tentative de déplacement du vecteur, de sorte que Rust rejette le programme. Si vous modifiez le programme comme indiqué ici, il n'y a pas de problème: `r`

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];           // ok: vector is still there
}
let aside = v;
```

Dans cette version, sort du champ d'application plus tôt, la durée de vie de la référence se termine avant d'être déplacée, et tout va bien. `r v`

Voici une façon différente de faire des ravages. Supposons que nous ayons une fonction pratique pour étendre un vecteur avec les éléments d'une tranche :

```

fn extend(vec: &mut Vec<f64>, slice: &[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}

```

Il s'agit d'une version moins flexible (et beaucoup moins optimisée) de la méthode de la bibliothèque standard sur les vecteurs. Nous pouvons l'utiliser pour construire un vecteur à partir de tranches d'autres vecteurs ou tableaux: `extend_from_slice`

```

let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];

extend(&mut wave, &head);      // extend wave with another vector
extend(&mut wave, &tail);      // extend wave with an array

assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);

```

Nous avons donc construit une période d'onde sinusoïdale ici. Si nous voulons ajouter une autre ondulation, pouvons-nous ajouter le vecteur à lui-même?

```

extend(&mut wave, &wave);
assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                      0.0, 1.0, 0.0, -1.0]);

```

Cela peut sembler bien lors d'une inspection occasionnelle. Mais rappelez-vous que lorsque nous ajoutons un élément à un vecteur, si son tampon est plein, il doit allouer un nouveau tampon avec plus d'espace. Supposons qu'il commence par de l'espace pour quatre éléments et qu'il faille donc allouer un tampon plus grand lorsque vous essayez d'en ajouter un cinquième. La mémoire finit par ressembler à [la Figure 5-8.](#)

wave extend

L'argument de la fonction emprunte (propriété de l'appelant), qui s'est alloué un nouveau tampon avec de l'espace pour huit éléments. Mais continue de pointer vers l'ancien tampon à quatre éléments, qui a été abandonné. `extend vec wave slice`

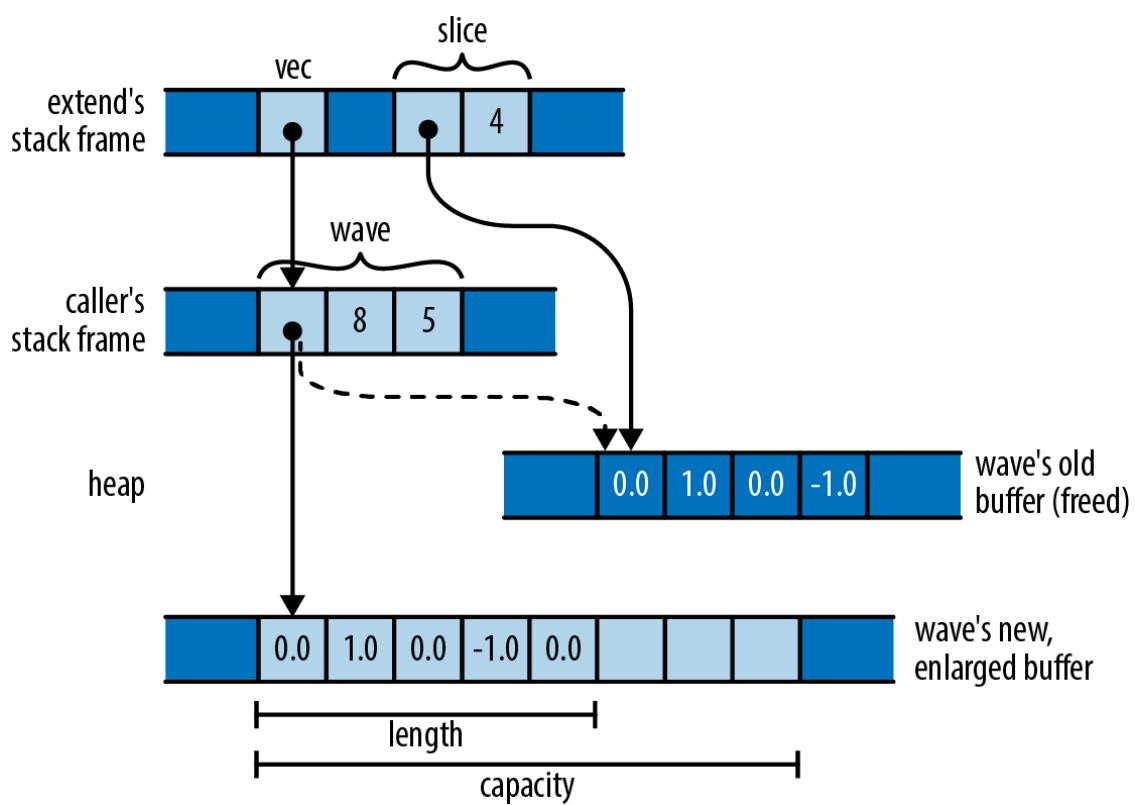


Figure 5-8. Une tranche transformée en pointeur pendant par une réaffectation vectorielle

Ce genre de problème n'est pas propre à Rust : modifier les collections tout en pointant vers elles est un territoire délicat dans de nombreuses langues. En C++, la spécification vous avertit que « la réallocation [de la mémoire tampon du vecteur] invalide toutes les références, pointeurs et itérateurs faisant référence aux éléments de la séquence ». De même, dit Java, de la modification d'un objet

```
: std::vector java.util.Hashtable
```

Si la table de hachage est structurellement modifiée à tout moment après la création de l'itérateur, de quelque manière que ce soit, sauf via la propre méthode remove de l'itérateur, l'itérateur lèvera une ConcurrentModificationException.

Ce qui est particulièrement difficile à propos de ce genre de bug, c'est que cela n'arrive pas tout le temps. Lors des tests, votre vecteur peut toujours avoir suffisamment d'espace, le tampon peut ne jamais être réaffecté et le problème peut ne jamais apparaître.

Rust, cependant, signale le problème avec notre appel à au moment de la compilation: `extend`

```
error: cannot borrow `wave` as immutable because it is also
      borrowed as mutable
|
9 |     extend(&mut wave, &wave);
|           ----- ^^^^-- mutable borrow ends here
```

```
|           |           |
|           |           immutable borrow occurs here
|           |           mutable borrow occurs here
```

En d'autres termes, nous pouvons emprunter une référence mutable au vecteur, et nous pouvons emprunter une référence partagée à ses éléments, mais les durées de vie de ces deux références ne doivent pas se chevaucher. Dans notre cas, les durées de vie des deux références contiennent l'appel à , donc Rust rejette le code. `extend`

Ces erreurs proviennent toutes deux de violations des règles de Rust en matière de mutation et de partage :

L'accès partagé est un accès en lecture seule.

Les valeurs empruntées par des références partagées sont en lecture seule. Tout au long de la durée de vie d'une référence partagée, ni son référent, ni rien d'accessible à partir de ce référent, ne peut être modifié *par quoi que ce soit*. Il n'existe aucune référence mutable en direct à quoi que ce soit dans cette structure, son propriétaire est tenu en lecture seule, et ainsi de suite. C'est vraiment gelé.

L'accès mutable est un accès exclusif.

Une valeur empruntée par une référence modifiable est accessible exclusivement via cette référence. Tout au long de la durée de vie d'une référence mutable, il n'y a pas d'autre chemin utilisable vers son référent ou vers une valeur accessible à partir de là. Les seules références dont les durées de vie peuvent chevaucher une référence mutable sont celles que vous empruntez à la référence mutable elle-même.

Rust a rapporté l'exemple comme une violation de la deuxième règle: puisque nous avons emprunté une référence mutable à , cette référence mutable doit être le seul moyen d'atteindre le vecteur ou ses éléments. La référence partagée à la tranche est elle-même un autre moyen d'atteindre les éléments, violant la deuxième règle. `extend wave`

Mais Rust aurait également pu traiter notre bug comme une violation de la première règle : puisque nous avons emprunté une référence partagée aux éléments de ', les éléments et le lui-même sont tous en lecture seule. Vous ne pouvez pas emprunter une référence modifiable à une valeur en lecture seule. `wave Vec`

Chaque type de référence affecte ce que nous pouvons faire avec les valeurs le long du chemin de propriété vers le référent, et les valeurs ac-

cessibles à partir du référentiel ([Figure 5-9](#)).

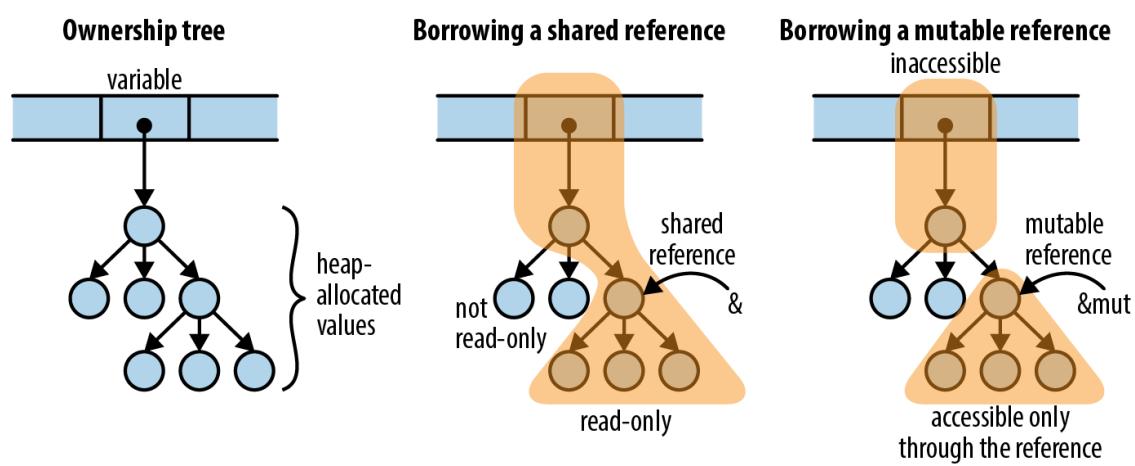


Figure 5-9. L'emprunt d'une référence affecte ce que vous pouvez faire avec d'autres valeurs dans le même arbre de propriété

Notez que dans les deux cas, le chemin de propriété menant au référentiel ne peut pas être modifié pendant la durée de vie de la référence. Pour un emprunt partagé, le chemin d'accès est en lecture seule ; pour un emprunt mutable, c'est complètement inaccessible. Il n'y a donc aucun moyen pour le programme de faire quoi que ce soit qui invaliderait la référence.

En réduisant ces principes aux exemples les plus simples possibles :

```

let mut x = 10;
let r1 = &x;
let r2 = &x;      // ok: multiple shared borrows permitted
x += 10;          // error: cannot assign to `x` because it is borrowed
let m = &mut x;  // error: cannot borrow `x` as mutable because it is
                 // also borrowed as immutable
println!("{}, {}, {}", r1, r2, m); // the references are used here,
                                   // so their lifetimes must last
                                   // at least this long

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y; // error: cannot borrow as mutable more than once
let z = y;        // error: cannot use `y` because it was mutably borr
println!("{}, {}, {}", m1, m2, z); // references are used here

```

Il est acceptable de réoutiller une référence partagée à partir d'une référence partagée :

```

let mut w = (107, 109);
let r = &w;
let r0 = &r.0;      // ok: reborrowing shared as shared

```

```
let m1 = &mut r.1;      // error: can't reborrow shared as mutable
println!("{}", r0);    // r0 gets used here
```

Vous pouvez réemprunter à partir d'une référence modifiable :

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // ok: reborrowing mutable from mutable
*m0 = 137;
let r1 = &m.1;          // ok: reborrowing shared from mutable,
                        // and doesn't overlap with m0
v.1;                  // error: access through other paths still for
println!("{}", r1);    // r1 gets used here
```

Ces restrictions sont assez strictes. Pour en revenir à notre tentative d'appel, il n'y a pas de moyen rapide et facile de réparer le code pour qu'il fonctionne comme nous le voudrions. Et Rust applique ces règles partout : si nous empruntons, disons, une référence partagée à une clé dans un , nous ne pouvons pas emprunter une référence mutable à la jusqu'à la fin de la durée de vie de la référence partagée.

```
extend(&mut wave,
&wave) HashMap
```

Mais il y a une bonne justification à cela : concevoir des collections pour prendre en charge l'itération et la modification simultanées sans restriction est difficile et empêche souvent des implémentations plus simples et plus efficaces. Java et C++ ne dérangent pas, et ni les dictionnaires Python ni les objets JavaScript ne définissent exactement comment un tel accès se comporte. D'autres types de collection en JavaScript le font, mais nécessitent des implémentations plus lourdes en conséquence. C++ promet que l'insertion de nouvelles entrées n'invalider pas les pointeurs vers d'autres entrées de la carte, mais en faisant cette promesse, la norme empêche des conceptions plus efficaces en cache comme celle de Rust, qui stocke plusieurs entrées dans chaque nœud de l'arborescence.

```
Hashtable vector std::map BTreeMap
```

Voici un autre exemple du type de bug détecté par ces règles. Considérez le code C++ suivant, destiné à gérer un descripteur de fichier. Pour simplifier les choses, nous n'allons afficher qu'un constructeur et un opérateur d'affectation de copie, et nous allons omettre la gestion des erreurs :

```
struct File {
    int descriptor;
```

```

File(int d) : descriptor(d) { }

File& operator=(const File &rhs) {
    close(descriptor);
    descriptor = dup(rhs.descriptor);
    return *this;
}
};

```

L'opérateur d'affectation est assez simple, mais échoue mal dans une situation comme celle-ci :

```

File f(open("foo.txt", ...));
...
f = f;

```

Si nous attribuons un à lui-même, les deux et sont le même objet, ferme donc le descripteur de fichier même qu'il est sur le point de passer à .

Nous détruisons la même ressource que nous étions censés copier. `File rhs *this operator= dup`

Dans Rust, le code analogue serait :

```

struct File {
    descriptor: i32
}

fn new_file(d: i32) -> File {
    File { descriptor: d }
}

fn clone_from(this: &mut File, rhs: &File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}

```

(Ce n'est pas de la rouille idiomatique. Il existe d'excellents moyens de donner aux types Rust leurs propres fonctions et méthodes de constructeur, que nous décrivons au [chapitre 9](#), mais les définitions précédentes fonctionnent pour cet exemple.)

Si nous écrivons le code Rust correspondant à l'utilisation de , nous obtenons: `File`

```
let mut f = new_file(open("foo.txt", ...));  
...  
clone_from(&mut f, &f);
```

Rust, bien sûr, refuse même de compiler ce code:

```
error: cannot borrow `f` as immutable because it is also  
borrowed as mutable  
|  
18 |     clone_from(&mut f, &f);  
|             -      ^- mutable borrow ends here  
|             |      |  
|             |      immutable borrow occurs here  
|             |      mutable borrow occurs here
```

Cela devrait vous sembler familier. Il s'avère que deux bogues C++ classiques – l'incapacité à faire face à l'auto-affectation et l'utilisation d'itérateurs invalidés – sont le même type de bogue sous-jacent ! Dans les deux cas, le code suppose qu'il modifie une valeur tout en en consultant une autre, alors qu'en fait, ils ont tous les deux la même valeur. Si vous avez accidentellement laissé la source et la destination d'un appel vers ou se chevaucher en C ou C++, c'est encore une autre forme que le bogue peut prendre. En exigeant que l'accès mutable soit exclusif, Rust a repoussé une large classe d'erreurs quotidiennes. `memcpy` `strcpy`

L'immiscibilité des références partagées et mutables démontre vraiment sa valeur lors de l'écriture de code simultané. Une course de données n'est possible que lorsqu'une valeur est à la fois mutable et partagée entre les threads, ce qui est exactement ce que les règles de référence de Rust éliminent. Un programme Rust simultané qui évite le code est exempt de courses de données *par construction*. Nous couvrirons cet aspect plus en détail lorsque nous parlerons de la concurrence dans le [chapitre 19](#), mais en résumé, la concurrence est beaucoup plus facile à utiliser dans Rust que dans la plupart des autres langues. `unsafe`

À première vue, les références partagées de Rust semblent ressembler étroitement aux pointeurs de valeurs de C et C++. Cependant, les règles de Rust pour les références partagées sont beaucoup plus strictes. Par exemple, considérez le code C suivant : const

```
int x = 42;           // int variable, not const
const int *p = &x;    // pointer to const int
assert(*p == 42);
x++;                // change variable directly
assert(*p == 43);   // "constant" referent's value has changed
```

Le fait que ce soit un moyen que vous ne puissiez pas modifier son référent via lui-même est interdit. Mais vous pouvez également vous attaquer directement au référent en tant que , ce qui n'est pas , et changer sa valeur de cette façon. Le mot-clé de la famille C a ses utilisations, mais il n'est pas constant. p const int * p (*p)++ x const const

Dans Rust, une référence partagée interdit toute modification de son référent, jusqu'à la fin de sa durée de vie :

```
let mut x = 42;          // non-const i32 variable
let p = &x;              // shared reference to i32
assert_eq!(*p, 42);
x += 1;                // error: cannot assign to x because it is bor
assert_eq!(*p, 42);    // if you take out the assignment, this is tru
```

Pour nous assurer qu'une valeur est constante, nous devons garder une trace de tous les chemins possibles vers cette valeur et nous assurer qu'ils ne permettent pas de modification ou ne peuvent pas être utilisés du tout. Les pointeurs C et C++ sont trop illimités pour que le compilateur puisse vérifier cela. Les références de Rust sont toujours liées à une durée de vie particulière, ce qui permet de les vérifier au moment de la compilation.

Prendre les armes contre une mer d'objets

Depuis l'essor de la gestion automatique de la mémoire dans les années 1990, l'architecture par défaut de tous les programmes a été la *mer d'objets*, illustrée à la [figure 5-10](#).

C'est ce qui se passe si vous avez un garbage collection et que vous commencez à écrire un programme sans rien concevoir. Nous avons tous construit des systèmes qui ressemblent à ceci.

Cette architecture présente de nombreux avantages qui n'apparaissent pas dans le diagramme : la progression initiale est rapide, il est facile de pirater des choses, et quelques années plus tard, vous n'aurez aucune difficulté à justifier une réécriture complète. (Cue AC/DC 'S « Highway to Hell. »)

Bien sûr, il y a aussi des inconvénients. Lorsque tout dépend de tout le reste comme celui-ci, il est difficile de tester, d'évoluer ou même de penser à un composant isolément.

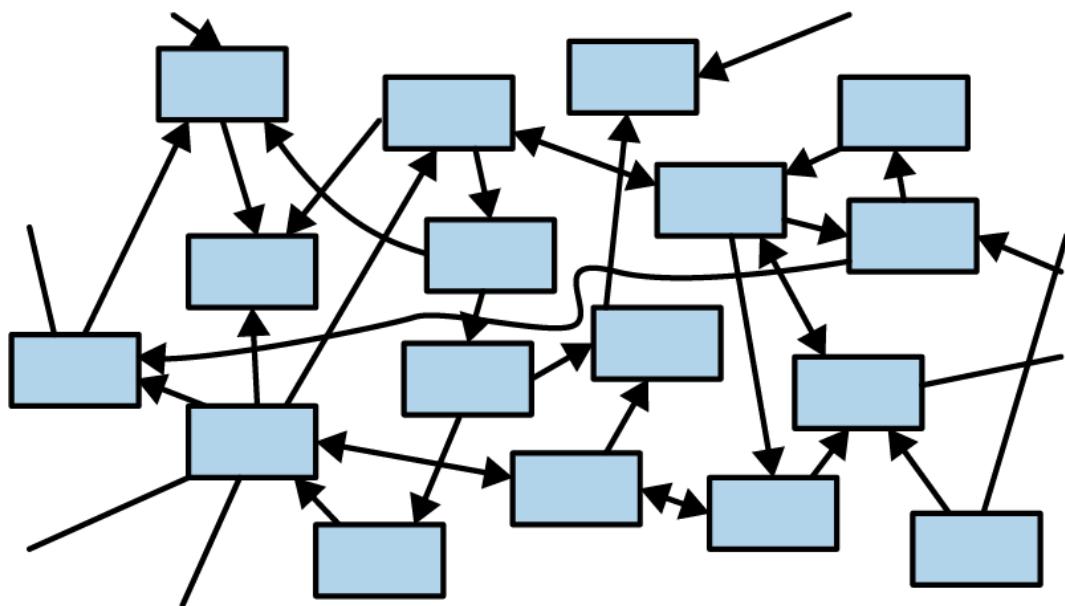


Figure 5-10. Une mer d'objets

Une chose fascinante à propos de Rust est que le modèle de propriété met un ralentisseur sur l'autoroute en enfer. Il faut un peu d'effort pour faire un cycle dans Rust, deux valeurs telles que chacune contient une référence pointant vers l'autre. Vous devez utiliser un type de pointeur intelligent, tel que , et [la mutabilité intérieure](#), un sujet que nous n'avons même pas encore abordé. Rust préfère que les pointeurs, la propriété et le flux de données traversent le système dans une direction, comme illustré à [la figure 5-11](#). Rc

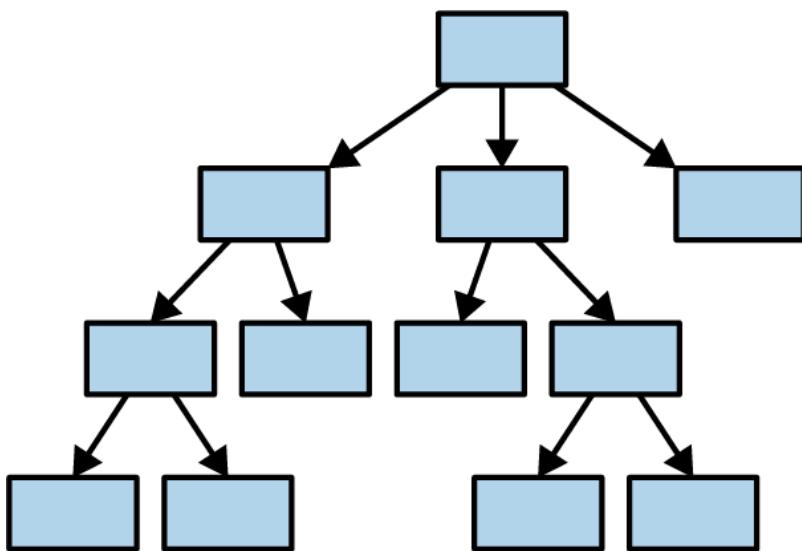


Figure 5-11. Un arbre de valeurs

La raison pour laquelle nous soulevons cette question en ce moment est qu'il serait naturel, après avoir lu ce chapitre, de vouloir courir tout de suite et de créer une « mer de structs », le tout lié avec des pointeurs intelligents, et de recréer tous les antimodèles orientés objet que vous connaissez. Cela ne fonctionnera pas pour vous tout de suite. Le modèle de propriété de Rust vous donnera quelques problèmes. Le remède consiste à faire une conception initiale et à élaborer un meilleur programme. Rc

Rust consiste à transférer la douleur de la compréhension de votre programme du futur au présent. Cela fonctionne déraisonnablement bien: non seulement Rust peut vous forcer à comprendre pourquoi votre programme est thread-safe, mais il peut même nécessiter une certaine quantité de conception architecturale de haut niveau.

Chapitre 6. Expressions

Les programmeurs LISP connaissent la valeur de tout, mais le coût de rien.

—Alan Perlis, épigramme #55

Dans ce chapitre, nous couvrirons les expressions de Rust, les *blocs de construction* qui composent le corps des fonctions Rust et donc la majorité du code Rust. La plupart des choses dans Rust sont des expressions. Dans ce chapitre, nous explorerons la puissance que cela apporte et comment travailler avec ses limites. Nous couvrirons le flux de contrôle, qui dans Rust est entièrement axé sur l'expression, et comment les opérateurs fondamentaux de Rust fonctionnent de manière isolée et combinée.

Quelques concepts qui entrent techniquement dans cette catégorie, tels que les fermetures et les itérateurs, sont suffisamment profonds pour que nous leur consacrons un chapitre entier plus tard. Pour l'instant, nous visons à couvrir autant de syntaxe que possible en quelques pages.

Un langage d'expression

Rust ressemble visuellement à la famille de langages C, mais c'est un peu une ruse. En C, il y a une nette distinction entre les *expressions*, les bits de code qui ressemblent à ceci :

```
5 * (fahr-32) / 9
```

et des *déclarations*, qui ressemblent plus à ceci :

```
for (; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

Les expressions ont des valeurs. Les déclarations ne le font pas.

La rouille est ce qu'on appelle un *langage d'expression*. Cela signifie qu'il suit une tradition plus ancienne, remontant à Lisp, où les expressions font tout le travail.

En C, et sont des déclarations. Ils ne produisent pas de valeur et ne peuvent pas être utilisés au milieu d'une expression. Dans Rust, et peut produire des valeurs. Nous avons déjà vu une expression qui produit une valeur numérique au [chapitre 2](#): `if switch if match match`

```

pixels[r * bounds.0 + c] =
    match escapes(Complex { re: point.0, im: point.1 }, 255) {
        None => 0,
        Some(count) => 255 - count as u8
    };

```

Une expression peut être utilisée pour initialiser une variable : if

```

let status =
    if cpu.temperature <= MAX_TEMP {
        HttpStatus::Ok
    } else {
        HttpStatus::ServerError // server melted
    };

```

Une expression peut être passée en tant qu'argument à une fonction ou une macro : match

```

println!("Inside the vat, you see {}.",  

    match vat.contents {  

        Some(brain) => brain.desc(),  

        None => "nothing of interest"  

    });

```

Cela explique pourquoi Rust n'a pas l'opérateur ternaire de C () . En C, il s'agit d'un analogue pratique au niveau de l'expression de l'instruction. Ce serait redondant dans Rust : l'expression gère les deux cas. expr1 ? expr2 : expr3 if if

La plupart des outils de flux de contrôle en C sont des instructions. Dans Rust, ce sont toutes des expressions.

Priorité et associativité

[Le tableau 6-1](#) résume la syntaxe de l'expression Rust. Nous discuterons de tous ces types d'expressions dans ce chapitre. Les opérateurs sont répertoriés par ordre de priorité, du plus élevé au plus bas. (Comme la plupart des langages de programmation, Rust a *la priorité des opérateurs* pour déterminer l'ordre des opérations lorsqu'une expression contient plusieurs opérateurs adjacents. Par exemple, dans , l'opérateur a la priorité la plus élevée, de sorte que l'accès au champ se produit en premier.) limit < 2 * broom.size + 1 .

Tableau 6-1. Expressions

Type d'expression	Exemple	Traits apparentés
Littéral de tableau	[1, 2, 3]	
Répéter le littéral de tableau	[0; 50]	
Tuple	(6, "crullers")	
Groupement	(2 + 2)	
Bloquer	{ f(); g() }	
Contrôler les expressions de flux	<pre>if ok { f() } if ok { 1 } else { 0 } if let Some(x) = f() { x } else { 0 } match x { None => 0, => 1 }</pre>	
	<pre>for v in e { f(v); }</pre>	<u>std::iterator</u>
	<pre>while ok { ok = f(); }</pre>	
	<pre>while let Some(x) = i t.next() { f(x); }</pre>	
	<pre>loop { next_event(); }</pre>	
	break	
	continue	

Type d'expression	Exemple	Traits apparentés
	return 0	
Appel de macro	println!("ok")	
Chemin	std::f64::consts::PI	
Littéral Struct	Point {x: 0, y: 0}	
Accès au champ	pair.0	<u>Deref</u> , <u>DerefM</u>
Tuple		<u>ut</u>
Accès au champ	point.x	<u>Deref</u> , <u>DerefM</u>
Struct		<u>ut</u>
Appel de méthode	point.translate(50, 5 0)	<u>Deref</u> , <u>DerefM</u> <u>ut</u>
Appel de fonction	stdin()	<u>Fn(Arg0, ...)</u> -> T, <u>FnMut(Arg0, ...)</u> -> T, <u>FnOnce(Arg0, ...)</u> -> T
Index	arr[0]	<u>Index</u> , <u>IndexMut</u> , <u>Deref</u> , <u>DerefM</u> <u>ut</u>
Vérification des erreurs	create_dir("tmp")?	
Logique/bitwise NOT	!ok	<u>Not</u>
Négation	-num	<u>Neg</u>
Déréférencement	*ptr	<u>Deref</u> , <u>DerefM</u> <u>ut</u>

Type d'expression	Exemple	Traits apparentés
Emprunter	<code>&val</code>	
Type coulé	<code>x as u32</code>	
Multiplication	<code>n * 2</code>	<u>Mul</u>
Division	<code>n / 2</code>	<u>Div</u>
Reste (module)	<code>n % 2</code>	<u>Rem</u>
Addition	<code>n + 1</code>	<u>Add</u>
Soustraction	<code>n - 1</code>	<u>Sub</u>
Décalage à gauche	<code>n << 1</code>	<u>Shl</u>
Décalage à droite	<code>n >> 1</code>	<u>Shr</u>
Bitwise ET	<code>n & 1</code>	<u>BitAnd</u>
Bitwise exclusif OU	<code>n ^ 1</code>	<u>BitXor</u>
Binaire OU	<code>n 1</code>	<u>BitOr</u>
Moins de	<code>n < 1</code>	<u>std::cmp::PartialOrd</u>
Inférieur ou égal à	<code>n <= 1</code>	<u>std::cmp::PartialOrd</u>
Plus grand que	<code>n > 1</code>	<u>std::cmp::PartialOrd</u>
Supérieur ou égal à	<code>n >= 1</code>	<u>std::cmp::PartialOrd</u>

Type d'expression	Exemple	Traits apparentés
Égal	<code>n == 1</code>	<u>std::cmp::PartialEq</u>
Pas égal	<code>n != 1</code>	<u>std::cmp::PartialEq</u>
Logique ET	<code>x.ok && y.ok</code>	
Logique OU	<code>x.ok backup.ok</code>	
Gamme exclusive finale	<code>start .. stop</code>	
Gamme end-inclusive	<code>start ..= stop</code>	
Mission	<code>x = val</code>	
Affectation composée	<code>x *= 1</code> <code>x /= 1</code> <code>x %= 1</code> <code>x += 1</code> <code>x -= 1</code> <code>x <= 1</code> <code>x >= 1</code> <code>x &= 1</code> <code>x ^= 1</code> <code>x = 1</code>	<u>MulAssign</u> <u>DivAssign</u> <u>RemAssign</u> <u>AddAssign</u> <u>SubAssign</u> <u>ShlAssign</u> <u>ShrAssign</u> <u>BitAndAssign</u> <u>BitXorAssign</u> <u>BitOrAssign</u>
Fermeture	<code> x, y x + y</code>	

Tous les opérateurs qui peuvent être enchaînés utilement sont associatifs à gauche. C'est-à-dire une chaîne d'opérations telle que est regroupée comme , et non . Les opérateurs qui peuvent être enchaînés de cette manière sont tous ceux auxquels vous pouvez vous attendre: a - b - c (a - b) - c a - (b - c)

* / % + - << >> & ^ | && || as

Les opérateurs de comparaison, les opérateurs d'affectation et les opérateurs de plage ne peuvent pas être enchaînés du tout. . . . =

Blocs et points-virgules

Les blocs sont le type d'expression le plus général. Un bloc produit une valeur et peut être utilisé partout où une valeur est nécessaire :

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

Le code après est l'expression simple . Le code après est une expression de bloc. Cela ne fait aucune différence pour Rust. La valeur du bloc est la valeur de sa dernière expression, . Some(author)
=> author.name() None => ip.to_string()

Notez qu'il n'y a pas de point-virgule après l'appel de méthode. La plupart des lignes de code Rust se terminent par un point-virgule ou des accolades, tout comme C ou Java. Et si un bloc ressemble à du code C, avec des points-virgules dans tous les endroits familiers, alors il fonctionnera comme un bloc C, et sa valeur sera . Comme nous l'avons mentionné au [chapitre 2](#), lorsque vous laissez le point-virgule sur la dernière ligne d'un bloc, cela fait de la valeur du bloc la valeur de son expression finale, plutôt que la valeur habituelle . ip.to_string() () ()

Dans certains langages, en particulier JavaScript, vous êtes autorisé à omettre des points-virgules, et le langage les remplit simplement pour vous, ce qui est une commodité mineure. C'est différent. Dans Rust, le point-virgule signifie en fait quelque chose :

```
let msg = {
    // let-declaration: semicolon is always required
```

```

let dandelion_control = puffball.open();

// expression + semicolon: method is called, return value dropped
dandelion_control.release_all_seeds(launch_codes);

// expression with no semicolon: method is called,
// return value stored in `msg`
dandelion_control.get_status()

};

```

Cette capacité des blocs à contenir des déclarations et à produire une valeur à la fin est une caractéristique intéressante, qui semble rapidement naturelle. Le seul inconvénient est que cela conduit à un message d'erreur étrange lorsque vous omettez un point-virgule par accident:

```

...
if preferences.changed() {
    page.compute_size() // oops, missing semicolon
}
...

```

Si vous avez fait cette erreur dans un programme C ou Java, le compilateur vous signalera simplement qu'il vous manque un point-virgule. Voici ce que dit Rust :

```

error: mismatched types
22 |     page.compute_size() // oops, missing semicolon
|     ^^^^^^^^^^^^^^^^^^- help: try adding a semicolon: `;`  

|     |
|     expected (), found tuple
|
= note: expected unit type `()`  

      found tuple `(u32, u32)`

```

Avec le point-virgule manquant, la valeur du bloc serait ce qui retourne, mais un sans un doit toujours renvoyer . Heureusement, Rust a déjà vu ce genre de chose et suggère d'ajouter le point-virgule. `page.compute_size() if else ()`

Déclarations

Outre les expressions et les points-virgules, un bloc peut contenir un nombre illimité de déclarations. Les plus courantes sont les déclarations, qui déclarent des variables locales : `let`

```
let name: type = expr;
```

Le type et l'initialiseur sont facultatifs. Le point-virgule est obligatoire. Comme tous les identificateurs dans Rust, les noms de variables doivent commencer par une lettre ou un trait de soulignement et ne peuvent contenir des chiffres qu'après ce premier caractère. Rust a une définition large de « lettre »: il comprend des lettres grecques, des caractères latins accentués et bien d'autres symboles - tout ce que l'annexe standard Unicode n ° 31 déclare approprié. Les emoji ne sont pas autorisés.

Une déclaration peut déclarer une variable sans l'initialiser. La variable peut ensuite être initialisée avec une affectation ultérieure. Ceci est parfois utile, car parfois une variable doit être initialisée à partir du milieu d'une sorte de construction de flux de contrôle : `let`

```
let name;
if user.hasNickname() {
    name = user.nickname();
} else {
    name = generateUniqueName();
    user.register(&name);
}
```

Ici, il y a deux façons différentes d'initialiser la variable locale, mais de toute façon, elle sera initialisée exactement une fois, il n'est donc pas nécessaire de la déclarer `.name name mut`

C'est une erreur d'utiliser une variable avant qu'elle ne soit initialisée. (Ceci est étroitement lié à l'erreur d'utilisation d'une valeur après son déplacement. Rust veut vraiment que vous n'utilisiez des valeurs que tant qu'elles existent!)

Vous pouvez parfois voir du code qui semble redéclarer une variable existante, comme ceci :

```
for line in file.lines() {
    let line = line?;
    ...
}
```

La déclaration crée une nouvelle deuxième variable d'un type différent. Le type de la première variable est `.file`. Le second est un fichier. Sa définition remplace celle du premier pour le reste du bloc. C'est ce qu'on appelle *l'ombrage* et c'est très courant dans les programmes Rust. Le code est équivalent à : `let line Result<String, io::Error> line String`

```
for line_result in file.lines() {
    let line = line_result?;
```

```
    ...  
}
```

Dans ce livre, nous nous en tiendrons à l'utilisation d'un suffixe dans de telles situations afin que les variables aient des noms distincts. `_result`

Un bloc peut également contenir des *déclarations d'élément*. Un élément est simplement une déclaration qui pourrait apparaître globalement dans un programme ou un module, tel qu'un `, , ou . fn struct use`

Les chapitres suivants couvriront les points en détail. Pour l'instant, c'est un exemple suffisant. Tout bloc peut contenir un : `fn fn`

```
use std::io;  
use std::cmp::Ordering;  
  
fn show_files() -> io::Result<()> {  
    let mut v = vec![];  
    ...  
  
    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering  
        a.timestamp.cmp(&b.timestamp) // first, compare timestamps  
            .reverse() // newest file first  
            .then(a.path.cmp(&b.path)) // compare paths to break ties  
    }  
  
    v.sort_by(cmp_by_timestamp_then_name);  
    ...  
}
```

Lorsqu'un est déclaré à l'intérieur d'un bloc, son étendue est le bloc entier, c'est-à-dire qu'il peut être *utilisé* dans tout le bloc englobant. Mais un imbriqué ne peut pas accéder aux variables locales ou aux arguments qui se trouvent être dans la portée. Par exemple, la fonction ne pouvait pas être utilisée directement. (Rust a également des fermetures, qui se voient dans des étendues englobantes. Voir [le chapitre 14.](#))

`fn fn cmp_by_timestamp_then_name v`

Un bloc peut même contenir un module entier. Cela peut sembler un peu beaucoup – avons-nous vraiment besoin d'être en mesure d'imbriquer *chaque* morceau du langage dans tous les autres morceaux ? – mais les programmeurs (et en particulier les programmeurs utilisant des macros) ont un moyen de trouver des utilisations pour chaque morceau d'orthogonalité fourni par le langage.

si et correspondre

La forme d'une expression est familière : if

```
if condition1 {  
    block1  
} else if condition2 {  
    block2  
} else {  
    block_n  
}
```

Chacun doit être une expression de type ; true à la forme, Rust ne convertit pas implicitement les nombres ou les pointeurs en valeurs booléennes. condition bool

Contrairement à C, les parenthèses ne sont pas nécessaires autour des conditions. En fait, émettra un avertissement si des parenthèses inutiles sont présentes. Les accolades, cependant, sont nécessaires. rustc

Les blocs, ainsi que la finale, sont facultatifs. Une expression sans bloc se comporte exactement comme si elle avait un bloc vide. else
if else if else else

match les expressions sont quelque chose comme l'instruction C, mais plus flexibles. Un exemple simple : switch

```
match code {  
    0 => println!("OK"),  
    1 => println!("Wires Tangled"),  
    2 => println!("User Asleep"),  
    _ => println!("Unrecognized Error {}", code)  
}
```

C'est quelque chose qu'une déclaration pourrait faire. Exactement un des quatre bras de cette expression s'exécuera, en fonction de la valeur de . Le modèle générique correspond à tout. C'est comme le cas dans une déclaration, sauf qu'elle doit venir en dernier; placer un modèle avant les autres modèles signifie qu'il aura préséance sur eux. Ces modèles ne correspondront jamais à rien (et le compilateur vous en avertira). switch match code _ default: switch _

Le compilateur peut optimiser ce type d'utilisation d'une table de saut, tout comme une instruction en C++. Une optimisation similaire est appliquée lorsque chaque bras d'un produit une valeur constante. Dans ce cas, le compilateur génère un tableau de ces valeurs et le est compilé dans un accès au tableau. En dehors d'une vérification des limites, il n'y a aucun branchement dans le code compilé. match switch match match

La polyvalence des provient de la variété de *motifs soutenus* qui peuvent être utilisés à gauche de dans chaque bras. Ci-dessus, chaque motif est simplement un entier constant. Nous avons également montré des expressions qui distinguent les deux types de valeur

```
:match => match Option
```

```
match params.get("name") {
    Some(name) => println!("Hello, {}!", name),
    None => println!("Greetings, stranger.")
}
```

Ce n'est qu'un indice de ce que les modèles peuvent faire. Un modèle peut correspondre à une plage de valeurs. Il peut déballer les tuples. Il peut correspondre à des champs individuels de structs. Il peut chasser les références, emprunter des parties d'une valeur, et plus encore. Les motifs de Rust sont un mini-langage qui leur est propre. Nous leur consacrerons plusieurs pages au [chapitre 10](#).

La forme générale d'une expression est :match

```
match value {
    pattern => expr,
    ...
}
```

La virgule après un bras peut être lâchée si le est un bloc. expr

Rust vérifie le donné par rapport à chaque motif à tour de rôle, en commençant par le premier. Lorsqu'un modèle correspond, le correspondant est évalué et l'expression est complète ; aucun autre modèle n'est vérifié. Au moins un des modèles doit correspondre. Rust interdit les expressions qui ne couvrent pas toutes les valeurs possibles

```
:value expr match match
```

```
let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
}; // error: nonexhaustive patterns
```

Tous les blocs d'une expression doivent produire des valeurs du même type :if

```
let suggested_pet =
    if with_wings { Pet::Buzzard } else { Pet::Hyena }; // ok

let favorite_number =
```

```
if user.is_hobbit() { "eleventy-one" } else { 9 }; // error

let best_sports_team =
  if is_hockey_season() { "Predators" }; // error
```

(Le dernier exemple est une erreur car en juillet, le résultat serait .) ()

De même, tous les bras d'une expression doivent avoir le même type

:match

```
let suggested_pet =
  match favorites.element {
    Fire => Pet::RedPanda,
    Air => Pet::Buffalo,
    Water => Pet::Orca,
    _ => None // error: incompatible types
  };
```

si laisser

Il existe une autre forme, l'expression: if let

```
if let pattern = expr {
  block1
} else {
  block2
}
```

Le donné correspond à , auquel cas s'exécute, ou ne correspond pas, et s'exécute. Parfois, c'est un bon moyen d'obtenir des données d'un ou

:expr pattern block1 block2 Option Result

```
if let Some(cookie) = request.session_cookie {
  return restore_session(cookie);
}

if let Err(err) = show_cheesy_anti_robot_task() {
  log_robot_attempt(err);
  politely_accuse_user_of_being_a_robot();
} else {
  session.mark_as_human();
}
```

Il n'est jamais strictement nécessaire de l'utiliser, car peut faire tout ce qu'il peut faire. Une expression est un raccourci pour un avec un seul motif: if let match if let if let match

```
match expr {  
    pattern => { block1 }  
    _ => { block2 }  
}
```

Boucles

Il existe quatre expressions en boucle :

```
while condition {  
    block  
}  
  
while let pattern = expr {  
    block  
}  
  
loop {  
    block  
}  
  
for pattern in iterable {  
    block  
}
```

Les boucles sont des expressions dans Rust, mais la valeur de a ou loop est toujours , donc leur valeur n'est pas très utile. Une expression peut produire une valeur si vous en spécifiez une. while for () loop

Une boucle se comporte exactement comme l'équivalent C, sauf que, encore une fois, le doit être du type exact .while condition bool

La boucle est analogue à . Au début de chaque itération de boucle, la valeur de soit correspond à la donnée , auquel cas le bloc s'exécute, ou ne s'exécute pas, auquel cas la boucle se ferme. while let if let expr pattern

Permet d'écrire des boucles infinies. Il exécute le répété pour toujours (ou jusqu'à ce qu'un ou soit atteint ou que le fil panique). loop block break return

Une boucle évalue l'expression, puis évalue une fois pour chaque valeur dans l'itérateur résultant. De nombreux types peuvent être itérés, y compris toutes les collections standard comme et . La boucle C standard : for iterable block Vec HashMap for

```

for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}

```

est écrit comme ceci dans Rust:

```

for i in 0..20 {
    println!("{}", i);
}

```

Comme en C, le dernier numéro imprimé est . 19

L'opérateur produit une *plage*, une structure simple avec deux champs : et . est identique à . Les plages peuvent être utilisées avec des boucles car il s'agit d'un type itérable : il implémente le trait, dont nous discuterons au [chapitre 15](#). Les collections standard sont toutes itérables, tout comme les tableaux et les tranches.

```
.. start end 0..20 std::ops::Range {
start: 0, end: 20 } for Range std::iter::IntoIterator
```

Conformément à la sémantique de déplacement de Rust, une boucle sur une valeur consomme la valeur : `for`

```

let strings: Vec<String> = error_messages();
for s in strings {                                // each String is moved into s here...
    println!("{}", s);
}                                              // ...and dropped here
println!("{} error(s)", strings.len()); // error: use of moved value

```

Cela peut être gênant. Le remède facile consiste à boucler une référence à la collection à la place. La variable de boucle sera alors une référence à chaque élément de la collection :

```

for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}

```

Ici, le type de est , et le type de est

```
.&strings &Vec<String> rs &String
```

L'itération sur une référence fournit une référence à chaque élément

```
:mut mut
```

```

for rs in &mut strings { // the type of rs is &mut String
    rs.push('\n'); // add a newline to each string
}

```

[Le chapitre 15](#) couvre les boucles plus en détail et montre de nombreuses autres façons d'utiliser les itérateurs. `for`

Contrôler le flux en boucles

Une expression quitte une boucle englobante. (Dans Rust, ne fonctionne qu'en boucles. Ce n'est pas nécessaire dans les expressions, qui sont différentes des déclarations à cet égard.) `break` `break` `match` `switch`

Dans le corps d'un `loop`, vous pouvez donner une expression, dont la valeur devient celle de la boucle : `loop break`

```
// Each call to `next_line` returns either `Some(line)`, where
// `line` is a line of input, or `None`, if we've reached the end of
// the input. Return the first line that starts with "answer: ".
// Otherwise, return "answer: nothing".
let answer = loop {
    if let Some(line) = next_line() {
        if line.starts_with("answer: ") {
            break line;
        }
    } else {
        break "answer: nothing";
    }
};
```

Naturellement, toutes les expressions d'un `loop` produisent des valeurs avec le même type, qui devient le type de l'eux-mêmes. `break` `loop` `loop`

Une expression passe à l'itération de boucle suivante : `continue`

```
// Read some data, one line at a time.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Jump back to the top of the loop and
        // move on to the next line of input.
        continue;
    }
    ...
}
```

Dans une boucle, passe à la valeur suivante de la collection. S'il n'y a plus de valeurs, la boucle se ferme. De même, dans une boucle, revérifie la condition de boucle. Si c'est maintenant faux, la boucle se ferme. `for` `continue` `while` `continue`

Une boucle peut être étiquetée avec une durée de vie. Dans l'exemple suivant, est une étiquette pour la boucle externe. Ainsi, sort de cette boucle, pas de la boucle interne: 'search: for break 'search

```
'search:  
for room in apartment {  
    for spot in room.hiding_spots() {  
        if spot.contains(keys) {  
            println!("Your keys are {} in the {}.", spot, room);  
            break 'search;  
        }  
    }  
}
```

A peut avoir à la fois une étiquette et une expression de valeur : break

```
// Find the square root of the first perfect square  
// in the series.  
let sqrt = 'outer: loop {  
    let n = next_number();  
    for i in 1.. {  
        let square = i * i;  
        if square == n {  
            // Found a square root.  
            break 'outer i;  
        }  
        if square > n {  
            // `n` isn't a perfect square, try the next  
            break;  
        }  
    }  
};
```

Les étiquettes peuvent également être utilisées avec . continue

expressions de retour

Une expression quitte la fonction active et renvoie une valeur à l'appelant. return

return sans valeur est un raccourci pour : return ()

```
fn f() {      // return type omitted: defaults to ()  
    return;   // return value omitted: defaults to ()  
}
```

Les fonctions n'ont pas besoin d'avoir une expression explicite. Le corps d'une fonction fonctionne comme une expression de bloc : si la dernière

expression n'est pas suivie d'un point-virgule, sa valeur est la valeur de retour de la fonction. En fait, c'est le moyen préféré de fournir la valeur de retour d'une fonction dans Rust. `return`

Mais cela ne signifie pas que c'est inutile, ou simplement une concession aux utilisateurs qui ne sont pas expérimentés avec les langages d'expression. Comme une expression, peut abandonner le travail en cours. Par exemple, dans [le chapitre 2](#), nous avons utilisé l'opérateur pour vérifier les erreurs après l'appel d'une fonction qui peut échouer

```
: return break return ?
```

```
let output = File::create(filename)?;
```

Nous avons expliqué qu'il s'agit d'un raccourci pour une expression: `match`

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

Ce code commence par appeler `.Ok`. Si cela renvoie `Ok`, alors l'expression entière est évaluée à `f`, donc est stockée dans `output`, et nous continuons avec la ligne de code suivante suivant le

```
.File::create(filename) Ok(f) match f f output match
```

Sinon, nous allons faire correspondre et frapper l'expression. Lorsque cela se produit, peu importe que nous soyons en train d'évaluer une expression pour déterminer la valeur de la variable `f`. Nous abandonnons tout cela et quittons la fonction englobante, renvoyant toute erreur que nous avons obtenue de

```
.Err(err) return match output File::create()
```

Nous couvrirons l'opérateur plus complètement dans [« Propagation des erreurs »](#). ?

Pourquoi Rust Has loop

Plusieurs éléments du compilateur Rust analysent le flux de contrôle à travers votre programme :

- Rust vérifie que chaque chemin d'accès à une fonction renvoie une valeur du type de retour attendu. Pour ce faire correctement, il doit savoir s'il est possible d'atteindre la fin de la fonction.
- Rust vérifie que les variables locales ne sont jamais utilisées sans initialisation. Cela implique de vérifier chaque chemin à travers une fonc-

tion pour s'assurer qu'il n'y a aucun moyen d'atteindre un endroit où une variable est utilisée sans avoir déjà passé par le code qui l'initialise.

- Rust met en garde contre le code inaccessible. Le code est inaccessible si aucun chemin *d'accès* à travers la fonction ne l'atteint.

C'est ce qu'on appelle des analyses *sensibles au flux*. Ils ne sont pas nouveaux; Java a une analyse « d'affectation définie », similaire à celle de Rust, pendant des années.

Lors de l'application de ce type de règle, un langage doit trouver un équilibre entre la simplicité, qui permet aux programmeurs de comprendre plus facilement de quoi parle parfois le compilateur, et l'intelligence, qui peut aider à éliminer les faux avertissements et les cas où le compilateur rejette un programme parfaitement sûr. Rust a opté pour la simplicité. Ses analyses sensibles au flux n'examinent pas du tout les conditions de boucle, mais supposent simplement que n'importe quelle condition d'un programme peut être vraie ou fausse.

Cela amène Rust à rejeter certains programmes sûrs:

```
fn wait_for_process(process: &mut Process) -> i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
} // error: mismatched types: expected i32, found ()
```

L'erreur ici est fausse. Cette fonction ne sort que via l'instruction, de sorte que le fait que la boucle ne produise pas de an n'est pas pertinent. `return while i32`

L'expression est proposée comme une solution « dites ce que vous voulez dire » à ce problème. `loop`

Le système de type Rust est également affecté par le flux de contrôle. Plus tôt, nous avons dit que toutes les branches d'une expression doivent avoir le même type. Mais il serait idiot d'appliquer cette règle sur les blocs qui se terminent par une expression ou une expression, un infini, ou un appel à ou . Ce que toutes ces expressions ont en commun, c'est qu'elles ne finissent jamais de la manière habituelle, produisant une valeur. A ou quitte brusquement le bloc actuel, un infini ne se termine jamais du tout, et ainsi de suite. `if break return loop panic!`

```
() std::process::exit() break return loop
```

Donc, dans Rust, ces expressions n'ont pas un type normal. Les expressions qui ne se terminent pas normalement se voient attribuer le type

spécial et sont exemptées des règles sur les types à faire correspondre.

Vous pouvez voir dans la signature de fonction de

```
:! ! std::process::exit()
```

```
fn exit(code: i32) -> !
```

Les moyens qui ne reviennent jamais. C'est une *fonction divergente*. ! `exit()`

Vous pouvez écrire vos propres fonctions divergentes en utilisant la même syntaxe, ce qui est parfaitement naturel dans certains cas:

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) -> ! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

Bien sûr, Rust considère alors qu'il s'agit d'une erreur si la fonction peut revenir normalement.

Avec ces blocs de construction de flux de contrôle à grande échelle en place, nous pouvons passer aux expressions plus fines généralement utilisées dans ce flux, comme les appels de fonction et les opérateurs arithmétiques.

Appels de fonction et de méthode

La syntaxe pour appeler des fonctions et des méthodes est la même dans Rust que dans de nombreux autres langages :

```
let x = gcd(1302, 462); // function call

let room = player.location(); // method call
```

Dans le deuxième exemple ici, `room` est une variable du type inventé , qui a une méthode inventée. (Nous montrerons comment définir vos propres méthodes lorsque nous commencerons à parler des types définis par l'utilisateur au [chapitre 9](#).) `player Player .location()`

Rust fait généralement une distinction nette entre les références et les valeurs auxquelles elles se réfèrent. Si vous passez à une fonction qui attend un , il s'agit d'une erreur de type. Vous remarquerez que l'opérateur assouplit un peu ces règles. Dans l'appel de méthode , il peut s'agir

d'un , d'une référence de type , ou d'un pointeur intelligent de type ou . La méthode peut prendre le joueur soit par valeur, soit par référence. La même syntaxe fonctionne dans tous les cas, car l'opérateur de Rust déréférence automatiquement ou emprunte une référence à celui-ci au besoin. `&i32 i32 . player.location() player Player &Player Box<Player> Rc<Player> .location() .location() .player`

Une troisième syntaxe est utilisée pour appeler des fonctions associées au type, comme : `Vec::new()`

```
let mut numbers = Vec::new(); // type-associated function call
```

Celles-ci sont similaires aux méthodes statiques dans les langages orientés objet : les méthodes ordinaires sont appelées sur des valeurs (comme), et les fonctions associées au type sont appelées sur des types (comme `).my_vec.len() Vec::new()`)

Naturellement, les appels de méthode peuvent être enchaînés :

```
// From the Actix-based web server in Chapter 2:  
server  
    .bind("127.0.0.1:3000").expect("error binding server to address")  
    .run().expect("error running server");
```

Une particularité de la syntaxe Rust est que dans un appel de fonction ou un appel de méthode, la syntaxe habituelle pour les types génériques, , ne fonctionne pas: `Vec<T>`

```
return Vec<i32>::with_capacity(1000); // error: something about chained c  
  
let ramp = (0 .. n).collect<Vec<i32>>(); // same error
```

Le problème est que dans les expressions, est l'opérateur inférieur. Le compilateur Rust suggère utilement d'écrire au lieu de dans ce cas, et cela résout le problème: `< ::<T> <T>`

```
return Vec::with_capacity(1000); // ok, using ::<  
  
let ramp = (0 .. n).collect::<Vec<i32>>(); // ok, using ::<
```

Le symbole est affectueusement connu dans la communauté Rust sous le nom de *turbofish*. `::<...>`

Alternativement, il est souvent possible de laisser tomber les paramètres de type et de laisser Rust les déduire:

```
return Vec::with_capacity(10); // ok, if the fn return type is Vec<i32>

let ramp: Vec<i32> = (0 .. n).collect(); // ok, variable's type is given
```

Il est considéré comme un bon style d'omettre les types chaque fois qu'ils peuvent être déduits.

Champs et éléments

Les champs d'une structure sont accessibles à l'aide d'une syntaxe familière. Les tuples sont les mêmes sauf que leurs champs ont des nombres plutôt que des noms :

```
game.black_pawns    // struct field
coords.1            // tuple element
```

Si la valeur à gauche du point est un type de référence ou de pointeur intelligent, elle est automatiquement déréférencée, tout comme pour les appels de méthode.

Les crochets accèdent aux éléments d'un tableau, d'une tranche ou d'un vecteur :

```
pieces[i]          // array element
```

La valeur à gauche des crochets est automatiquement déréférencée.

Des expressions comme celles-ci sont *appelées lvalues*, car elles peuvent apparaître sur le côté gauche d'une affectation :

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

Bien entendu, cela n'est autorisé que si , et sont déclarés en tant que variables. game coords pieces mut

L'extraction d'une tranche à partir d'un tableau ou d'un vecteur est simple :

```
let second_half = &game_moves[midpoint .. end];
```

Ici peut être un tableau, une tranche ou un vecteur; le résultat, quoi qu'il en soit, est une tranche de longueur empruntée. est considéré comme em-

```
prunté pour la durée de vie de .game_moves end -  
midpoint game_moves second_half
```

L'opérateur permet d'omettre l'un ou l'autre opérande ; il produit jusqu'à quatre types d'objets différents en fonction des opérandes présents : ..

```
..      // RangeFull  
a ..    // RangeFrom { start: a }  
.. b    // RangeTo { end: b }  
a .. b // Range { start: a, end: b }
```

Les deux dernières formes sont *exclusives à la fin* (ou à *moitié ouvertes*) : la valeur finale n'est pas incluse dans la plage représentée. Par exemple, la plage comprend les nombres , et .0 .. 3 0 1 2

L'opérateur produit des plages *finales inclusives* (ou *fermées*), qui incluent la valeur finale : ..=

```
..= b    // RangeToInclusive { end: b }  
a ..= b // RangeInclusive::new(a, b)
```

Par exemple, la plage comprend les nombres , , et .0 ..= 3 0 1 2 3

Seules les plages qui incluent une valeur de départ sont itérables, car une boucle doit avoir un endroit pour commencer. Mais dans le découpage de tableau, les six formes sont utiles. Si le début ou la fin de la plage est omis, il est par défaut le début ou la fin des données découpées.

Ainsi, une implémentation de quicksort, l'algorithme de tri classique de diviser pour régner, pourrait ressembler, en partie, à ceci:

```
fn quicksort<T: Ord>(slice: &mut [T]) {  
    if slice.len() <= 1 {  
        return; // Nothing to sort.  
    }  
  
    // Partition the slice into two parts, front and back.  
    let pivot_index = partition(slice);  
  
    // Recursively sort the front half of `slice`.  
    quicksort(&mut slice[.. pivot_index]);  
  
    // And the back half.  
    quicksort(&mut slice[pivot_index + 1 ..]);  
}
```

Opérateurs de référence

L'adresse des opérateurs et , sont traitées au [chapitre 5](#). & &mut

L'opérateur unary est utilisé pour accéder à la valeur indiquée par une référence. Comme nous l'avons vu, Rust suit automatiquement les références lorsque vous utilisez l'opérateur pour accéder à un champ ou à une méthode, de sorte que l'opérateur n'est nécessaire que lorsque nous voulons lire ou écrire la valeur entière vers laquelle pointe la référence. * . *

Par exemple, il arrive qu'un itérateur produise des références, mais que le programme ait besoin des valeurs sous-jacentes :

```
let padovan: Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

Dans cet exemple, le type de est , donc est un . elem &u64 *elem u64

Opérateurs arithmétiques, binaires, de comparaison et logiques

Les opérateurs binaires de Rust sont comme ceux de beaucoup d'autres langues. Pour gagner du temps, nous supposons une familiarité avec l'une de ces langues et nous nous concentrons sur les quelques points où Rust s'écarte de la tradition.

Rust a les opérateurs arithmétiques habituels, , , , et . Comme mentionné dans [le chapitre 3](#), le dépassement d'entier est détecté et provoque une panique dans les versions de débogage. La bibliothèque standard fournit des méthodes telles que l'arithmétique non cochée. + -

```
* / % a.wrapping_add(b)
```

La division des entiers arrondit vers zéro, et la division d'un entier par zéro déclenche une panique même dans les versions de version. Les entiers ont une méthode qui renvoie un (si est nul) et ne panique jamais. a.checked_div(b) Option None b

Unary annule un certain nombre. Il est pris en charge pour tous les types numériques à l'exception des entiers non signés. Il n'y a pas d'opérateur unary. - +

```
println!("{}", -100);      // -100
println!("{}", -100u32);   // error: can't apply unary `--` to type `u32`
println!("{}", +100);      // error: expected expression, found `+`
```

Comme en C, calcule le reste signé, ou module, de l'arrondi de division vers zéro. Le résultat a le même signe que l'opérande gauche. Notez que cela peut être utilisé sur des nombres à virgule flottante ainsi que sur des entiers : a % b %

```
let x = 1234.567 % 10.0; // approximately 4.567
```

Rust hérite également des opérateurs entiers binaires de C, , , et .

Cependant, Rust utilise au lieu de pour bitwise NOT: & | ^ << >> ! ~

```
let hi: u8 = 0xe0;
let lo = !hi; // 0x1f
```

Cela signifie qu'il ne peut pas être utilisé sur un entier pour signifier « n est nul ». Pour cela, écrivez n == 0 . !n n

Le décalage de bits s'étend toujours sur les types entiers signés et l'extension zéro sur les types entiers non signés. Étant donné que Rust a des entiers non signés, il n'a pas besoin d'un opérateur shift non signé, comme l'opérateur de Java. >>>

Les opérations binaires ont une priorité plus élevée que les comparaisons, contrairement à C, donc si vous écrivez , cela signifie , comme vous l'aviez probablement prévu. C'est beaucoup plus utile que l'interprétation de C, qui teste le mauvais morceau! x & BIT != 0 (x & BIT) != 0 x & (BIT != 0)

Les opérateurs de comparaison de Rust sont , , , et . Les deux valeurs comparées doivent avoir le même type. == != < <= > >=

Rust a également les deux opérateurs logiques de court-circuit et . Les deux opérandes doivent avoir le type exact . && || bool

Mission

L'opérateur peut être utilisé pour affecter des variables et leurs champs ou éléments. Mais l'affectation n'est pas aussi courante dans Rust que dans d'autres langages, car les variables sont immuables par défaut. = mut

Comme décrit au [chapitre 4](#), si la valeur a un non-type, l'affectation la déplace vers la destination. La propriété de la valeur est transférée de la source à la destination. La valeur antérieure de la destination, le cas échéant, est supprimée. Copy

L'affectation composée est prise en charge :

```
total += item.price;
```

Cela équivaut à . D'autres opérateurs sont également pris en charge : , , et ainsi de suite. La liste complète est donnée dans [le tableau 6-1](#), plus haut dans ce chapitre. `total = total + item.price; -= *=`

Contrairement à C, Rust ne prend pas en charge l'affectation de chaînage : vous ne pouvez pas écrire pour affecter la valeur à la fois à et . L'affectation est suffisamment rare dans Rust pour que vous ne manquiez pas ce raccourci. `a = b = 3` `3 a b`

Rust n'a pas d'opérateurs d'incrément et de décrément de C et . `++` `--`

Caractères moulés

La conversion d'une valeur d'un type à un autre nécessite généralement une conversion explicite dans Rust. Les casts utilisent le mot-clé : `as`

```
let x = 17;           // x is type i32
let index = x as usize; // convert to usize
```

Plusieurs types de moulages sont autorisés:

- Les nombres peuvent être convertis de n'importe quel type numérique intégré à n'importe quel autre.

La conversion d'un entier en un autre type d'entier est toujours bien définie. La conversion en un type plus étroit entraîne une troncature. Un entier signé converti en un type plus large est étendu par signe, un entier non signé est étendu à zéro, etc. Bref, il n'y a pas de surprises. La conversion d'un type à virgule flottante en un type entier arrondit vers zéro : la valeur de est . Si la valeur est trop grande pour tenir dans le type entier, la distribution produit la valeur la plus proche que le type entier peut représenter : la valeur de est `-1.99 as i32` `-1 1e6 as u8 255`

- Les valeurs de type ou , ou de type C, peuvent être converties en n'importe quel type entier. (Nous couvrirons les énumérations au [chapitre 10](#).) `bool char enum`

La conversion dans l'autre sens n'est pas autorisée, car , et les types ont tous des restrictions sur leurs valeurs qui devraient être appliquées avec des contrôles d'exécution. Par exemple, la conversion d'un type à type est interdite car certaines valeurs, comme , correspondent à des points de code de substitution Unicode et ne feraient donc pas de valeurs valides. Il existe une méthode standard, , qui effectue la vérification d'exécution et renvoie un ; mais plus précisément, le besoin de

ce type de conversion est devenu rare. Nous convertissons généralement des chaînes ou des flux entiers à la fois, et les algorithmes sur le texte Unicode sont souvent non triviaux et il est préférable de les laisser aux

```
bibliothèques. bool char enum u16 char u16 0xd800 char std::char::from_u32() Option<char>
```

À titre d'exception, a peut être converti en type , car tous les entiers compris entre 0 et 255 sont des points de code Unicode valides pour être conservés. u8 char char

- Certains moulages impliquant des types de pointeurs dangereux sont également autorisés. Voir [« Pointeurs bruts »](#).

Nous avons dit qu'une conversion nécessite *généralement* un casting.

Quelques conversions impliquant des types de référence sont si simples que le langage les effectue même sans cast. Un exemple trivial est la conversion d'une référence en une non-référence. mut mut

Plusieurs conversions automatiques plus importantes peuvent se produire, cependant:

- Les valeurs de type sont converties automatiquement en type sans moulage. &String &str
- Valeurs de type auto-conversion en . &Vec<i32> &[i32]
- Valeurs de type auto-conversion en . &Box<Chessboard> &Chessboard

Celles-ci sont *appelées coercitions deref*, car elles s'appliquent aux types qui implémentent le trait intégré. Le but de la coercition est de faire en sorte que les types de pointeurs intelligents, comme , se comportent autant que possible comme la valeur sous-jacente. L'utilisation d'un est la plupart du temps comme l'utilisation d'un simple , grâce à

```
. Deref Deref Box Box<Chessboard> Chessboard Deref
```

Les types définis par l'utilisateur peuvent également implémenter le trait. Lorsque vous devez écrire votre propre type de pointeur intelligent, voir [« Deref et DerefMut »](#). Deref

Fermetures

La rouille a *des fermetures*, des valeurs de fonction légères. Une fermeture se compose généralement d'une liste d'arguments, donnée entre des barres verticales, suivie d'une expression :

```
let is_even = |x| x % 2 == 0;
```

Rust déduit les types d'arguments et le type de retour. Vous pouvez également les écrire explicitement, comme vous le feriez pour une fonction. Si vous spécifiez un type de retour, le corps de la fermeture doit être un bloc, par souci de santé syntaxique:

```
let is_even = |x: u64| -> bool x % 2 == 0; // error  
let is_even = |x: u64| -> bool { x % 2 == 0 }; // ok
```

L'appel d'une fermeture utilise la même syntaxe que l'appel d'une fonction :

```
assert_eq!(is_even(14), true);
```

Les fermetures sont l'une des caractéristiques les plus délicieuses de Rust, et il y a beaucoup plus à dire à leur sujet. Nous dis-le au [chapitre 14](#).

En avant

Les expressions sont ce que nous considérons comme du « code en cours d'exécution ». Ils font partie d'un programme Rust qui compile les instructions de la machine. Pourtant, ils ne représentent qu'une petite fraction de l'ensemble de la langue.

Il en va de même dans la plupart des langages de programmation. Le premier travail d'un programme est de s'exécuter, mais ce n'est pas son seul travail. Les programmes doivent communiquer. Ils doivent être testables. Ils doivent rester organisés et flexibles pour pouvoir continuer à évoluer. Ils doivent interagir avec le code et les services construits par d'autres équipes. Et même juste pour s'exécuter, les programmes dans un langage typé statiquement comme Rust ont besoin de plus d'outils pour organiser les données que de simples tuples et tableaux.

À venir, nous passerons plusieurs chapitres à parler des fonctionnalités dans ce domaine: les modules et les caisses, qui donnent la structure de votre programme, puis les structs et les enums, qui font de même pour vos données.

Tout d'abord, nous allons consacrer quelques pages au sujet important de ce qu'il faut faire lorsque les choses tournent mal.

Chapitre 7. Gestion des erreurs

Je savais que si je restais assez longtemps, quelque chose comme ça arriverait.

—George Bernard Shaw sur la mort

L'approche de Rust en matière de gestion des erreurs est suffisamment inhabituelle pour justifier un court chapitre sur le sujet. Il n'y a pas d'idées difficiles ici, juste des idées qui pourraient être nouvelles pour vous. Ce chapitre couvre les deux différents types de gestion des erreurs dans Rust: panique et s. Result

Les erreurs ordinaires sont gérées à l'aide du type. représentent généralement des problèmes causés par des éléments extérieurs au programme, tels qu'une entrée erronée, une panne de réseau ou un problème d'autorisations. Que de telles situations se produisent ne dépend pas de nous; même un programme sans bug les rencontrera de temps en temps. La majeure partie de ce chapitre est consacrée à ce genre d'erreur. Nous allons d'abord couvrir la panique, cependant, parce que c'est le plus simple des deux. Result Result

La panique est pour l'autre type d'erreur, celle qui *ne devrait jamais arriver.*

Panique

Un programme panique lorsqu'il rencontre quelque chose de si gâché qu'il doit y avoir un bogue dans le programme lui-même. Quelque chose comme :

- Accès au tableau hors limites
- Division des entiers par zéro
- Faire appel à un qui se trouve être .expect() Result Err
- Échec de l'assertion

(Il y a aussi la macro , pour les cas où votre propre code découvre qu'il a mal tourné, et vous devez donc déclencher une panique directement. accepte les arguments facultatifs de style pour créer un message d'erreur.) panic!() panic!() println!()

Ce que ces conditions ont en commun, c'est qu'elles sont toutes, pour ne pas mettre un point trop fin, la faute du programmeur. Une bonne règle de base est : « Ne paniquez pas. »

Mais nous faisons tous des erreurs. Lorsque ces erreurs qui ne devraient pas se produire se produisent, que se passe-t-il alors? Remarquablement, Rust vous donne le choix. La rouille peut soit dérouler la pile lorsqu'une panique se produit, soit interrompre le processus. Le dénouement est la valeur par défaut.

Déroulement

Lorsque les pirates séparent le butin d'un raid, le capitaine obtient la moitié du butin. Les membres d'équipage ordinaires gagnent des parts égales de l'autre moitié. (Les pirates détestent les fractions, donc si l'une ou l'autre division ne sort pas même, le résultat est arrondi vers le bas, le reste allant au perroquet du navire.)

```
fn pirate_share(total: u64, crew_size: usize) -> u64 {
    let half = total / 2;
    half / crew_size as u64
}
```

Cela peut bien fonctionner pendant des siècles jusqu'au jour où il s'avère que le capitaine est le seul survivant d'un raid. Si nous passons un de zéro à cette fonction, elle sera divisée par zéro. En C++, il s'agit d'un comportement non défini. Dans Rust, cela déclenche une panique, qui se déroule généralement comme suit: `crew_size`

- Un message d'erreur est imprimé sur le terminal :

```
thread 'main' panicked at 'attempt to divide by zero', pirates.rs:3780
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

Si vous définissez la variable d'environnement, comme le suggèrent les messages, Rust videra également la pile à ce stade. `RUST_BACKTRACE`

- La pile est déroulée. Cela ressemble beaucoup à la gestion des exceptions C++.

Toutes les valeurs temporaires, variables locales ou arguments utilisés par la fonction actuelle sont supprimés, à l'inverse de l'ordre dans lequel ils ont été créés. Laisser tomber une valeur signifie simplement nettoyer après elle: tous les `s` ou `s` que le programme utilisait sont libérés, tous les `s` ouverts sont fermés, etc. Les méthodes définies par

l'utilisateur sont également appelées ; voir [« Déposer »](#). Dans le cas particulier de , il n'y a rien à

```
nettoyer. String Vec File drop_pirate_share()
```

Une fois l'appel de fonction actuel nettoyé, nous passons à son appelant, en déposant ses variables et ses arguments de la même manière. Ensuite, nous passons à l'appelant de *cette* fonction, et ainsi de suite en haut de la pile.

- Enfin, le thread se ferme. Si le thread de panique était le thread principal, alors l'ensemble du processus se ferme (avec un code de sortie non nul).

Peut-être que *la panique* est un nom trompeur pour ce processus ordonné. Une panique n'est pas un crash. Ce n'est pas un comportement indéfini. C'est plus comme un en Java ou un en C++. Le comportement est bien défini ; cela ne devrait tout simplement pas se

```
produire. RuntimeException std::logic_error
```

La panique est sans danger. Il ne viole aucune des règles de sécurité de Rust; même si vous parvenez à paniquer au milieu d'une méthode de bibliothèque standard, elle ne laissera jamais un pointeur pendant ou une valeur à moitié initialisée en mémoire. L'idée est que Rust attrape l'accès au tableau non valide, ou quoi que ce soit, *avant* que quelque chose de mauvais ne se produise. Il serait dangereux de continuer, alors Rust déroule la pile. Mais le reste du processus peut continuer à fonctionner.

La panique est par fil. Un fil peut paniquer tandis que d'autres fils de discussion parlent de leurs activités normales. Dans [le chapitre 19](#), nous montrerons comment un thread parent peut savoir quand un thread enfant panique et gérer l'erreur avec élégance.

Il existe également un moyen *d'attraper* le déroulement de la pile, permettant au fil de survivre et de continuer à fonctionner. C'est ce que fait la fonction de bibliothèque standard. Nous ne verrons pas comment l'utiliser, mais c'est le mécanisme utilisé par le harnais de test de Rust pour récupérer lorsqu'une assertion échoue dans un test. (Cela peut également être nécessaire lors de l'écriture de code Rust qui peut être appelé à partir de C ou C++, car le déroulement sur du code non-Rust est un comportement non défini; voir [le chapitre 22](#).) `std::panic::catch_unwind()`

Idéalement, nous aurions tous un code sans bug qui ne panique jamais. Mais personne n'est parfait. Vous pouvez utiliser des threads et gérer la panique, ce qui rend votre programme plus robuste. Une mise en garde importante est que ces outils n'attrapent que les paniques qui déroulent

la pile. Toutes les paniques ne se déroulent pas de cette façon. `catch_unwind()`

Abandon

Le déroulage de la pile est le comportement de panique par défaut, mais il existe deux circonstances dans lesquelles Rust n'essaie pas de dérouler la pile.

Si une méthode déclenche une deuxième panique alors que Rust essaie toujours de nettoyer après la première, cela est considéré comme fatal. La rouille cesse de se dérouler et interrompt tout le processus. `.drop()`

En outre, le comportement de panique de Rust est personnalisable. Si vous compilez avec `-C panic=abort`, la première panique dans votre programme interrompt immédiatement le processus. (Avec cette option, Rust n'a pas besoin de savoir comment dérouler la pile, ce qui peut réduire la taille de votre code compilé.)

Ceci conclut notre discussion sur la panique dans Rust. Il n'y a pas grand-chose à dire, car le code Rust ordinaire n'a aucune obligation de gérer la panique. Même si vous utilisez des threads ou `std::sync::mpsc`, tout votre code de gestion de panique sera probablement concentré à quelques endroits. Il est déraisonnable de s'attendre à ce que chaque fonction d'un programme anticipe et gère les bogues dans son propre code. Les erreurs causées par d'autres facteurs sont une autre marmite de poisson. `catch_unwind()`

Résultat

La rouille n'a pas d'exceptions. Au lieu de cela, les fonctions qui peuvent échouer ont un type de retour qui dit ceci :

```
fn get_weather(location: LatLng) -> Result<WeatherReport, io::Error>
```

Le type indique une défaillance possible. Lorsque nous appelons la fonction, elle renvoie soit un *résultat de réussite*, où est une nouvelle valeur, soit un *résultat d'erreur*, où est une explication de ce qui s'est mal passé. `Result<WeatherReport, io::Error>`

Rust nous oblige à écrire une sorte de gestion des erreurs chaque fois que nous appelons cette fonction. Nous ne pouvons pas aller à la sans faire

quelque chose à la , et vous obtiendrez un avertissement du compilateur si une valeur n'est pas utilisée. `WeatherReport` `Result` `Result`

Dans [le chapitre 10](#), nous verrons comment la bibliothèque standard définit et comment vous pouvez définir vos propres types similaires. Pour l'instant, nous allons adopter une approche de « livre de recettes » et nous concentrer sur la façon d'utiliser `s` pour obtenir le comportement de gestion des erreurs que vous souhaitez. Nous verrons comment détecter, propager et signaler les erreurs, ainsi que les modèles courants d'organisation et de travail avec les types. `Result` `Result` `Result`

Détection des erreurs

La façon la plus complète de traiter `a` est la façon dont nous l'avons montré au [chapitre 2](#) : utiliser une expression `Result match`

```
match get_weather(hometown) {
    Ok(report) => {
        display_weather(hometown, &report);
    }
    Err(err) => {
        println!("error querying the weather: {}", err);
        schedule_weather_retry();
    }
}
```

C'est l'équivalent de Rust dans d'autres langues. C'est ce que vous utilisez lorsque vous souhaitez gérer les erreurs de front, et non les transmettre à votre appelant. `try/catch`

`match` est un peu verbeux, offre donc une variété de méthodes qui sont utiles dans des cas particuliers courants. Chacune de ces méthodes a une expression dans sa mise en œuvre. (Pour la liste complète des méthodes, consultez la documentation en ligne. Les méthodes énumérées ici sont celles que nous utilisons le plus.) `Result<T, E> match Result`

```
result.is_ok(), result.is_err()
```

Renvoyer un indiquant s'il s'agit d'un résultat de réussite ou d'un résultat d'erreur. `bool result`

```
result.ok()
```

Renvoie la valeur de réussite, le cas échéant, sous la forme d'un fichier . Si `est` un résultat réussi, cela renvoie ; sinon, il renvoie , en ignorant la valeur d'erreur. `Option<T> result Some(success_value) None`

```
result.err()
```

Renvoie la valeur d'erreur, le cas échéant, sous la forme d'un fichier

```
.Option<E>
```

```
result.unwrap_or(fallback)
```

Renvoie la valeur de réussite, si est un résultat de réussite. Sinon, il renvoie , en ignorant la valeur d'erreur. `result fallback`

```
// A fairly safe prediction for Southern California.  
const THE_USUAL: WeatherReport = WeatherReport::Sunny(72);  
  
// Get a real weather report, if possible.  
// If not, fall back on the usual.  
let report = get_weather(los_angeles).unwrap_or(THE_USUAL);  
display_weather(los_angeles, &report);
```

C'est une bonne alternative à car le type de retour est , pas . Bien sûr, cela ne fonctionne que lorsqu'il existe une valeur de secours appropriée. `.ok() T Option<T>`

```
result.unwrap_or_else(fallback_fn)
```

C'est la même chose, mais au lieu de passer directement une valeur de secours, vous passez une fonction ou une fermeture. Ceci est pour les cas où il serait inutile de calculer une valeur de secours si vous n'allez pas l'utiliser. Le n'est appelé que si nous avons un résultat d'erreur. `fallback_fn`

```
let report =  
    get_weather(hometown)  
    .unwrap_or_else(|_err| vague_prediction(hometown));
```

([Le chapitre 14](#) traite en détail des fermetures.)

```
result.unwrap()
```

Renvoie également la valeur de réussite, si est un résultat de réussite. Cependant, s'il s'agit d'un résultat d'erreur, cette méthode panique. Cette méthode a ses utilités; nous en reparlerons plus tard. `result result`

```
result.expect(message)
```

C'est la même chose que , mais vous permet de fournir un message qu'il imprime en cas de panique. `.unwrap()`

Enfin, les méthodes de travail avec les références dans un : `Result`

```
result.as_ref()
```

Convertit à en fichier .Result<T, E> Result<&T, &E>

```
result.as_mut()
```

C'est la même chose, mais emprunte une référence mutable. Le type de retour est
.Result<&mut T, &mut E>

L'une des raisons pour lesquelles ces deux dernières méthodes sont utiles est que toutes les autres méthodes énumérées ici, à l'exception et , *consomment* le sur lequel elles fonctionnent. C'est-à-dire qu'ils prennent l'argument par valeur. Parfois, il est très pratique d'accéder aux données à l'intérieur d'un sans les détruire, et c'est ce que nous faisons. Par exemple, supposons que vous souhaitez appeler , mais que vous devez être laissé intact. Vous pouvez écrire , qui emprunte simplement , en renvoyant un plutôt qu'un

```
..is_ok() .is_err() result self result .as_ref() .as_mut() result.ok() result result.as_ref().ok() result Option<&T> Option<T>
```

Alias de type de résultat

Parfois, vous verrez la documentation Rust qui semble omettre le type d'erreur d'un : Result

```
fn remove_file(path: &Path) -> Result<()>
```

Cela signifie qu'un alias de type est utilisé. Result

Un alias de type est une sorte de raccourci pour les noms de type. Les modules définissent souvent un alias de type pour éviter d'avoir à répéter un type d'erreur utilisé de manière cohérente par presque toutes les fonctions du module. Par exemple, le module de la bibliothèque standard inclut cette ligne de code : Result std::io

```
pub type Result<T> = result::Result<T, Error>;
```

Cela définit un type public . C'est un alias pour , mais code en dur comme type d'erreur. En termes pratiques, cela signifie que si vous écrivez , alors Rust comprendra comme raccourci pour

```
.std::io::Result<T> Result<T, E> std::io::Error use std::io; io::Result<String> Result<String, io::Error>
```

Lorsque quelque chose comme apparaît dans la documentation en ligne, vous pouvez cliquer sur l'identifiant pour voir quel alias de type est utilisé et apprendre le type d'erreur. En pratique, c'est généralement évident d'après le contexte. `Result<()> Result`

Erreurs d'impression

Parfois, la seule façon de gérer une erreur est de la déverser sur le terminal et de passer à autre chose. Nous avons déjà montré une façon de le faire:

```
println!("error querying the weather: {}", err);
```

La bibliothèque standard définit plusieurs types d'erreur avec des noms ennuyeux : , , , etc. Tous implémentent une interface commune, le trait, ce qui signifie qu'ils partagent les fonctionnalités et méthodes suivantes: `std::io::Error std::fmt::Error std::str::Utf8Error std::error::Error`

```
println!()
```

Tous les types d'erreur sont imprimables à l'aide de ce document.

L'impression d'une erreur avec le spécificateur de format n'affiche généralement qu'un bref message d'erreur. Vous pouvez également imprimer avec le spécificateur de format pour obtenir une vue de l'erreur. Ceci est moins convivial, mais comprend des informations techniques supplémentaires. `{}` `{:?}` `Debug`

```
// result of `println!("error: {}", err);`  
error: failed to look up address information: No address associated  
hostname
```

```
// result of `println!("error: {:?}", err);`  
error: Error { repr: Custom(Custom { kind: Other, error: StringErro  
"failed to look up address information: No address associated with  
hostname" }) }
```

```
err.to_string()
```

Renvoie un message d'erreur sous la forme d'un fichier `.String`

```
err.source()
```

Renvoie une des erreurs sous-jacentes, le cas échéant, qui ont provoqué . Par exemple, une erreur de mise en réseau peut entraîner l'échec d'une transaction bancaire, ce qui peut entraîner la reprise de possession de votre bateau. Si est ,

peut renvoyer une erreur sur la transaction ayant échoué. Cette erreur peut être , et il peut s'agir de détails sur la panne de réseau spécifique qui a causé tout le tapage. Cette troisième erreur est la cause première, de sorte que sa méthode renverrait . Étant donné que la bibliothèque standard n'inclut que des fonctionnalités de niveau plutôt bas, la source des erreurs renvoyées par la bibliothèque standard est généralement

```
.Option err err.to_string() "boat was  
repossessed" err.source() .to_string() "failed to  
transfer $300 to United Yacht  
Supply" .source() io::Error .source() None None
```

L'impression d'une valeur d'erreur n'imprime pas également sa source. Si vous voulez être sûr d'imprimer toutes les informations disponibles, utilisez cette fonction:

```
use std::error::Error;  
use std::io::{Write, stderr};  
  
/// Dump an error message to `stderr`.  
///  
/// If another error happens while building the error message or  
/// writing to `stderr`, it is ignored.  
fn print_error(mut err: &dyn Error) {  
    let _ = writeln!(stderr(), "error: {}", err);  
    while let Some(source) = err.source() {  
        let _ = writeln!(stderr(), "caused by: {}", source);  
        err = source;  
    }  
}
```

La macro fonctionne comme , sauf qu'elle écrit les données dans un flux de votre choix. Ici, nous écrivons les messages d'erreur dans le flux d'erreur standard, . Nous pourrions utiliser la macro pour faire la même chose, mais panique si une erreur se produit. Dans , nous voulons ignorer les erreurs qui surviennent lors de l'écriture du message; nous expliquons pourquoi dans [« Ignorer les erreurs »](#), plus loin dans le chapitre. writeln! println! std::io::stderr eprintln! eprintln! print_error

Les types d'erreur de la bibliothèque standard n'incluent pas de trace de pile, mais la caisse populaire fournit un type d'erreur prêt à l'emploi qui le fait, lorsqu'il est utilisé avec une version instable du compilateur Rust. (À partir de Rust 1.56, les fonctions de la bibliothèque standard pour la capture des backtraces n'étaient pas encore stabilisées.) anyhow

Propagation des erreurs

Dans la plupart des endroits où nous essayons quelque chose qui pourrait échouer, nous ne voulons pas attraper et gérer l'erreur immédiatement. C'est tout simplement trop de code pour utiliser une instruction de 10 lignes à chaque endroit où quelque chose pourrait mal tourner. `match`

Au lieu de cela, si une erreur se produit, nous voulons généralement laisser notre appelant s'en occuper. Nous voulons que les erreurs *se propagent* vers le haut de la pile d'appels.

Rust a un opérateur qui fait cela. Vous pouvez ajouter un `à` n'importe quelle expression qui produit un `,`, tel que le résultat d'un appel de fonction : `? ? Result`

```
let weather = get_weather(hometown)?;
```

Le comportement dépend du fait que cette fonction renvoie un résultat de réussite ou un résultat d'erreur : `?`

- Sur le succès, il déballe le pour obtenir la valeur de succès à l'intérieur. Le type d'ici n'est pas mais simplement `WeatherReport.Result` `weather` `Result<WeatherReport, io::Error>`
- En cas d'erreur, il retourne immédiatement de la fonction englobante, en transmettant le résultat de l'erreur à la chaîne d'appel. Pour s'assurer que cela fonctionne, ne peut être utilisé que sur une fonction dans qui ont un type de retour. `? Result` `Result`

Il n'y a rien de magique chez l'opérateur. Vous pouvez exprimer la même chose en utilisant une expression, bien que ce soit beaucoup plus verbeux: `? match`

```
let weather = match get_weather(hometown) {  
    Ok(success_value) => success_value,  
    Err(err) => return Err(err)  
};
```

Les seules différences entre celui-ci et l'opérateur sont quelques points fins impliquant des types et des conversions. Nous couvrirons ces détails dans la section suivante. `?`

Dans le code plus ancien, vous pouvez voir la macro, qui était le moyen habituel de propager les erreurs jusqu'à ce que l'opérateur soit introduit

dans Rust 1.13 : try!() ?

```
let weather = try!(get_weather(hometown));
```

La macro se développe en une expression, comme celle précédente. match

Il est facile d'oublier à quel point la possibilité d'erreurs est omniprésente dans un programme, en particulier dans le code qui s'interface avec le système d'exploitation. L'opérateur apparaît parfois sur presque toutes les lignes d'une fonction : ?

```
use std::fs;
use std::io;
use std::path::Path;

fn move_all(src: &Path, dst: &Path) -> io::Result<()> {
    for entry_result in src.read_dir()? { // opening dir could fail
        let entry = entry_result?;           // reading dir could fail
        let dst_file = dst.join(entry.file_name());
        fs::rename(entry.path(), dst_file)?; // renaming could fail
    }
    Ok(()) // phew!
}
```

? fonctionne également de la même manière avec le type. Dans une fonction qui renvoie , vous pouvez utiliser pour décompresser une valeur et la renvoyer tôt dans le cas de : Option Option ? None

```
let weather = get_weather(hometown).ok()?;

```

Utilisation de plusieurs types d'erreurs

Souvent, plus d'une chose pourrait mal tourner. Supposons que nous lisons simplement des nombres à partir d'un fichier texte :

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

/// Read integers from a text file.
/// The file should have one number on each line.
fn read_numbers(file: &mut dyn BufRead) -> Result<Vec<i64>, io::Error>
{
    let mut numbers = vec![];
    for line_result in file.lines() {
        let line = line_result?;           // reading lines can fail
        if let Ok(number) = line.parse()
            numbers.push(number);
    }
    Ok(numbers)
}
```

```

        numbers.push(line.parse()?);      // parsing integers can fail
    }
    Ok(numbers)
}

```

Rust nous donne une erreur de compilateur:

```

error: `?` couldn't convert the error to `std::io::Error`

numbers.push(line.parse()?);      // parsing integers can fail
                                ^
the trait `std::convert::From<std::num::ParseIntError>`
is not implemented for `std::io::Error`

note: the question mark operation (`?`) implicitly performs a conversion
on the error value using the `From` trait

```

Les termes de ce message d'erreur auront plus de sens lorsque nous atteindrons [le chapitre 11](#), qui couvre les traits. Pour l'instant, notez simplement que Rust se plaint que l'opérateur ne peut pas convertir une valeur en type `.? std::num::ParseIntError std::io::Error`

Le problème ici est que la lecture d'une ligne d'un fichier et l'analyse d'un entier produisent deux types d'erreurs potentielles différents. Le type de `line` est `.Result<String, std::io::Error>`. Le type de retour de notre fonction ne s'adapte qu'à `s`. Rust essaie de faire face à la `line` en la convertissant en `un`, mais il n'y a pas une telle conversion, donc nous obtenons une erreur de

```

type line_result Result<String,
std::io::Error> line.parse() Result<i64, std::num::Parse
IntError> read_numbers() io::Error ParseIntError io::Error

```

Il y a plusieurs façons de traiter cela. Par exemple, la caisse que nous avons utilisée dans le [chapitre 2](#) pour créer des fichiers image de l'ensemble Mandelbrot définit son propre type d'erreur et implémente les conversions de `ImageError` et plusieurs autres types d'erreur vers `.Error`. Si vous souhaitez suivre cette voie, essayez la caisse, qui est conçue pour vous aider à définir de bons types d'erreurs avec seulement quelques lignes de code.

Une approche plus simple consiste à utiliser ce qui est intégré dans Rust. Tous les types d'erreur de bibliothèque standard peuvent être convertis en type `.Error`. C'est un peu une bouchée, mais représente « toute erreur » et permet de passer en toute sécurité entre les fils, ce que vous voudrez souvent.

```
Box<dyn std::error::Error + Send + Sync + 'static> dyn
```

```
std::error::Error Send + Sync + 'static' Pour plus de commodité, vous pouvez définir des alias de type :
```

```
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

Ensuite, remplacez le type de retour par . Avec ce changement, la fonction se compile. L'opérateur convertit automatiquement l'un ou l'autre type d'erreur en un au

```
besoin. read_numbers() GenericResult<Vec<i64>> ? GenericError
```

Incidemment, l'opérateur effectue cette conversion automatique en utilisant une méthode standard que vous pouvez utiliser vous-même. Pour convertir une erreur en type, appelez `GenericError::from()` :

```
GenericError
```

```
let io_error = io::Error::new(           // make our own io::Error
    io::ErrorKind::Other, "timed out");
return Err(GenericError::from(io_error)); // manually convert to Gener
```

Nous couvrirons le trait et sa méthode en détail dans [le chapitre 13](#). From `from()`

L'inconvénient de l'approche est que le type de retour ne communique plus précisément les types d'erreurs auxquelles l'appelant peut s'attendre. L'appelant doit être prêt à tout. `GenericError`

Si vousappelez une fonction qui renvoie `a` et que vous souhaitez gérer un type particulier d'erreur mais laisser toutes les autres se propager, utilisez la méthode générique . Il emprunte une référence à l'erreur, *s'il s'agit du type particulier d'erreur que vous recherchez*: `GenericResult error.downcast_ref::<ErrorType>()`

```
loop {
    match compile_project() {
        Ok(_) => return Ok(),
        Err(err) => {
            if let Some(mse) = err.downcast_ref::<MissingSemicolonError>
                insert_semicolon_in_source_code(mse.file(), mse.line())
                continue; // try again!
            }
            return Err(err);
    }
}
```

```
}
```

De nombreux langages ont une syntaxe intégrée pour ce faire, mais il s'avère que cela est rarement nécessaire. Rust a une méthode pour cela à la place.

Faire face aux erreurs qui « ne peuvent pas se produire »

Parfois, nous *savons* simplement qu'une erreur ne peut pas se produire. Par exemple, supposons que nous écrivions du code pour analyser un fichier de configuration et qu'à un moment donné, nous constatons que la prochaine chose dans le fichier est une chaîne de chiffres :

```
if next_char.is_digit(10) {  
    let start = current_index;  
    current_index = skip_digits(&line, current_index);  
    let digits = &line[start..current_index];  
    ...  
}
```

Nous voulons convertir cette chaîne de chiffres en un nombre réel. Il existe une méthode standard qui fait ceci:

```
let num = digits.parse::<u64>();
```

Maintenant, le problème: la méthode ne renvoie pas un `fichier`. Il renvoie un `fichier`. Il peut échouer, car certaines chaînes ne sont pas numériques
`:str.parse::<u64>() u64 Result`

```
"bleen".parse::<u64>() // ParseIntError: invalid digit
```

Mais il se trouve que nous savons que dans ce cas, se compose entièrement de chiffres. Que devrions-nous faire? `digits`

Si le code que nous écrivons renvoie déjà un , nous pouvons taper sur un et l'oublier. Sinon, nous sommes confrontés à la perspective irritante de devoir écrire du code de gestion des erreurs pour une erreur qui ne peut pas se produire. Le meilleur choix serait alors d'utiliser , une méthode qui panique si le résultat est un , mais renvoie simplement la valeur de succès d'un : `GenericResult ? .unwrap() Result Err Ok`

```
let num = digits.parse::<u64>().unwrap();
```

C'est comme ça, sauf que si nous nous trompons sur cette erreur, si cela peut arriver, alors dans ce cas, nous paniquerions. ?

En fait, nous nous trompons dans ce cas particulier. Si l'entrée contient une chaîne de chiffres suffisamment longue, le nombre sera trop grand pour tenir dans un : u64

```
"99999999999999999999".parse::<u64>()      // overflow error
```

L'utilisation dans ce cas particulier serait donc un bug. Une fausse entrée ne devrait pas provoquer de panique. .unwrap()

Cela dit, des situations se présentent où une valeur ne peut vraiment pas être une erreur. Par exemple, dans [le chapitre 18](#), vous verrez que le trait définit un ensemble commun de méthodes (et d'autres) pour le texte et la sortie binaire. Toutes ces méthodes renvoient des s, mais s'il vous arrive d'écrire à un , elles ne peuvent pas échouer. Dans de tels cas, il est acceptable d'utiliser ou de se passer du

```
s.Result Write .write() io::Result Vec<u8> .unwrap() .expect  
(message) Result
```

Ces méthodes sont également utiles lorsqu'une erreur indiquerait une condition si grave ou bizarre que la panique est exactement la façon dont vous voulez la gérer:

```
fn print_file_age(filename: &Path, last_modified: SystemTime) {  
    let age = last_modified.elapsed().expect("system clock drift");  
    ...  
}
```

Ici, la méthode ne peut échouer que si l'heure système est *antérieure* à celle de la création du fichier. Cela peut se produire si le fichier a été créé récemment et que l'horloge système a été ajustée à l'envers pendant l'exécution de notre programme. Selon la façon dont ce code est utilisé, c'est un appel raisonnable à la panique dans ce cas, plutôt que de gérer l'erreur ou de la propager à l'appelant. .elapsed()

Ignorer les erreurs

Parfois, nous voulons simplement ignorer complètement une erreur. Par exemple, dans notre fonction, nous avons dû gérer la situation improbable où l'impression de l'erreur déclenche une autre erreur. Cela pourrait

se produire, par exemple, si est canalisé vers un autre processus, et que ce processus est tué. L'erreur d'origine que nous essayions de signaler est probablement plus importante à propager, nous voulons donc simplement ignorer les problèmes avec , mais le compilateur Rust met en garde contre les valeurs inutilisées: `print_error() stderr stderr Result`

```
writeln!(stderr(), "error: {}", err); // warning: unused result
```

L'idiome est utilisé pour faire taire cet avertissement: `let _ = ...`

```
let _ = writeln!(stderr(), "error: {}", err); // ok, ignore result
```

Gestion des erreurs dans main()

Dans la plupart des endroits où un est produit, laisser la bulle d'erreur jusqu'à l'appelant est le bon comportement. C'est pourquoi il s'agit d'un seul personnage dans Rust. Comme nous l'avons vu, dans certains programmes, il est utilisé sur de nombreuses lignes de code d'affilée. `Result` ?

Mais si vous propagez une erreur assez longtemps, elle finit par atteindre , et quelque chose doit être fait avec elle. Normalement, ne peut pas utiliser car son type de retour n'est pas : `main() main() ? Result`

```
fn main() {
    calculate_tides()?;
}
```

Le moyen le plus simple de gérer les erreurs est d'utiliser

```
:main().expect()
```

```
fn main() {
    calculate_tides().expect("error");
}
```

Si renvoie un résultat d'erreur, la méthode panique. Paniquer dans le thread principal imprime un message d'erreur, puis se ferme avec un code de sortie non nul, ce qui est à peu près le comportement souhaité. Nous l'utilisons tout le temps pour de minuscules programmes. C'est un début. `calculate_tides().expect()`

Le message d'erreur est un peu intimidant, cependant:

```
$ tidecalc --planet mercury
thread 'main' panicked at 'error: "moon not found"', src/main.rs:2:23
note: run with `RUST_BACKTRACE=1` environment variable to display a bac
```

Le message d'erreur est perdu dans le bruit. En outre, c'est un mauvais conseil dans ce cas particulier. `RUST_BACKTRACE=1`

Cependant, vous pouvez également modifier la signature de type pour renvoyer un type, afin que vous puissiez utiliser : `main() Result <T, E>` ?

```
fn main() -> Result<(), TideCalcError> {
    let tides = calculate_tides()?;
    print_tides(tides);
    Ok(())
}
```

Cela fonctionne pour tout type d'erreur qui peut être imprimé avec le formateur, ce que tous les types d'erreur standard, comme `String`, peuvent être. Cette technique est facile à utiliser et donne un message d'erreur un peu plus agréable, mais ce n'est pas idéal: `{:?} std::io::Error`

```
$ tidecalc --planet mercury
Error: TideCalcError { error_type: NoMoon, message: "moon not found" }
```

Si vous avez des types d'erreur plus complexes ou si vous souhaitez inclure plus de détails dans votre message, il est utile d'imprimer le message d'erreur vous-même :

```
fn main() {
    if let Err(err) = calculate_tides() {
        print_error(&err);
        std::process::exit(1);
    }
}
```

Ce code utilise une expression pour imprimer le message d'erreur uniquement si l'appel renvoie un résultat d'erreur. Pour plus d'informations sur les expressions, [reportez-vous au chapitre 10](#). La fonction est répertoriée dans [« Erreurs d'impression »](#).

```
let calculate_tides() if let print_error
```

Maintenant, la sortie est belle et bien rangée:

```
$ tidecalc --planet mercury
error: moon not found
```

Déclaration d'un type d'erreur personnalisé

Supposons que vous écrivez un nouvel analyseur JSON et que vous souhaitez qu'il ait son propre type d'erreur. (Nous n'avons pas encore couvert les types définis par l'utilisateur; cela vient dans quelques chapitres. Mais les types d'erreur sont pratiques, nous allons donc inclure un aperçu ici.)

Approximativement le code minimum que vous écririez est :

```
// json/src/error.rs

#[derive(Debug, Clone)]
pub struct JsonError {
    pub message: String,
    pub line: usize,
    pub column: usize,
}
```

Cette structure sera appelée , et lorsque vous souhaitez déclencher une erreur de ce type, vous pouvez écrire : json::error::JsonError

```
return Err(JsonError {
    message: "expected ']' at end of array".to_string(),
    line: current_line,
    column: current_column
});
```

Cela fonctionnera bien. Toutefois, si vous souhaitez que votre type d'erreur fonctionne comme les types d'erreur standard, comme les utilisateurs de votre bibliothèque s'y attendent, vous avez encore un peu de travail à faire :

```
use std::fmt;

// Errors should be printable.
impl fmt::Display for JsonError {
    fn fmt(&self, f: &mut fmt::Formatter) -> Result<(), fmt::Error> {
        write!(f, "{} ({})", self.message, self.line, self.column)
    }
}
```

```
// Errors should implement the std::error::Error trait,  
// but the default definitions for the Error methods are fine.  
impl std::error::Error for JsonError { }
```

Encore une fois, la signification du mot-clé , et tout le reste seront expliqués dans les prochains chapitres. `impl self`

Comme pour de nombreux aspects du langage Rust, des caisses existent pour rendre la gestion des erreurs beaucoup plus facile et plus concise. Il y a toute une variété, mais l'un des plus utilisés est , qui fait tout le travail précédent pour vous, vous permettant d'écrire des erreurs comme celle-ci: `thiserror`

```
use thiserror::Error;  
#[derive(Error, Debug)]  
#[error("{message:} ({line:}, {column})")]  
pub struct JsonError {  
    message: String,  
    line: usize,  
    column: usize,  
}
```

La directive indique de générer le code affiché précédemment, ce qui peut économiser beaucoup de temps et d'efforts. #
`[derive(Error)] thiserror`

Pourquoi les résultats?

Maintenant, nous en savons assez pour comprendre ce que Rust veut dire en choisissant s plutôt que des exceptions. Voici les points clés de la conception: `Result`

- Rust exige que le programmeur prenne une sorte de décision et l'enregistre dans le code, à chaque point où une erreur pourrait se produire. C'est bien parce que sinon, il est facile de se tromper de gestion des erreurs par négligence.
- La décision la plus courante est de permettre aux erreurs de se propager, et c'est écrit avec un seul caractère, . Ainsi, la plomberie d'erreur n'encombre pas votre code comme elle le fait en C and Go. Pourtant, il est toujours visible: vous pouvez regarder un morceau de code et voir en un coup d'œil tous les endroits où les erreurs se propagent. ?

- Étant donné que la possibilité d'erreurs fait partie du type de retour de chaque fonction, il est clair quelles fonctions peuvent échouer et lesquelles ne le peuvent pas. Si vous modifiez une fonction pour qu'elle soit faillible, vous modifiez son type de retour, de sorte que le compilateur vous obligera à mettre à jour les utilisateurs en aval de cette fonction.
- Rust vérifie que les valeurs sont utilisées, de sorte que vous ne pouvez pas laisser accidentellement une erreur passer silencieusement (une erreur courante en C). `Result`
- Comme il s'agit d'un type de données comme un autre, il est facile de stocker les résultats de réussite et d'erreur dans la même collection. Cela facilite la modélisation du succès partiel. Par exemple, si vous écrivez un programme qui charge des millions d'enregistrements à partir d'un fichier texte et que vous avez besoin d'un moyen de faire face au résultat probable que la plupart réussiront, mais que certains échoueront, vous pouvez représenter cette situation en mémoire à l'aide d'un vecteur de `s.Result` `Result`

Le coût est que vous vous retrouverez à penser et à gérer les erreurs d'ingénierie plus dans Rust que dans d'autres langues. Comme dans beaucoup d'autres domaines, le point de vue de Rust sur la gestion des erreurs est un peu plus serré que ce à quoi vous êtes habitué. Pour la programmation de systèmes, cela en vaut la peine.

- 1** Vous devriez également envisager d'utiliser la caisse populaire, qui fournit des types d'erreur et de résultat très similaires à nos et , mais avec quelques fonctionnalités supplémentaires intéressantes. `anyhow GenericError GenericResult`

Chapitre 8. Caisses et modules

C'est une note dans un thème Rust: les programmeurs de systèmes peuvent avoir de belles choses.

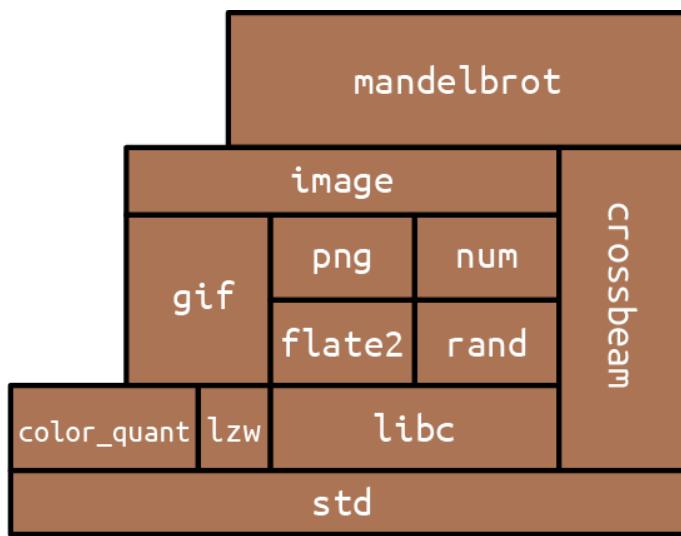
—Robert O'Callahan, [«Pensées aléatoires sur rust: crates.io et les IDE »](#)

Supposons que vous écrivez un programme qui simule la croissance des fougères, à partir du niveau des cellules individuelles. Votre programme, comme une fougère, commencera très simplement, avec tout le code, peut-être, dans un seul fichier, juste la spore d'une idée. Au fur et à mesure qu'il grandit, il commencera à avoir une structure interne. Différentes pièces auront des objectifs différents. Il se ramifiera en plusieurs fichiers. Il peut couvrir toute une arborescence de répertoires. Avec le temps, il peut devenir une partie importante de tout un écosystème logiciel. Pour tout programme qui se développe au-delà de quelques structures de données ou de quelques centaines de lignes, une certaine organisation est nécessaire.

Ce chapitre couvre les fonctionnalités de Rust qui aident à garder votre programme organisé: caisses et modules. Nous couvrirons également d'autres sujets liés à la structure et à la distribution d'une caisse Rust, notamment comment documenter et tester le code Rust, comment faire taire les avertissements indésirables du compilateur, comment utiliser Cargo pour gérer les dépendances et le contrôle de version du projet, comment publier des bibliothèques open source sur le référentiel public de caisses de Rust, crates.io, comment Rust évolue à travers les éditions de langage, et plus encore, en utilisant le simulateur de fougère comme exemple de course.

Caisses

Les programmes de rouille sont *faits de caisses*. Chaque caisse est une unité complète et cohésive : tout le code source d'une seule bibliothèque ou exécutable, ainsi que tous les tests, exemples, outils, configurations et autres fichiers indésirables associés. Pour votre simulateur de fougère, vous pouvez utiliser des bibliothèques tierces pour les graphiques 3D, la bioinformatique, le calcul parallèle, etc. Ces bibliothèques sont distribuées sous forme de caisses (voir [Figure 8-1](#)).



Graphique 8-1. Une caisse et ses dépendances

Le moyen le plus simple de voir ce que sont les caisses et comment elles fonctionnent ensemble est de les utiliser avec l'indicateur pour créer un projet existant qui a des dépendances. Nous l'avons fait en utilisant « [Un programme Mandelbrot simultané](#) » comme exemple. Les résultats sont présentés ici : cargo build --verbose

```

$ cd mandelbrot
$ cargo clean      # delete previously compiled code
$ cargo build --verbose
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Downloading autocfg v1.0.0
    Downloading semver-parser v0.7.0
    Downloading gif v0.9.0
    Downloading png v0.7.0

    ...
    ... (downloading and compiling many more crates)

Compiling jpeg-decoder v0.1.18
    Running `rustc
        --crate-name jpeg_decoder
        --crate-type lib
        ...
        --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
        ...
    Compiling image v0.13.0
    Running `rustc
        --crate-name image
        --crate-type lib
        ...
        --extern byteorder=.../libbyteorder-29efdd0b59c6f920.rmeta
        --extern gif=.../libgif-a7006d35f1b58927.rmeta
        --extern jpeg_decoder=.../libjpeg_decoder-5c10558d0d57d300.rmeta
Compiling mandelbrot v0.1.0 (/tmp/rustbook-test-files/mandelbrot)
    Running `rustc

```

```
--edition=2021
--crate-name mandelbrot
--crate-type bin
...
--extern crossbeam=.../libcrossbeam-f87b4b3d3284acc2.rlib
--extern image=.../libimage-b5737c12bd641c43.rlib
--extern num=.../libnum-1974e9a1dc582ba7.rlib -C link-arg=-fuse-1
Finished dev [unoptimized + debuginfo] target(s) in 16.94s
$
```

Nous avons reformaté les lignes de commande pour plus de lisibilité, et nous avons supprimé beaucoup d'options du compilateur qui ne sont pas pertinentes pour notre discussion, en les remplaçant par des points de suspension (). `rustc ...`

Vous vous souviendrez peut-être qu'au moment où nous avons terminé, la `main.rs` du programme Mandelbrot contenait plusieurs déclarations pour les articles provenant d'autres caisses: `use`

```
use num::Complex;
// ...
use image::ColorType;
use image::png::PNGEncoder;
```

Nous avons également spécifié dans notre fichier `Cargo.toml` quelle version de chaque caisse nous voulions:

```
[dependencies]
num = "0.4"
image = "0.13"
crossbeam = "0.8"
```

Le mot *dépendances* ici signifie simplement d'autres caisses que ce projet utilise : du code dont nous dépendons. Nous avons trouvé ces caisses sur [crates.io](#), le site de la communauté Rust pour les caisses open source. Par exemple, nous avons découvert la bibliothèque en allant sur crates.io et en recherchant une bibliothèque d'images. La page de chaque caisse sur crates.io affiche son fichier `README.md` et des liens vers la documentation et la source, ainsi qu'une ligne de configuration comme celle que vous pouvez copier et ajouter à votre `Cargo.toml`. Les numéros de version indiqués ici sont simplement les dernières versions de ces trois paquets au moment où nous avons écrit le programme. `image image = "0.13"`

La transcription de Cargo raconte l'histoire de la façon dont cette information est utilisée. Lorsque nous exécutons , Cargo commence par

télécharger le code source pour les versions spécifiées de ces caisses à partir de crates.io. Ensuite, il lit les fichiers *Cargo.toml* de ces caisses, télécharge *leurs* dépendances, etc. de manière récursive. Par exemple, le code source de la version 0.13.0 de la caisse contient un fichier *Cargo.toml* qui inclut ceci : `cargo build image`

```
[dependencies]
byteorder = "1.0.0"
num-iter = "0.1.32"
num-rational = "0.1.32"
num-trait = "0.1.32"
enum_primitive = "0.1.0"
```

Voyant cela, Cargo sait qu'avant de pouvoir utiliser, il doit également aller chercher ces caisses. Plus tard, nous verrons comment dire à Cargo de récupérer le code source d'un référentiel Git ou du système de fichiers local plutôt que de crates.io. `image`

Puisque dépend indirectement de ces caisses, par son utilisation de la caisse, nous les appelons dépendances *transitives* de . La collection de toutes ces relations de dépendance, qui indique à Cargo tout ce qu'elle doit savoir sur les caisses à construire et dans quel ordre, est connue sous le nom de *graphique de dépendance* de la caisse. La gestion automatique par Cargo du graphique de dépendance et des dépendances transitives est une énorme victoire en termes de temps et d'efforts du programmeur. `mandelbrot image mandelbrot`

Une fois qu'il a le code source, Cargo compile toutes les caisses. Il exécute , le compilateur Rust, une fois pour chaque caisse dans le graphique de dépendance du projet. Lors de la compilation de bibliothèques, Cargo utilise l'option. Cela indique de ne pas rechercher une fonction, mais plutôt de produire un fichier *.rlib* contenant du code compilé qui peut être utilisé pour créer des fichiers binaires et d'autres fichiers `.rlib. rustc --crate-type lib rustc main()`

Lors de la compilation d'un programme, Cargo utilise , et le résultat est un exécutable binaire pour la plate-forme cible: *mandelbrot.exe* sur Windows, par exemple. `--crate-type bin`

À chaque commande, Cargo passe les options, en donnant le nom de fichier de chaque bibliothèque que la caisse utilisera. De cette façon, lorsqu'il voit une ligne de code comme , il peut comprendre que c'est le nom d'une autre caisse, et grâce à Cargo, il sait où trouver cette caisse compilée sur le disque. Le compilateur Rust a besoin d'accéder à ces fichiers *.rlib* car ils

contiennent le code compilé de la bibliothèque. Rust liera statiquement ce code à l'exécutable final. Le fichier `.rlib` contient également des informations de type afin que Rust puisse vérifier que les fonctionnalités de bibliothèque que nous utilisons dans notre code existent réellement dans la caisse et que nous les utilisons correctement. Il contient également une copie des fonctions en ligne publiques, des génériques et des macros de la caisse, des fonctionnalités qui ne peuvent pas être entièrement compilées en code machine tant que Rust ne voit pas comment nous les utilisons.

```
rustc --extern rustc use  
image::png::PNGEncoder image
```

`cargo build` prend en charge toutes sortes d'options, dont la plupart dépassent le cadre de ce livre, mais nous en mentionnerons une ici: produit une version optimisée. Les versions s'exécutent plus rapidement, mais leur compilation prend plus de temps, elles ne vérifient pas le débordement d'entiers, elles ignorent les assertions et les traces de pile qu'elles génèrent en cas de panique sont généralement moins fiables.

```
cargo build --release debug_assert!()
```

Éditions

Rust a des garanties de compatibilité extrêmement fortes. Tout code compilé sur Rust 1.0 doit compiler tout aussi bien sur Rust 1.50 ou, s'il est jamais publié, Rust 1.900.

Mais parfois, il existe des propositions convaincantes d'extensions du langage qui empêcheraient la compilation d'un code plus ancien. Par exemple, après de longues discussions, Rust a opté pour une syntaxe pour la prise en charge de la programmation asynchrone qui réutilise les identificateurs et les mots-clés (voir [le chapitre 20](#)). Mais ce changement de langage briserait tout code existant qui utilise ou comme nom d'une variable.

```
async await async await
```

Pour évoluer sans casser le code existant, Rust utilise des *éditions*. L'édition 2015 de Rust est compatible avec Rust 1.0. L'édition 2018 a changé et est devenue des mots-clés et a rationalisé le système de modules, tandis que l'édition 2021 a amélioré l'ergonomie des tableaux et a rendu certaines définitions de bibliothèque largement utilisées disponibles partout par défaut. Il s'agissait d'améliorations importantes apportées au langage, mais elles auraient brisé le code existant. Pour éviter cela, chaque caisse indique dans quelle édition de Rust elle est écrite avec une ligne comme celle-ci dans la section au sommet de son fichier

```
Cargo.toml: async await [package]
```

```
edition = "2021"
```

Si ce mot-clé est absent, l'édition 2015 est supposée, de sorte que les vieilles caisses n'ont pas à changer du tout. Mais si vous souhaitez utiliser des fonctions asynchrones ou le nouveau système de modules, vous aurez besoin ou plus tard dans votre fichier *Cargo.toml*.

```
edition = "2018"
```

Rust promet que le compilateur acceptera toujours toutes les éditions existantes du langage, et les programmes peuvent mélanger librement des caisses écrites dans différentes éditions. Il est même acceptable qu'une caisse de l'édition 2015 dépende d'une caisse de l'édition 2021. En d'autres termes, l'édition d'une caisse n'affecte que la façon dont son code source est interprété; les distinctions d'édition ont disparu au moment où le code a été compilé. Cela signifie qu'il n'y a aucune pression pour mettre à jour les vieilles caisses juste pour continuer à participer à l'écosystème moderne de Rust. De même, il n'y a aucune pression pour garder votre caisse sur une édition plus ancienne afin d'éviter de déranger ses utilisateurs. Vous n'avez besoin de changer d'édition que lorsque vous souhaitez utiliser de nouvelles fonctionnalités de langage dans votre propre code.

Les éditions ne sortent pas chaque année, seulement lorsque le projet Rust en décide une. Par exemple, il n'y a pas d'édition 2020. La définition de `sur` provoque une erreur. Le [Guide de l'édition Rust](#) couvre les modifications introduites dans chaque édition et fournit un bon contexte sur le système d'édition.

```
edition = "2020"
```

C'est presque toujours une bonne idée d'utiliser la dernière édition, en particulier pour le nouveau code. crée de nouveaux projets sur la dernière édition par défaut. Ce livre utilise l'édition 2021 tout au long.

```
cargo new
```

Si vous avez une caisse écrite dans une ancienne édition de Rust, la commande peut vous aider à mettre automatiquement à niveau votre code vers la nouvelle édition. Le Guide de l'édition Rust explique la commande en détail.

```
cargo fix cargo fix
```

Créer des profils

Il existe plusieurs paramètres de configuration que vous pouvez placer dans votre fichier *Cargo.toml* qui affectent les lignes de commande générées ([Tableau 8-1](#)).

```
rustc cargo
```

Tableau 8-1. Sections de configuration de *Cargo.toml*

Ligne de commande	Section <i>Cargo.toml</i> utilisée
<code>cargo build</code>	<code>[profile.dev]</code>
<code>cargo build --release</code>	<code>[profile.release]</code>
<code>cargo test</code>	<code>[profile.test]</code>

Les valeurs par défaut sont généralement correctes, mais une exception que nous avons trouvée est lorsque vous souhaitez utiliser un profileur, un outil qui mesure où votre programme passe son temps CPU. Pour obtenir les meilleures données d'un profileur, vous avez besoin à la fois d'optimisations (généralement activées uniquement dans les versions de version) et de symboles de débogage (généralement activés uniquement dans les versions de débogage). Pour activer les deux, ajoutez ceci à votre *Cargo.toml*:

```
[profile.release]
debug = true # enable debug symbols in release builds
```

Le paramètre contrôle l'option de . Avec cette configuration, lorsque vous tapez , vous obtiendrez un binaire avec des symboles de débogage. Les paramètres d'optimisation ne sont pas affectés. `debug -g rustc cargo build --release`

[La documentation Cargo](#) répertorie de nombreux autres paramètres que vous pouvez ajuster dans *Cargo.toml*.

Modules

Alors que les caisses concernent le partage de code entre les projets, les *modules* concernent l'organisation *du code au sein d'un projet*. Ils agissent comme des espaces de noms rust, des conteneurs pour les fonctions, les types, les constantes, etc. qui composent votre programme ou bibliothèque Rust. Un module ressemble à ceci :

```
mod spores {
    use cells::{Cell, Gene};

    /// A cell made by an adult fern. It disperses on the wind as part of
    /// the fern life cycle. A spore grows into a prothallus -- a whole
    /// separate organism, up to 5mm across -- which produces the zygote
```

```

    /// that grows into a new fern. (Plant sex is complicated.)
pub struct Spore {
    ...
}

/// Simulate the production of a spore by meiosis.
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}

/// Extract the genes in a particular spore.
pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
    ...
}

/// Mix genes to prepare for meiosis (part of interphase).
fn recombine(parent: &mut Cell) {
    ...
}

...
}

```

Un module est une collection *d'éléments*, nommés fonctions telles que la structure et les deux fonctions de cet exemple. Le mot-clé rend un élément public, de sorte qu'il est accessible depuis l'extérieur du module. `Spore pub`

Une fonction est marquée `,`, ce qui signifie qu'elle est disponible n'importe où à l'intérieur de cette caisse, mais n'est pas exposée dans le cadre de l'interface externe. Il ne peut pas être utilisé par d'autres caisses, et il n'apparaîtra pas dans la documentation de cette caisse. `pub(crate)`

Tout ce qui n'est pas marqué est privé et ne peut être utilisé que dans le même module dans lequel il est défini, ou dans n'importe quel module enfant : `pub`

```

let s = spores::produce_spore(&mut factory); // ok

spores::recombine(&mut cell); // error: `recombine` is private

```

Marquer un élément comme on l'appelle souvent « exporter » cet élément. `pub`

Le reste de cette section couvre les détails que vous devez connaître pour utiliser pleinement les modules:

- Nous montrons comment imbriquer des modules et les répartir sur différents fichiers et répertoires, si nécessaire.
- Nous expliquons la syntaxe de chemin utilisée par Rust pour faire référence aux éléments d'autres modules et montrons comment importer des éléments afin que vous puissiez les utiliser sans avoir à écrire leurs chemins complets.
- Nous abordons le contrôle à grain fin de Rust pour les champs de struct.
- Nous introduisons des modules *de prélude*, qui réduisent le standard en rassemblant des importations communes dont presque tous les utilisateurs auront besoin.
- Nous présentons des *constantes* et des *statiques*, deux façons de définir des valeurs nommées, pour plus de clarté et de cohérence.

Modules imbriqués

Les modules peuvent s'imbriquer, et il est assez courant de voir un module qui n'est qu'une collection de sous-modules :

```
mod plant_structures {
    pub mod roots {
        ...
    }
    pub mod stems {
        ...
    }
    pub mod leaves {
        ...
    }
}
```

Si vous souhaitez qu'un élément d'un module imbriqué soit visible par d'autres caisses, veillez à le marquer, ainsi que *tous les modules qui l'en-tourent*, comme publics. Sinon, vous pouvez voir un avertissement comme celui-ci:

```
warning: function is never used: `is_square`
|
23 | /           pub fn is_square(root: &Root) -> bool {
24 | |             root.cross_section_shape().is_square()
25 | |         }
| |_____ ^
|
```

Peut-être que cette fonction est vraiment du code mort pour le moment. Mais si vous aviez l'intention de l'utiliser dans d'autres caisses, Rust vous fait savoir qu'il n'est pas réellement visible pour eux. Vous devez vous assurer que ses modules d'enceinte sont tous aussi bien. pub

Il est également possible de spécifier , rendant un élément visible uniquement par le module parent, et , ce qui le rend visible dans un module parent spécifique et ses descendants. Ceci est particulièrement utile avec les modules profondément imbriqués : pub(super) pub(in <path>)

```
mod plant_structures {
    pub mod roots {
        pub mod products {
            pub(in crate::plant_structures::roots) struct Cytokinin {
                ...
            }
        }
    }

    use products::Cytokinin; // ok: in `roots` module
}

use roots::products::Cytokinin; // error: `Cytokinin` is private
// error: `Cytokinin` is private
use plant_structures::roots::products::Cytokinin;
```

De cette façon, nous pourrions écrire un programme entier, avec une énorme quantité de code et toute une hiérarchie de modules, liés de la manière que nous voulions, le tout dans un seul fichier source.

Travailler de cette façon est pénible, cependant, alors il y a une alternative.

Modules dans des fichiers séparés

Un module peut également être écrit comme ceci:

```
mod spores;
```

Plus tôt, nous avons inclus le corps du module, enveloppé dans des accolades bouclées. Ici, nous disons plutôt au compilateur Rust que le module vit dans un fichier séparé, appelé `spores.rs`: `spores spores`

```
// spores.rs
```

```

/// A cell made by an adult fern...
pub struct Spore {
    ...
}

/// Simulate the production of a spore by meiosis.
pub fn produce_spore(factory: &mut Sporangium) -> Spore {
    ...
}

/// Extract the genes in a particular spore.
pub(crate) fn genes(spore: &Spore) -> Vec<Gene> {
    ...
}

/// Mix genes to prepare for meiosis (part of interphase).
fn recombine(parent: &mut Cell) {
    ...
}

```

spores.rs contient uniquement les éléments qui composent le module. Il n'a besoin d'aucune sorte de passe-partout pour déclarer qu'il s'agit d'un module.

L'emplacement du code est la *seule* différence entre ce module et la version que nous avons montrée dans la section précédente. Les règles sur ce qui est public et ce qui est privé sont exactement les mêmes dans les deux cas. Et Rust ne compile jamais les modules séparément, même s'ils sont dans des fichiers séparés : lorsque vous construisez une caisse Rust, vous recompilez tous ses modules. *spores*

Un module peut avoir son propre répertoire. Lorsque Rust voit , il vérifie à la fois *les spores.rs* et *les spores / mod.rs*; si aucun des deux fichiers n'existe, ou si les deux existent, c'est une erreur. Pour cet exemple, nous avons utilisé *spores.rs*, car le module n'avait pas de sous-modules. Mais considérez le module que nous avons écrit plus tôt. Si nous décidons de diviser ce module et ses trois sous-modules dans leurs propres fichiers, le projet résultant ressemblerait à ceci:

```
mod
spores; spores plant_structures
```

```

fern_sim/
└── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/

```

```
└── mod.rs
└── leaves.rs
└── roots.rs
└── stems.rs
```

Dans *main.rs*, nous déclarons le module: `plant_structures`

```
pub mod plant_structures;
```

Cela amène Rust à charger *plant_structures/mod.rs*, qui déclare les trois sous-modules :

```
// in plant_structures/mod.rs
pub mod roots;
pub mod stems;
pub mod leaves;
```

Le contenu de ces trois modules est stocké dans des fichiers distincts nommés *leaves.rs*, *roots.rs* et *stems.rs*, situés à côté de *mod.rs* dans le répertoire *plant_structures*.

Il est également possible d'utiliser un fichier et un répertoire portant le même nom pour constituer un module. Par exemple, si nécessaire pour inclure des modules appelés *et*, nous pourrions choisir de conserver dans *plant_structures/stems.rs* et d'ajouter un répertoire

```
stems: stems xylem phloem stems
```

```
fern_sim/
└── Cargo.toml
└── src/
    ├── main.rs
    ├── spores.rs
    └── plant_structures/
        ├── mod.rs
        ├── leaves.rs
        ├── roots.rs
        ├── stems/
            ├── phloem.rs
            └── xylem.rs
        └── stems.rs
```

Ensuite, dans *stems.rs*, nous déclarons les deux nouveaux sous-modules :

```
// in plant_structures/stems.rs
pub mod xylem;
pub mod phloem;
```

Ces trois options (modules dans leur propre fichier, modules dans leur propre répertoire avec un `mod.rs` et modules dans leur propre fichier avec un répertoire supplémentaire contenant des sous-modules) donnent au système de modules suffisamment de flexibilité pour prendre en charge presque toutes les structures de projet que vous pourriez souhaiter.

Chemins d'accès et importations

L'opérateur `:` est utilisé pour accéder aux fonctionnalités d'un module. Le code n'importe où dans votre projet peut faire référence à n'importe quelle fonctionnalité de bibliothèque standard en écrivant son chemin d'accès `::::`:

```
if s1 > s2 {  
    std::mem::swap(&mut s1, &mut s2);  
}
```

`std` est le nom de la bibliothèque standard. Le chemin fait référence au module de niveau supérieur de la bibliothèque standard. `mem` est un sous-module de la bibliothèque standard et est une fonction publique de ce module. `std std::mem std::mem::swap`

Vous pouvez écrire tout votre code de cette façon, en épelant et chaque fois que vous voulez un cercle ou un dictionnaire, mais ce serait fastidieux à taper et difficile à lire. L'alternative consiste à *importer des fonctionnalités* dans les modules où elles sont utilisées

```
: std::f64::consts::PI std::collections::HashMap::new
```

```
use std::mem;  
  
if s1 > s2 {  
    mem::swap(&mut s1, &mut s2);  
}
```

La déclaration fait que le nom est un alias local pour l'ensemble du bloc ou du module englobant. `use mem std::mem`

Nous pourrions écrire pour importer la fonction elle-même au lieu du module. Cependant, ce que nous avons fait précédemment est généralement considéré comme le meilleur style: importez des types, des traits et des modules (comme `std::mem`) puis utilisez des chemins relatifs pour accéder aux fonctions, constantes et autres membres à l'intérieur. `use std::mem::swap; swap mem std::mem`

Plusieurs noms peuvent être importés à la fois :

```
use std::collections::{HashMap, HashSet}; // import both

use std::fs::{self, File}; // import both `std::fs` and `std::fs::File`.

use std::io::prelude::*;

// import everything
```

Ceci est juste un raccourci pour écrire toutes les importations individuelles:

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

use std::fs;
use std::fs::File;

// all the public items in std::io::prelude:
use std::io::prelude::Read;
use std::io::prelude::Write;
use std::io::prelude::BufRead;
use std::io::prelude::Seek;
```

Vous pouvez l'utiliser pour importer un élément mais lui donner un nom différent localement : as

```
use std::io::Result as IOResult;

// This return type is just another way to write `std::io::Result<()>`:
fn save_spore(spore: &Spore) -> IOResult<()>
    ...


```

Les modules *n'héritent pas* automatiquement des noms de leurs modules parents. Par exemple, supposons que nous ayons ceci dans nos *protéines/mod.rs* :

```
// proteins/mod.rs
pub enum AminoAcid { ... }
pub mod synthesis;
```

Ensuite, le code dans *synthesis.rs* ne voit pas automatiquement le type :AminoAcid

```
// proteins/synthesis.rs
pub fn synthesize(seq: &[AminoAcid]) // error: can't find type `AminoAcid
    ...
}
```

Au lieu de cela, chaque module commence par une ardoise vierge et doit importer les noms qu'il utilise :

```
// proteins/synthesis.rs
use super::AminoAcid; // explicitly import from parent

pub fn synthesize(seq: &[AminoAcid]) // ok
    ...

```

Par défaut, les chemins sont relatifs au module actuel :

```
// in proteins/mod.rs

// import from a submodule
use synthesis::synthesize;
```

`self` est également un synonyme du module actuel, nous pourrions donc écrire soit :

```
// in proteins/mod.rs

// import names from an enum,
// so we can write `Lys` for lysine, rather than `AminoAcid::Lys`
use self::AminoAcid::*;


```

ou simplement :

```
// in proteins/mod.rs

use AminoAcid::*;


```

(L'exemple ici est, bien sûr, un écart par rapport à la règle de style que nous avons mentionnée précédemment concernant uniquement l'importation de types, de traits et de modules. Si notre programme comprend de longues séquences d'acides aminés, cela est justifié en vertu de la sixième règle d'Orwell: « Enfreignez l'une de ces règles plus tôt que de dire quoi que ce soit de carrément barbare. ») AminoAcid

Les mots-clés `super` et `crate` ont une signification particulière dans les chemins: `super` se réfère au module parent, et `crate` à la caisse contenant le module actuel. `super crate` `super crate`

L'utilisation de chemins relatifs à la racine de la caisse plutôt qu'au module actuel facilite le déplacement du code dans le projet, car toutes les im-

portations ne se briseront pas si le chemin du module actuel change. Par exemple, nous pourrions écrire `synthesis.rs` en utilisant : `crate`

```
// proteins/synthesis.rs
use crate:::proteins::AminoAcid; // explicitly import relative to crate root

pub fn synthesize(seq: &[AminoAcid]) // ok
    ...
}
```

Les sous-modules peuvent accéder aux éléments privés de leurs modules parents avec `.use super::*`

Si vous avez un module portant le même nom qu'une caisse que vous utilisez, il faut faire attention à leur contenu. Par exemple, si votre programme répertorie la caisse comme une dépendance dans son fichier `Cargo.toml`, mais possède également un module nommé `image`, alors les chemins commençant par sont ambigus : `image image image`

```
mod image {
    pub struct Sampler {
        ...
    }
}

// error: Does this refer to our `image` module, or the `image` crate?
use image::Pixels;
```

Même si le module n'a pas de type, l'ambiguïté est toujours considérée comme une erreur: il serait déroutant si l'ajout d'une telle définition plus tard pouvait changer silencieusement ce à quoi se réfèrent les chemins ailleurs dans le programme. `image Pixels`

Pour résoudre l'ambiguïté, Rust a un type spécial de chemin appelé *chemin absolu*, commençant par `:`, qui se réfère toujours à une caisse externe. Pour faire référence au type dans la caisse, vous pouvez écrire `:: Pixels image`

```
use ::image::Pixels;           // the `image` crate's `Pixels`
```

Pour faire référence au type de votre propre module, vous pouvez écrire `: Sampler`

```
use self::image::Sampler;     // the `image` module's `Sampler`
```

Les modules ne sont pas la même chose que les fichiers, mais il existe une analogie naturelle entre les modules et les fichiers et répertoires d'un système de fichiers Unix. Le mot-clé crée des alias, tout comme la commande crée des liens. Les chemins, comme les noms de fichiers, se présentent sous des formes absolues et relatives. et sont comme les répertoires spéciaux. `use ln self super . . .`

Le Prélude Standard

Nous avons dit tout à l'heure que chaque module commence par une « ardoise vierge », en ce qui concerne les noms importés. Mais l'ardoise n'est pas *complètement* vierge.

D'une part, la bibliothèque standard est automatiquement liée à chaque projet. Cela signifie que vous pouvez toujours utiliser ou faire référence à des éléments par leur nom, comme inline dans votre code. De plus, quelques noms particulièrement pratiques, comme `et`, sont inclus dans le *prélude standard* et automatiquement importés. Rust se comporte comme si chaque module, y compris le module racine, commençait par l'importation suivante : `std use`

```
std::whatever std std::mem::swap() Vec Result
```

```
use std::prelude::v1::*;


```

Le prélude standard contient quelques dizaines de traits et de types couramment utilisés.

Dans [le chapitre 2](#), nous avons mentionné que les bibliothèques fournissent parfois des modules nommés `prélude`. Mais c'est le seul prélude jamais importé automatiquement. Nommer un module n'est qu'une convention qui indique aux utilisateurs qu'il est destiné à être importé à l'aide de `.std::prelude::v1 prelude *`

Utilisation de Déclarations pub

Même si les déclarations ne sont que des alias, elles peuvent être publiques : `use`

```
// in plant_structures/mod.rs
...
pub use self::leaves::Leaf;
pub use self::roots::Root;
```

Cela signifie que et sont des éléments publics du module. Ce sont toujours de simples alias pour et

```
.Leaf Root plant_structures plant_structures::leaves::Leaf p  
lant_structures::roots::Root
```

Le prélude standard est écrit comme une telle série d'importations. pub

Faire pub Struct Fields

Un module peut inclure des types de structure définis par l'utilisateur, introduits à l'aide du mot-clé. Nous les couvrons en détail dans [le chapitre 9](#), mais c'est un bon point pour mentionner comment les modules interagissent avec la visibilité des champs de structure. struct

Une structure simple ressemble à ceci:

```
pub struct Fern {  
    pub roots: RootSet,  
    pub stems: StemSet  
}
```

Les champs d'une struct, même les champs privés, sont accessibles dans tout le module où la struct est déclarée, et ses sous-modules. En dehors du module, seuls les champs publics sont accessibles.

Il s'avère que l'application du contrôle d'accès par module, plutôt que par classe comme en Java ou C++, est étonnamment utile pour la conception de logiciels. Il réduit les méthodes standard « getter » et « setter », et il élimine en grande partie le besoin de tout ce qui ressemble à des déclarations C++. Un seul module peut définir plusieurs types qui travaillent en étroite collaboration, tels que peut-être et , accéder aux champs privés de l'autre selon les besoins, tout en cachant ces détails d'implémentation au reste de votre

programme. friend frond::LeafMap frond::LeafMapIter

Statique et constantes

Outre les fonctions, les types et les modules imbriqués, les modules peuvent également définir des *constantes* et *des statiques*.

Le mot-clé introduit une constante. La syntaxe est exactement comme, sauf qu'elle peut être marquée , et le type est requis. En outre, sont conventionnels pour les constantes: const let pub UPPERCASE_NAMES

```
pub const ROOM_TEMPERATURE: f64 = 20.0; // degrees Celsius
```

Le mot-clé introduit un élément statique, ce qui est presque la même chose: `static`

```
pub static ROOM_TEMPERATURE: f64 = 68.0; // degrees Fahrenheit
```

Une constante est un peu comme un C++ : la valeur est compilée dans votre code à chaque endroit où elle est utilisée. Une statique est une variable qui est configurée avant que votre programme ne commence à s'exécuter et qui dure jusqu'à ce qu'il se ferme. Utilisez des constantes pour les nombres magiques et les chaînes dans votre code. Utilisez la statique pour de plus grandes quantités de données, ou chaque fois que vous devez emprunter une référence à la valeur constante. `#define`

Il n'y a pas de constantes. La statique peut être marquée, mais comme discuté au [chapitre 5](#), Rust n'a aucun moyen d'appliquer ses règles sur l'accès exclusif sur la statique. Ils sont donc intrinsèquement non threadsafe, et le code safe ne peut pas les utiliser du tout : `mut` `mut` `mut`

```
static mut PACKETS_SERVED: usize = 0;

println!("{} served", PACKETS_SERVED); // error: use of mutable static
```

La rouille décourage l'état mutable global. Pour une discussion sur les alternatives, voir [« Variables globales »](#).

Transformer un programme en bibliothèque

Lorsque votre simulateur de fougère commence à décoller, vous décidez que vous avez besoin de plus d'un seul programme. Supposons que vous ayez un programme de ligne de commande qui exécute la simulation et enregistre les résultats dans un fichier. Maintenant, vous voulez écrire d'autres programmes pour effectuer une analyse scientifique des résultats enregistrés, afficher des rendus 3D des plantes en croissance en temps réel, rendre des images photoréalistes, etc. Tous ces programmes doivent partager le code de simulation de fougère de base. Vous devez créer une bibliothèque.

La première étape consiste à factoriser votre projet existant en deux parties : une caisse de bibliothèque, qui contient tout le code partagé, et un exécutable, qui contient le code nécessaire uniquement pour votre programme de ligne de commande existant.

Pour montrer comment vous pouvez le faire, utilisons un exemple de programme grossièrement simplifié:

```
struct Fern {
    size: f64,
    growth_rate: f64
}

impl Fern {
    /// Simulate a fern growing for one day.
    fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// Run a fern simulation for some number of days.
fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

Nous supposerons que ce programme a un fichier *Cargo.toml* trivial:

```
[package]
name = "fern_sim"
version = "0.1.0"
authors = [ "You <you@example.com>" ]
edition = "2021"
```

Transformer ce programme en bibliothèque est facile. Voici les étapes :

1. Renommez le fichier *src/main.rs* en *src/lib.rs*.
2. Ajoutez le mot-clé aux éléments *dans src/lib.rs* qui seront des fonctionnalités publiques de notre bibliothèque. `pub`
3. Déplacez la fonction vers un fichier temporaire quelque part. Nous y reviendrons dans une minute. `main`

Le fichier `src/lib.rs` résultant ressemble à ceci :

```
pub struct Fern {
    pub size: f64,
    pub growth_rate: f64
}

impl Fern {
    /// Simulate a fern growing for one day.
    pub fn grow(&mut self) {
        self.size *= 1.0 + self.growth_rate;
    }
}

/// Run a fern simulation for some number of days.
pub fn run_simulation(fern: &mut Fern, days: usize) {
    for _ in 0 .. days {
        fern.grow();
    }
}
```

Notez que nous n'avons pas eu besoin de changer quoi que ce soit dans `Cargo.toml`. En effet, notre fichier `Cargo.toml` minimal laisse Cargo à son comportement par défaut. Par défaut, examine les fichiers dans notre répertoire source et détermine ce qu'il faut construire. Lorsqu'il voit le fichier `src/lib.rs`, il sait construire une bibliothèque. `cargo build`

Le code dans `src/lib.rs` forme le *module racine* de la bibliothèque. Les autres caisses qui utilisent notre bibliothèque ne peuvent accéder qu'aux éléments publics de ce module racine.

Le répertoire src/bin

Faire fonctionner à nouveau le programme de ligne de commande d'origine est également simple: Cargo dispose d'un support intégré pour les petits programmes qui vivent dans la même caisse qu'une bibliothèque. `fern_sim`

En fait, Cargo lui-même est écrit de cette façon. La majeure partie du code se trouve dans une bibliothèque Rust. Le programme de ligne de commande que nous avons utilisé tout au long de ce livre est un programme d'emballage mince qui appelle la bibliothèque pour tout le travail lourd. La bibliothèque et le programme de ligne de commande vivent dans le même référentiel source, `cargo`

Nous pouvons également garder notre programme et notre bibliothèque dans la même caisse. Placez ce code dans un fichier nommé `src/bin/efern.rs`:

```
use fern_sim:::{Fern, run_simulation};

fn main() {
    let mut fern = Fern {
        size: 1.0,
        growth_rate: 0.001
    };
    run_simulation(&mut fern, 1000);
    println!("final fern size: {}", fern.size);
}
```

La fonction est celle que nous avons mise de côté plus tôt. Nous avons ajouté une déclaration pour certains articles de la caisse et . En d'autres termes, nous utilisons cette caisse comme

```
bibliothèque. main use fern_sim Fern run_simulation
```

Parce que nous avons mis ce fichier dans `src/bin`, Cargo compilera à la fois la bibliothèque et ce programme la prochaine fois que nous exécuterons . Nous pouvons exécuter le programme en utilisant . Voici à quoi cela ressemble, en utilisant pour afficher les commandes que Cargo est en cours d'exécution: `fern_sim cargo build efern cargo run --bin efern --verbose`

```
$ cargo build --verbose
Compiling fern_sim v0.1.0 (file:///.../fern_sim)
  Running `rustc src/lib.rs --crate-name fern_sim --crate-type lib ...
  Running `rustc src/bin/efern.rs --crate-name efern --crate-type bin
$ cargo run --bin efern --verbose
  Fresh fern_sim v0.1.0 (file:///.../fern_sim)
  Running `target/debug/efern`
final fern size: 2.7169239322355985
```

Nous n'avons toujours pas eu à apporter de modifications à `Cargo.toml`, car, encore une fois, la valeur par défaut de Cargo est de regarder vos fichiers sources et de comprendre les choses. Il traite automatiquement les fichiers `.rs` dans `src/bin` comme des programmes supplémentaires à construire.

Nous pouvons également créer des programmes plus volumineux dans le répertoire `src/bin` à l'aide de sous-répertoires. Supposons que nous voulions fournir un deuxième programme qui dessine une fougère à

l'écran, mais le code de dessin est grand et modulaire, il appartient donc à son propre fichier. Nous pouvons donner au deuxième programme son propre sous-répertoire:

```
fern_sim/
└── Cargo.toml
└── src/
    └── bin/
        ├── efern.rs
        └── draw_fern/
            ├── main.rs
            └── draw.rs
```

Cela a l'avantage de permettre aux binaires plus volumineux d'avoir leurs propres sous-modules sans encombrer le code de la bibliothèque ou le répertoire *src/bin*.

Bien sûr, maintenant que c'est une bibliothèque, nous avons aussi une autre option. Nous aurions pu mettre ce programme dans son propre projet isolé, dans un répertoire complètement séparé, avec sa propre liste *Cargo.toml* comme dépendance: `fern_sim fern_sim`

```
[dependencies]
fern_sim = { path = "../fern_sim" }
```

C'est peut-être ce que vous ferez pour d'autres programmes de simulation de fougères plus tard. Le répertoire *src/bin* est parfait pour les programmes simples comme `efern draw_fern`

Attributs

Tout élément d'un programme Rust peut être décoré avec des *attributs*. Les attributs sont la syntaxe fourre-tout de Rust pour écrire diverses instructions et conseils au compilateur. Par exemple, supposons que vous recevez cet avertissement :

```
libgit2.rs: warning: type `git_revspec` should have a camel case name
such as `GitRevspec`, #[warn(non_camel_case_types)] on by default
```

Mais vous avez choisi ce nom pour une raison, et vous aimeriez que Rust se taise à ce sujet. Vous pouvez désactiver l'avertissement en ajoutant un attribut sur le type : `#[allow]`

```
#[allow(non_camel_case_types)]  
pub struct git_revspec {  
    ...  
}
```

La compilation conditionnelle est une autre fonctionnalité écrite à l'aide d'un attribut, à savoir : `#[cfg]`

```
// Only include this module in the project if we're building for Android.  
#[cfg(target_os = "android")]  
mod mobile;
```

La syntaxe complète de est spécifiée dans la [référence Rust](#) ; les options les plus couramment utilisées sont énumérées dans [le tableau 8-2.](#) `#[cfg]`

Tableau 8-2. Options les plus couramment utilisées #[cfg]

#[cf g (. . .)] option	Activé lorsque
test	Les tests sont activés (compilation avec ou). cargo test rustc --test
debug _asse rtion s	Les assertions de débogage sont activées (généralement dans les builds non optimisées).
unix	Compilation pour Unix, y compris macOS.
wind ows	Compilation pour Windows.
targe t_poi nter_ width = "6 4"	Cibler une plateforme 64 bits. L'autre valeur possible est ". "32"
targe t_arc h = "x86_ 64"	Cibler x86-64 en particulier. Autres valeurs: , , , « powerpc » , , . "x86" "arm" "aarch64" "powerpc64" "mips"
targe t_os = "ma cos"	Compilation pour macOS. Autres valeurs : , , , , , . "windows" "ios" "android" "linux" "freebsd" "openbsd" "netbsd" "dragonfly"

```
#[cf
g          Activé lorsque
( . . . )]
option
```

feature La fonctionnalité définie par l'utilisateur nommée est activée (compilation avec ou). Les fonctionnalités sont déclarées dans la section [caractéristiques] de Cargo.toml. "robots" cargo build --feature robots rustc --cfg feature=' "robots" '

not(A n'est pas satisfait. Pour fournir deux implémentations différentes d'une fonction, marquez l'une avec et l'autre avec .#[cfg(X)] #[cfg(not(X))]

all(A et B sont tous deux satisfait (l'équivalent de). && AB ,)

any(A ou B est satisfait (l'équivalent de). || AB ,)

Parfois, nous devons microgérer l'expansion en ligne des fonctions, une optimisation que nous sommes généralement heureux de laisser au compilateur. Nous pouvons utiliser l'attribut pour cela: #[inline]

```
/// Adjust levels of ions etc. in two adjacent cells
/// due to osmosis between them.
#[inline]
fn do_osmosis(c1: &mut Cell, c2: &mut Cell) {
    ...
}
```

Il y a une situation où l'inlining ne se produira pas sans . Lorsqu'une fonction ou une méthode définie dans une caisse est appelée dans une autre caisse, Rust ne l'infiltre pas à moins qu'elle ne soit générique (elle a des paramètres de type) ou qu'elle soit explicitement marquée. #[inline] #[inline]

Sinon, le compilateur traite comme une suggestion. Rust prend également en charge les plus insistantes, pour demander qu'une fonction soit étendue en ligne à chaque site d'appel, et pour demander qu'une fonction ne soit jamais insérée. #[inline] #[inline(always)] #[inline(never)]

Certains attributs, comme `cfg`, peuvent être attachés à un module entier et s'appliquer à tout ce qu'il contient. D'autres, comme `allow`, doivent être attachés à des éléments individuels. Comme vous pouvez vous y attendre pour une fonctionnalité fourre-tout, chaque attribut est personnalisé et possède son propre ensemble d'arguments pris en charge. La référence Rust documente en détail [l'ensemble complet des attributs pris en charge](#).

Pour attacher un attribut à une caisse entière, ajoutez-le en haut du *fichier main.rs ou lib.rs*, avant tout élément, et écrivez à la place de `,` comme ceci :

```
// libgit2_sys/lib.rs
#![allow(non_camel_case_types)]  
  
pub struct git_revspec {  
    ...  
}  
  
pub struct git_error {  
    ...  
}
```

Le dit à Rust d'attacher un attribut à l'élément englobant plutôt que ce qui vient ensuite: dans ce cas, l'attribut se fixe à l'ensemble de la caisse, pas seulement `.#!` `#![allow]` `libgit2_sys struct git_revspec`

`#!` peut également être utilisé à l'intérieur des fonctions, des structures, etc., mais il n'est généralement utilisé qu'au début d'un fichier, pour attacher un attribut à l'ensemble du module ou de la caisse. Certains attributs utilisent toujours la syntaxe car ils ne peuvent être appliqués qu'à une caisse entière. `#!`

Par exemple, l'attribut `feature` est utilisé pour activer des *fonctionnalités instables* du langage et des bibliothèques Rust, des fonctionnalités expérimentales et qui peuvent donc présenter des bogues ou être modifiées ou supprimées à l'avenir. Par exemple, au moment où nous écrivons ceci, Rust dispose d'un support expérimental pour tracer l'expansion de macros comme `macro_rules!`, mais comme ce support est expérimental, vous ne pouvez l'utiliser qu'en (1) installant la version nocturne de Rust et (2) déclarant explicitement que votre caisse utilise le suivi des macros: `#![feature(trace_macros)]`

```
[feature] assert!  
  
#![feature(trace_macros)]
```

```

fn main() {
    // I wonder what actual Rust code this use of assert_eq!
    // gets replaced with!
    trace_macros!(true);
    assert_eq!(10*10*10 + 9*9*9, 12*12*12 + 1*1*1);
    trace_macros!(false);
}

```

Au fil du temps, l'équipe Rust *stabilise* parfois une fonctionnalité expérimentale afin qu'elle devienne une partie standard du langage. L'attribut devient alors superflu, et Rust génère un avertissement vous conseillant de le supprimer. `#![feature]`

Tests et documentation

Comme nous l'avons vu dans [« Writing and Running Unit Tests »](#), un cadre de test unitaire simple est intégré à Rust. Les tests sont des fonctions ordinaires marquées de l'attribut `#[test]`

```

#[test]
fn math_works() {
    let x: i32 = 1;
    assert!(x.is_positive());
    assert_eq!(x + 1, 2);
}

```

`cargo test` exécute tous les tests de votre projet :

```

$ cargo test
Compiling math_test v0.1.0 (file:///.../math_test)
Running target/release/math_test-e31ed91ae51ebf22

running 1 test
test math_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out

```

(Vous verrez également des résultats sur les « doc-tests », sur lesquels nous reviendrons dans une minute.)

Cela fonctionne de la même manière, que votre caisse soit un exécutable ou une bibliothèque. Vous pouvez exécuter des tests spécifiques en transmettant des arguments à Cargo : exécute tous les tests qui contiennent quelque part dans leur nom. `cargo test math math`

Les tests utilisent généralement les macros de la bibliothèque standard Rust. réussit si est vrai. Sinon, il panique, ce qui provoque l'échec du test. est identique, sauf que si l'assertion échoue, le message d'erreur affiche les deux valeurs. assert! assert_eq! assert!

```
(expr) expr assert_eq!(v1, v2) assert!(v1 == v2)
```

Vous pouvez utiliser ces macros dans du code ordinaire, pour vérifier les invariants, mais notez cela et sont inclus même dans les versions de version. Utilisez et à la place pour écrire des assertions qui sont vérifiées uniquement dans les versions de débogage. assert! assert_eq! debug_assert! debug_assert_eq!

Pour tester les cas d'erreur, ajoutez l'attribut à votre test : #

```
[should_panic]
```

```
// This test passes only if division by zero causes a panic,  
// as we claimed in the previous chapter.  
#[test]  
#[allow(unconditional_panic, unused_must_use)]  
#[should_panic(expected="divide by zero")]  
fn test_divide_by_zero_error() {  
    1 / 0; // should panic!  
}
```

Dans ce cas, nous devons également ajouter un attribut pour dire au compilateur de nous laisser faire des choses qu'il peut prouver statiquement vont paniquer, et effectuer des divisions et simplement jeter la réponse, parce que normalement, il essaie d'arrêter ce genre de bêtise. allow

Vous pouvez également retourner un de vos tests. Tant que la variante d'erreur est , ce qui est généralement le cas, vous pouvez simplement renvoyer un en utilisant pour jeter la variante: Result<(),

```
E> Debug Result ? Ok
```

```
use std::num::ParseIntError;  
  
// This test will pass if "1024" is a valid number, which it is.  
#[test]  
fn explicit_radix() -> Result<(), ParseIntError> {  
    i32::from_str_radix("1024", 10)?;  
    Ok(())  
}
```

Les fonctions marquées par sont compilées conditionnellement. Un code simple ou ignore le code de test. Mais lorsque vous exécutez, Cargo con-

struit votre programme deux fois: une fois de la manière ordinaire et une fois avec vos tests et le harnais de test activé. Cela signifie que vos tests unitaires peuvent vivre à côté du code qu'ils testent, en accédant aux détails de l'implémentation interne s'ils en ont besoin, et pourtant il n'y a pas de coût d'exécution. Cependant, cela peut entraîner certains avertissements. Par exemple: `#[test] cargo build cargo build --release cargo test`

```
fn roughly_equal(a: f64, b: f64) -> bool {
    (a - b).abs() < 1e-6
}

#[test]
fn trig_works() {
    use std::f64::consts::PI;
    assert!(roughly_equal(PI.sin(), 0.0));
}
```

Dans les builds qui omettent le code de test, semble inutilisé, et Rust se plaindra : `roughly_equal`

```
$ cargo build
   Compiling math_test v0.1.0 (file:///.../math_test)
warning: function is never used: `roughly_equal`
|
7 | / fn roughly_equal(a: f64, b: f64) -> bool {
8 | |     (a - b).abs() < 1e-6
9 | | }
| |
| = note: #[warn(dead_code)] on by default
```

Ainsi, la convention, lorsque vos tests deviennent suffisamment importants pour nécessiter un code de support, est de les placer dans un module et de déclarer que l'ensemble du module est testé uniquement à l'aide de l'attribut: `tests #[cfg]`

```
#[cfg(test)] // include this module only when testing
mod tests {
    fn roughly_equal(a: f64, b: f64) -> bool {
        (a - b).abs() < 1e-6
    }

    #[test]
    fn trig_works() {
        use std::f64::consts::PI;
```

```
        assert!(roughly_equal(PI.sin(), 0.0));
    }
}
```

Le harnais de test de Rust utilise plusieurs threads pour exécuter plusieurs tests à la fois, un avantage secondaire appréciable de votre code Rust étant thread-safe par défaut. Pour désactiver cette option, exécutez un seul test ou exécutez . (Le premier garantit que l'option passe à l'exécutable de test.) Cela signifie que, techniquement, le programme Mandelbrot que nous avons montré dans le [chapitre 2](#) n'était pas le deuxième programme multithread de ce chapitre, mais le troisième! L'exécution dans « [Writing and Running Unit Tests](#) » a été la première. cargo test testname cargo test -- --test-threads 1 -- cargo test -- test-threads cargo test

Normalement, le faisceau de test n'affiche que la sortie des tests qui ont échoué. Pour afficher la sortie des tests qui réussissent également, exécutez . cargo test -- --no-capture

Tests d'intégration

Votre simulateur de fougère continue de croître. Vous avez décidé de mettre toutes les fonctionnalités majeures dans une bibliothèque qui peut être utilisée par plusieurs exécutables. Ce serait bien d'avoir des tests qui se lient à la bibliothèque comme le ferait un utilisateur final, en utilisant *fern_sim.rlib* comme caisse externe. En outre, vous avez des tests qui commencent par charger une simulation enregistrée à partir d'un fichier binaire, et il est gênant d'avoir ces fichiers de test volumineux dans votre répertoire *src*. Les tests d'intégration aident à résoudre ces deux problèmes.

Les tests d'intégration sont *des fichiers .rs* qui se trouvent dans un répertoire *de tests* à côté du répertoire *src* de votre projet. Lorsque vous exécutez , Cargo compile chaque test d'intégration sous la forme d'une caisse distincte et autonome, liée à votre bibliothèque et au harnais de test Rust. Voici un exemple : cargo test

```
// tests/unfurl.rs - Fiddleheads unfurl in sunlight

use fern_sim::Terrarium;
use std::time::Duration;

#[test]
fn test_fiddlehead_unfurling() {
    let mut world = Terrarium::load("tests/unfurl_files/fiddlehead.tm");
```

```
    assert!(world.fern(0).is_furled());
    let one_hour = Duration::from_secs(60 * 60);
    world.apply_sunlight(one_hour);
    assert!(world.fern(0).is_fully_unfurled());
}
}
```

Les tests d'intégration sont précieux en partie parce qu'ils voient votre caisse de l'extérieur, tout comme le ferait un utilisateur. Ils testent l'API publique de la caisse.

`cargo test` exécute à la fois des tests unitaires et des tests d'intégration. Pour exécuter uniquement les tests d'intégration dans un fichier particulier (par exemple, `tests/unfurl.rs`), utilisez la commande `.cargo test --test unfurl`

Documentation

La commande crée une documentation HTML pour votre bibliothèque

```
:cargo doc
```

```
$ cargo doc --no-deps --open
Documenting fern_sim v0.1.0 (file:///.../fern_sim)
```

L'option indique à Cargo de générer de la documentation uniquement pour elle-même, et non pour toutes les caisses dont elle dépend. `--no-deps fern_sim`

L'option indique à Cargo d'ouvrir la documentation dans votre navigateur par la suite. `--open`

Vous pouvez voir le résultat à [la figure 8-2](#). Cargo enregistre les nouveaux fichiers de documentation dans `target/doc`. La page de démarrage est `target/doc/fern_sim/index.html`.

Click or press 'S' to search, '?' for more options...

crate `fern_sim`

[[-](#)] [[src](#)]

[[-](#)] Simulate the growth of ferns, from the level of individual cells on up.

Reexports

```
pub use plant_structures::Fern;  
pub use simulation::Terrarium;
```

Modules

<code>cells</code>	The simulation of biological cells, which is as low-level as we go.
<code>plant_structures</code>	Higher-level biological structures.
<code>simulation</code>	Overall simulation control.
<code>spores</code>	Fern reproduction.

Graphique 8-2. Exemple de documentation générée par `rustdoc`

La documentation est générée à partir des fonctionnalités de votre bibliothèque, ainsi que de tous les *commentaires de document* que vous y avez joints. Nous avons déjà vu quelques commentaires de doc dans ce chapitre. Ils ressemblent à des commentaires: pub

```
/// Simulate the production of a spore by meiosis.  
pub fn produce_spore(factory: &mut Sporangium) -> Spore {  
    ...  
}
```

Mais lorsque Rust voit des commentaires qui commencent par trois barres obliques, il les traite plutôt comme un attribut. Rust traite l'exemple précédent exactement de la même manière que ceci : #[doc]

```
#[doc = "Simulate the production of a spore by meiosis."]  
pub fn produce_spore(factory: &mut Sporangium) -> Spore {  
    ...  
}
```

Lorsque vous compilez une bibliothèque ou un fichier binaire, ces attributs ne changent rien, mais lorsque vous générez de la documentation, des commentaires de document sur les fonctionnalités publiques sont inclus dans la sortie.

De même, les commentaires commençant par sont traités comme des attributs et sont attachés à la fonction englobante, généralement un module ou une caisse. Par exemple, votre fichier `fern_sim/src/lib.rs` peut commencer comme suit : `///! #![doc]`

```
//! Simulate the growth of ferns, from the level of
//! individual cells on up.
```

Le contenu d'un commentaire de document est traité comme Markdown, une notation abrégée pour une mise en forme HTML simple. Les astérisques sont utilisés pour `*`, une ligne vide est traitée comme un saut de paragraphe, et ainsi de suite. Vous pouvez également inclure des balises HTML, qui sont copiées textuellement dans la documentation formatée. `*italics* **bold type**`

Une particularité des commentaires de document dans Rust est que les liens Markdown peuvent utiliser des chemins d'élément Rust, comme `,`, au lieu d'URL relatives, pour indiquer à quoi ils se réfèrent. Cargo recherchera à quoi le chemin fait référence et sous-titra un lien au bon endroit dans la bonne page de documentation. Par exemple, la documentation générée à partir de ce code renvoie aux pages de documentation pour `,`, et `:leaves::Leaf VascularPath Leaf Root`

```
/// Create and return a [ `VascularPath` ] which represents the path of
/// nutrients from the given [ `Root` ][r] to the given [ `Leaf` ](leaves::Leaf)
///
/// [r]: roots::Root
pub fn trace_path(leaf: &leaves::Leaf, root: &roots::Root) -> VascularPath
{
    ...
}
```

Vous pouvez également ajouter des alias de recherche pour faciliter la recherche d'éléments à l'aide de la fonction de recherche intégrée. La recherche de « chemin » ou de « route » dans la documentation de cette caisse conduira à `:VascularPath`

```
#[doc(alias = "route")]
pub struct VascularPath {
    ...
}
```

Pour des blocs de documentation plus longs ou pour rationaliser votre flux de travail, vous pouvez inclure des fichiers externes dans votre documentation. Par exemple, si *le fichier README.md* de votre référentiel contient le même texte que vous souhaitez utiliser comme documentation de niveau supérieur de votre caisse, vous pouvez le mettre en haut de ou :

```
:lib.rs main.rs
```

```
#![doc = include_str!("../README.md")]
```

Vous pouvez l'utiliser pour définir des bits de code au milieu du texte en cours d'exécution. Dans la sortie, ces extraits seront formatés dans une police à largeur fixe. Des exemples de code plus grands peuvent être ajoutés en mettant en retrait quatre espaces : `backticks`

```
/// A block of code in a doc comment:  
///  
///     if samples::everything().works() {  
///         println!("ok");  
///     }
```

Vous pouvez également utiliser des blocs de code clôturés Markdown. Cela a exactement le même effet:

```
/// Another snippet, the same code, but written differently:  
///  
/// ``  
/// if samples::everything().works() {  
///     println!("ok");  
/// }  
/// ``
```

Quel que soit le format que vous utilisez, une chose intéressante se produit lorsque vous incluez un bloc de code dans un commentaire de document. La rouille le transforme automatiquement en test.

Doc-Tests

Lorsque vous exécutez des tests dans une caisse de bibliothèque Rust, Rust vérifie que tout le code qui apparaît dans votre documentation s'exécute et fonctionne réellement. Pour ce faire, il prend chaque bloc de code qui apparaît dans un commentaire de document, le compile en tant que caisse exécutable distincte, le lie à votre bibliothèque et l'exécute.

Voici un exemple autonome de doc-test. Créez un nouveau projet en exécutant (l'indicateur indique à Cargo que nous créons une caisse de bibliothèque, pas une caisse exécutable) et placez le code suivant dans `ranges/src/lib.rs`:

```
cargo new --lib ranges --lib
```

```
use std::ops::Range;

/// Return true if two ranges overlap.
///
///     assert_eq!(ranges::overlap(0..7, 3..10), true);
///     assert_eq!(ranges::overlap(1..5, 101..105), false);
///
/// If either range is empty, they don't count as overlapping.
///
///     assert_eq!(ranges::overlap(0..0, 0..10), false);
///

pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool {
    r1.start < r1.end && r2.start < r2.end &&
        r1.start < r2.end && r2.start < r1.end
}
```

Les deux petits blocs de code dans le commentaire de la documentation apparaissent dans la documentation générée par `cargo doc`, comme illustré à [la figure 8-3](#).

Function `ranges::overlap`

[[-](#)] [[src](#)]

```
pub fn overlap(r1: Range<usize>, r2: Range<usize>) -> bool
```

[[-](#)] Return true if two ranges overlap.

```
assert_eq!(ranges::overlap(0..7, 3..10), true);
assert_eq!(ranges::overlap(1..5, 101..105), false);
```

If either range is empty, they don't count as overlapping.

```
assert_eq!(ranges::overlap(0..0, 0..10), false);
```

Figure 8-3. Documentation montrant quelques doc-tests

Ils deviennent également deux tests distincts:

```
$ cargo test
  Compiling ranges v0.1.0 (file:///.../ranges)
...
Doc-tests ranges

running 2 tests
```

```
test overlap_0 ... ok
test overlap_1 ... ok
```

```
test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured; 0 filtered out
```

Si vous passez le drapeau à Cargo, vous verrez qu'il est utilisé pour exécuter ces deux tests. stocke chaque exemple de code dans un fichier distinct, en ajoutant quelques lignes de code standard, pour produire deux programmes. Voici la première :--verbose rustdoc --test rustdoc

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..7, 3..10), true);
    assert_eq!(ranges::overlap(1..5, 101..105), false);
}
```

Et voici la seconde :

```
use ranges;
fn main() {
    assert_eq!(ranges::overlap(0..0, 0..10), false);
}
```

Les tests réussissent si ces programmes sont compilés et exécutés avec succès.

Ces deux exemples de code contiennent des assertions, mais c'est simplement parce que dans ce cas, les assertions font une documentation décente. L'idée derrière les doc-tests n'est pas de mettre tous vos tests en commentaires. Au lieu de cela, vous écrivez la meilleure documentation possible, et Rust s'assure que les exemples de code de votre documentation sont réellement compilés et exécutés.

Très souvent, un exemple de travail minimal inclut certains détails, tels que les importations ou le code d'installation, qui sont nécessaires pour compiler le code, mais qui ne sont tout simplement pas assez importants pour être affichés dans la documentation. Pour masquer une ligne d'un exemple de code, placez un suivi d'un espace au début de cette ligne :#

```
/// Let the sun shine in and run the simulation for a given
/// amount of time.
///
///     # use fern_sim::Terrarium;
///     # use std::time::Duration;
///     # let mut tm = Terrarium::new();
///     tm.apply_sunlight(Duration::from_secs(60));
```

```

/// 
pub fn apply_sunlight(&mut self, time: Duration) {
    ...
}

```

Parfois, il est utile d'afficher un exemple complet de programme dans la documentation, y compris une fonction. Évidemment, si ces morceaux de code apparaissent dans votre exemple de code, vous ne voulez pas non plus les ajouter automatiquement. Le résultat ne se compilerait pas. Il traite donc tout bloc de code contenant la chaîne exacte comme un programme complet et n'y ajoute rien. `main rustdoc rustdoc fn main`

Les tests peuvent être désactivés pour des blocs de code spécifiques. Pour demander à Rust de compiler votre exemple, mais de ne pas l'exécuter réellement, utilisez un bloc de code clôturé avec l'annotation : `no_run`

```

/// Upload all local terrariums to the online gallery.
///
/// ````no_run
/// let mut session = fern_sim::connect();
/// session.upload_all();
/// ````

pub fn upload_all(&mut self) {
    ...
}

```

Si le code n'est même pas censé être compilé, utilisez `ignore` à la place de `no_run`. Les blocs marqués avec `ignore` n'apparaissent pas dans la sortie de `rustdoc`, mais les tests apparaissent comme ayant réussi s'ils compilent. Si le bloc de code n'est pas du tout du code Rust, utilisez le nom de la langue, comme `c` ou `sh` pour le texte brut. Rust ne connaît pas les noms de centaines de langages de programmation ; il traite plutôt toute annotation qu'il ne reconnaît pas comme indiquant que le bloc de code n'est pas Rust. Cela désactive la mise en surbrillance du code ainsi que les tests de documents.

```

ignore no_run ignore cargo
run no_run c++ sh text rustdoc

```

Spécification des dépendances

Nous avons vu une façon de dire à Cargo où obtenir le code source pour les caisses dont dépend votre projet : par numéro de version.

```
image = "0.6.1"
```

Il existe plusieurs façons de spécifier les dépendances, et certaines choses plutôt nuancées que vous voudrez peut-être dire sur les versions à utiliser, il vaut donc la peine de passer quelques pages à ce sujet.

Tout d'abord, vous pouvez utiliser des dépendances qui ne sont pas publiées sur crates.io du tout. Une façon de le faire consiste à spécifier une URL de référentiel Git et une révision :

```
image = { git = "https://github.com/Piston/image.git", rev = "528f19c" }
```

Cette caisse particulière est open source, hébergée sur GitHub, mais vous pouvez tout aussi bien pointer vers un référentiel Git privé hébergé sur votre réseau d'entreprise. Comme indiqué ici, vous pouvez spécifier le , ou à utiliser. (Ce sont toutes des façons de dire à Git quelle révision du code source extraire.) `rev tag branch`

Une autre alternative consiste à spécifier un répertoire qui contient le code source de la caisse :

```
image = { path = "vendor/image" }
```

Ceci est pratique lorsque votre équipe dispose d'un référentiel de contrôle de version unique qui contient le code source de plusieurs caisses, ou peut-être l'ensemble du graphique de dépendance. Chaque caisse peut spécifier ses dépendances à l'aide de chemins relatifs.

Avoir ce niveau de contrôle sur vos dépendances est puissant. Si jamais vous décidez que l'une des caisses open source que vous utilisez n'est pas exactement à votre goût, vous pouvez trivialement le fork: appuyez simplement sur le bouton Fork sur GitHub et changez une ligne dans votre fichier `Cargo.toml`. Votre prochain utilisera de manière transparente votre fourchette de la caisse au lieu de la version officielle. `cargo build`

Versions

Lorsque vous écrivez quelque chose comme dans votre fichier `Cargo.toml`, Cargo interprète cela de manière assez lâche. Il utilise la version la plus récente de qui est considérée comme compatible avec la version 0.13.0. `image = "0.13.0"` `image`

Les règles de compatibilité sont adaptées du [contrôle de version séquentielle](#).

- Un numéro de version qui commence par 0.0 est si brut que Cargo ne suppose jamais qu'il est compatible avec une autre version.
- Numéro de version commençant par 0. x, où x est non nul, est considéré comme compatible avec d'autres versions ponctuelles dans le 0. x série. Nous avons spécifié la version 0.6.1, mais Cargo utiliserait 0.6.3 si disponible. (Ce n'est pas ce que dit la norme Semantic Versioning à propos de 0. x numéros de version, mais la règle s'est avérée trop utile pour être omise.) `image`
- Une fois qu'un projet atteint la version 1.0, seules les nouvelles versions majeures rompent la compatibilité. Donc, si vous demandez la version 2.0.1, Cargo peut utiliser 2.17.99 à la place, mais pas 3.0.

Les numéros de version sont flexibles par défaut car sinon le problème de la version à utiliser deviendrait rapidement trop contraignant. Supposons qu'une bibliothèque, , soit utilisée tandis qu'une autre, , utilise . Si les numéros de version nécessitaient des correspondances exactes, aucun projet ne serait en mesure d'utiliser ces deux bibliothèques ensemble. Permettre à Cargo d'utiliser n'importe quelle version compatible est une valeur par défaut beaucoup plus pratique.

```
libA num = "0.1.31"
libB num = "0.1.29"
```

Pourtant, différents projets ont des besoins différents en matière de dépendances et de gestion des versions. Vous pouvez spécifier une version exacte ou une plage de versions à l'aide d'opérateurs, comme illustré dans [le tableau 8-3](#).

Tableau 8-3. Spécification des versions dans un fichier Cargo.toml

Ligne Cargo.toml	Signification
<code>image = "=0.1.0"</code>	Utilisez uniquement la version exacte 0.10.0
<code>image = ">=1.0.5"</code>	Utilisez 1.0.5 ou <i>toute version</i> supérieure (même 2.9, si elle est disponible)
<code>image = ">1.0.5 <1.1.9"</code>	Utilisez une version supérieure à 1.0.5, mais inférieure à 1.1.9
<code>image = "<=2.7.10"</code>	Utilisez n'importe quelle version jusqu'à 2.7.10

Une autre spécification de version que vous verrez occasionnellement est le caractère générique . Cela indique à Cargo que n'importe quelle ver-

sion fera l'affaire. À moins qu'un autre fichier *Cargo.toml* ne contienne une contrainte plus spécifique, Cargo utilisera la dernière version disponible. [La documentation Cargo de doc.crates.io](#) couvre encore plus en détail les spécifications de version. *

Notez que les règles de compatibilité signifient que les numéros de version ne peuvent pas être choisis uniquement pour des raisons de marketing. Ils signifient en fait quelque chose. Il s'agit d'un contrat entre les responsables de la maintenance d'une caisse et ses utilisateurs. Si vous maintenez une caisse qui se trouve à la version 1.7 et que vous décidez de supprimer une fonction ou d'apporter toute autre modification qui n'est pas entièrement rétrocompatible, vous devez passer votre numéro de version à 2.0. Si vous deviez l'appeler 1.8, vous prétendriez que la nouvelle version est compatible avec 1.7, et vos utilisateurs pourraient se retrouver avec des builds cassés.

Cargo.lock

Les numéros de version dans *Cargo.toml* sont délibérément flexibles, mais nous ne voulons pas que Cargo nous mette à niveau vers les dernières versions de bibliothèque chaque fois que nous construisons. Imaginez être au milieu d'une session de débogage intense lorsque vous passez soudainement à une nouvelle version d'une bibliothèque. Cela pourrait être incroyablement perturbateur. Tout ce qui change au milieu du débogage est mauvais. En fait, quand il s'agit de bibliothèques, il n'y a jamais de bon moment pour un changement inattendu. `cargo build`

Cargo dispose donc d'un mécanisme intégré pour éviter cela. La première fois que vous créez un projet, Cargo génère un fichier *Cargo.lock* qui enregistre la version exacte de chaque caisse utilisée. Les versions ultérieures consulteront ce fichier et continueront à utiliser les mêmes versions. Cargo passe à des versions plus récentes uniquement lorsque vous le lui demandez, soit en augmentant manuellement le numéro de version dans votre fichier *Cargo.toml*, soit en exécutant : `cargo update`

```
$ cargo update
    Updating registry `https://github.com/rust-lang/crates.io-index`
    Updating libc v0.2.7 -> v0.2.11
    Updating png v0.4.2 -> v0.4.3
```

`cargo update` ne met à niveau que vers les dernières versions compatibles avec ce que vous avez spécifié dans *Cargo.toml*. Si vous avez spécifié , et que vous souhaitez effectuer une mise à niveau vers la version 0.10.0, vous devrez modifier cela dans *Cargo.toml*. La prochaine fois que vous

construirez, Cargo mettra à jour vers la nouvelle version de la bibliothèque et stockera le nouveau numéro de version dans *Cargo.lock*. `image = "0.6.1"` `image`

L'exemple précédent montre Cargo mettant à jour deux caisses hébergées sur crates.io. Quelque chose de très similaire se produit pour les dépendances stockées dans Git. Supposons que notre fichier *Cargo.toml* contienne ceci :

```
image = { git = "https://github.com/Piston/image.git", branch = "master"
```

`cargo build` n'extraira pas les nouvelles modifications du référentiel Git s'il voit que nous avons un fichier *Cargo.lock*. Au lieu de cela, il lit *Cargo.lock* et utilise la même révision que la dernière fois. Mais tirera de sorte que notre prochaine version utilise la dernière révision. `cargo update master`

Cargo.lock est généré automatiquement pour vous, et vous ne le modifierez normalement pas à la main. Néanmoins, si votre projet est un exécutable, vous devez valider *Cargo.lock* au contrôle de version. De cette façon, tous ceux qui construisent votre projet obtiendront systématiquement les mêmes versions. L'historique de votre fichier *Cargo.lock* enregistrera vos mises à jour de dépendance.

Si votre projet est une bibliothèque Rust ordinaire, ne vous embêtez pas à valider *Cargo.lock*. Les utilisateurs en aval de votre bibliothèque disposeront de fichiers *Cargo.lock* contenant des informations de version pour l'ensemble de leur graphique de dépendance ; ils ignoreront le fichier *Cargo.lock* de votre bibliothèque. Dans les rares cas où votre projet est une bibliothèque partagée (c'est-à-dire que la sortie est un fichier *.dll*, *.dylib* ou *.so*), il n'existe pas d'utilisateur en aval et vous devez donc valider *Cargo.lock*. `cargo`

Les spécificateurs de version flexibles de *Cargo.toml* facilitent l'utilisation des bibliothèques Rust dans votre projet et optimisent la compatibilité entre les bibliothèques. La comptabilité de *Cargo.lock* prend en charge des constructions cohérentes et reproductibles sur toutes les machines. Ensemble, ils contribuent grandement à vous aider à éviter l'enfer de la dépendance.

Publication de caisses sur crates.io

Vous avez décidé de publier votre bibliothèque de simulation de fougères en tant que logiciel open source. Félicitations! Cette partie est facile.

Tout d'abord, assurez-vous que Cargo peut emballer la caisse pour vous.

```
$ cargo package
warning: manifest has no description, license, license-file, documentation
homepage or repository. See http://doc.crates.io/manifest.html#package-me
for more info.

    Packaging fern_sim v0.1.0 (file:///.../fern_sim)
    Verifying fern_sim v0.1.0 (file:///.../fern_sim)
    Compiling fern_sim v0.1.0 (file:///.../fern_sim/target/package/fern_si
```

La commande crée un fichier (dans ce cas, *target/package/fern_sim-0.1.0.crate*) contenant tous les fichiers sources de votre bibliothèque, y compris *Cargo.toml*. Il s'agit du fichier que vous allez télécharger sur crates.io pour le partager avec le monde. (Vous pouvez l'utiliser pour voir quels fichiers sont inclus.) Cargo vérifie ensuite son travail en construisant votre bibliothèque à partir du fichier *.crate*, tout comme vos éventuels utilisateurs. `cargo package cargo package --list`

Cargo avertit qu'il manque à la section *de Cargo.toml* certaines informations qui seront importantes pour les utilisateurs en aval, telles que la licence sous laquelle vous distribuez le code. L'URL dans l'avertissement est une excellente ressource, nous n'expliquerons donc pas tous les champs en détail ici. En bref, vous pouvez corriger l'avertissement en ajoutant quelques lignes à *Cargo.toml*: [package]

```
[package]
name = "fern_sim"
version = "0.1.0"
edition = "2021"
authors = [ "You <you@example.com>" ]
license = "MIT"
homepage = "https://fernsm.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsm.example.com/docs"
description = """
Fern simulation, from the cellular level up.
"""

```

NOTE

Une fois que vous publiez cette caisse sur crates.io, toute personne qui télécharge votre caisse peut voir le fichier *Cargo.toml*. Donc, si le champ contient une adresse e-mail que vous préférez garder privée, il est maintenant temps de la modifier. `authors`

Un autre problème qui se pose parfois à ce stade est que votre fichier *Cargo.toml* peut spécifier l'emplacement d'autres caisses par , comme indiqué dans « [Spécification des dépendances](#) »: path

```
image = { path = "vendor/image" }
```

Pour vous et votre équipe, cela pourrait bien fonctionner. Mais naturellement, lorsque d'autres personnes téléchargent la bibliothèque, elles n'auront pas les mêmes fichiers et répertoires sur leur ordinateur que vous. Cargo ignore donc la clé dans les *bibliothèques téléchargées* automatiquement, ce qui peut provoquer des erreurs de construction. La solution, cependant, est simple: si votre bibliothèque doit être publiée sur crates.io, ses dépendances doivent également être sur crates.io. Spécifiez un numéro de version au lieu d'un : `fern_sim path path`

```
image = "0.13.0"
```

Si vous préférez, vous pouvez spécifier à la fois a , qui a priorité pour vos propres builds locales et a pour tous les autres utilisateurs
: path version

```
image = { path = "vendor/image", version = "0.13.0" }
```

Bien sûr, dans ce cas, il est de votre responsabilité de vous assurer que les deux restent synchronisés.

Enfin, avant de publier une caisse, vous devez vous connecter à crates.io et obtenir une clé API. Cette étape est simple: une fois que vous avez un compte sur crates.io, votre page « Paramètres du compte » affichera une commande, comme celle-ci: `cargo login`

```
$ cargo login 5j0dv54Bj1XBpUUbFIj7G9DvNl1vsWW1
```

Cargo enregistre la clé dans un fichier de configuration et la clé API doit rester secrète, comme un mot de passe. Exécutez donc cette commande uniquement sur un ordinateur que vous contrôlez.

Cela fait, la dernière étape consiste à exécuter : `cargo publish`

```
$ cargo publish
  Updating registry `https://github.com/rust-lang/crates.io-index`
  Uploading fern_sim v0.1.0 (file:///.../fern_sim)
```

Avec cela, votre bibliothèque rejoint des milliers d'autres sur crates.io.

Espaces de travail

Au fur et à mesure que votre projet continue de grandir, vous finissez par écrire de nombreuses caisses. Ils vivent côté à côté dans un référentiel source unique :

```
fernsoft/
└── .git/...
└── fern_sim/
    ├── Cargo.toml
    ├── Cargo.lock
    └── src/...
        └── target/...
└── fern_img/
    ├── Cargo.toml
    ├── Cargo.lock
    └── src/...
        └── target/...
└── fern_video/
    ├── Cargo.toml
    ├── Cargo.lock
    └── src/...
        └── target/...
```

La façon dont Cargo fonctionne, chaque caisse a son propre répertoire de build, qui contient une build séparée de toutes les dépendances de cette caisse. Ces répertoires de build sont complètement indépendants. Même si deux caisses ont une dépendance commune, elles ne peuvent partager aucun code compilé. C'est du gaspillage.

Vous pouvez économiser du temps de compilation et de l'espace disque à l'aide d'un *espace de travail* Cargo, d'une collection de caisses qui partagent un répertoire de build commun et un fichier *Cargo.lock*.

Tout ce que vous avez à faire est de créer un fichier *Cargo.toml* dans le répertoire racine de votre référentiel et d'y mettre ces lignes:

```
[workspace]
members = ["fern_sim", "fern_img", "fern_video"]
```

Voici etc. les noms des sous-répertoires contenant vos caisses. Supprimez tous les fichiers *Cargo.lock* restants et les répertoires *cibles* qui existent dans ces sous-répertoires. *fern_sim*

Une fois que vous avez fait cela, dans n'importe quelle caisse créera et utilisera automatiquement un répertoire de build partagé sous le réper-

toire racine (dans ce cas, *fernsoft / target*). La commande génère toutes les caisses de l'espace de travail actuel. et acceptez également l'option. cargo build cargo build --workspace cargo test cargo doc --workspace

Plus de belles choses

Au cas où vous ne seriez pas encore ravi, la communauté Rust a encore quelques chances et fins pour vous:

- Lorsque vous publiez une caisse open source sur [crates.io](#), votre documentation est automatiquement rendue et hébergée sur *docs.rs* grâce à Onur Aslan.
- Si votre projet est sur GitHub, Travis CI peut générer et tester votre code à chaque push. C'est étonnamment facile à configurer; voir [travis-ci.org](#) pour plus de détails. Si vous êtes déjà familier avec Travis, ce fichier *.travis.yml* vous aidera à démarrer:

```
language: rust
rust:
  - stable
```

- Vous pouvez générer un fichier *README.md* à partir du doc-commen-taire de niveau supérieur de votre caisse. Cette fonctionnalité est pro-posée en tant que plug-in Cargo tiers par Livio Ribeiro. Exécutez pour installer le plug-in, puis pour apprendre à l'utiliser. cargo install cargo-readme cargo readme --help

Nous pourrions continuer.

Rust est nouveau, mais il est conçu pour soutenir de grands projets am-bitieux. Il dispose d'excellents outils et d'une communauté active. Les programmeurs système peuvent avoir de belles choses.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 9. Structures

Il y a longtemps, lorsque les bergers voulaient voir si deux troupeaux de moutons étaient isomorphes, ils cherchaient un isomorphisme explicite.

—John C. Baez et James Dolan, [« Catégorisation »](#)

Les structures Rust, parfois appelées *structures*, ressemblent à des types en C et C++, à des classes en Python et à des objets en JavaScript. Une structure assemble plusieurs valeurs de types assortis en une seule valeur afin que vous puissiez les traiter en tant qu'unité. Étant donné une structure, vous pouvez lire et modifier ses composants individuels. Et une struct peut avoir des méthodes qui lui sont associées qui fonctionnent sur ses composants. `struct`

Rust a trois types de struct, *named-field*, *tuple-like* et *unit-like*, qui diffèrent dans la façon dont vous vous référez à leurs composants: une struct de champ nommé donne un nom à chaque composant, tandis qu'une struct de type tuple les identifie par l'ordre dans lequel ils apparaissent. Les structures de type unitaire n'ont aucun composant; ceux-ci ne sont pas courants, mais plus utiles que vous ne le pensez.

Dans ce chapitre, nous expliquerons chaque type en détail et montrerons à quoi ils ressemblent en mémoire. Nous verrons comment y ajouter des méthodes, comment définir des types de struct génériques qui fonctionnent avec de nombreux types de composants différents et comment demander à Rust de générer des implémentations de traits pratiques courants pour vos structs.

Structures de champ nommé

La définition d'un type de structure de champ nommé ressemble à ceci :

```
// A rectangle of eight-bit grayscale pixels.  
struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

Cela déclare un type avec deux champs nommés et , des types donnés. La convention dans Rust est que tous les types, y compris les structs, ont des noms qui mettent en majuscule la première lettre de chaque mot, comme , une convention appelée *CamelCase* (ou *PascalCase*). Les champs et les méthodes sont en minuscules, avec des mots séparés par des traits de soulignement. C'est ce qu'on appelle

```
snake_case. GrayscaleMap pixels size GrayscaleMap
```

Vous pouvez construire une valeur de ce type avec une *expression struct*, comme ceci :

```
let width = 1024;
let height = 576;
let image = GrayscaleMap {
    pixels: vec![0; width * height],
    size: (width, height)
};
```

Une expression struct commence par le nom du type () et répertorie le nom et la valeur de chaque champ, tous entourés d'accolades. Il existe également un raccourci pour remplir des champs à partir de variables locales ou d'arguments portant le même nom : `GrayscaleMap`

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) -> GrayscaleMap {
    assert_eq!(pixels.len(), size.0 * size.1);
    GrayscaleMap { pixels, size }
}
```

L'*expression struct* est l'abréviation de . Vous pouvez utiliser la syntaxe pour certains champs et la sténographie pour d'autres dans la même expression struct. `GrayscaleMap { pixels, size } GrayscaleMap { pixels: pixels, size: size }` `key: value`

Pour accéder aux champs d'une structure, utilisez l'opérateur familier : .

```
assert_eq!(image.size, (1024, 576));
assert_eq!(image.pixels.len(), 1024 * 576);
```

Comme tous les autres éléments, les structs sont privées par défaut, visibles uniquement dans le module où elles sont déclarées et ses sous-modules. Vous pouvez rendre une structure visible en dehors de son module en préfixant sa définition par . Il en va de même pour chacun de ses champs, qui sont également privés par défaut : pub

```
// A rectangle of eight-bit grayscale pixels.  
pub struct GrayscaleMap {  
    pub pixels: Vec<u8>,  
    pub size: (usize, usize)  
}
```

Même si une struct est déclarée, ses champs peuvent être privés : pub

```
// A rectangle of eight-bit grayscale pixels.  
pub struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

D'autres modules peuvent utiliser cette struct et toutes les fonctions publiques associées qu'il pourrait avoir, mais ne peuvent pas accéder aux champs privés par nom ou utiliser des expressions struct pour créer de nouvelles valeurs. Autrement dit, la création d'une valeur struct nécessite que tous les champs de la struct soient visibles. C'est pourquoi vous ne pouvez pas écrire une expression struct pour créer un nouveau ou . Ces types standard sont des structs, mais tous leurs champs sont privés. Pour en créer un, vous devez utiliser des fonctions publiques associées au type telles que .GrayscaleMap String Vec::new()

Lors de la création d'une valeur struct de champ nommé, vous pouvez utiliser une autre struct du même type pour fournir des valeurs pour les champs que vous omettez. Dans une expression struct, si les champs nommés sont suivis de , alors tous les champs non mentionnés prennent leurs valeurs de , qui doit être une autre valeur du même type struct. Supposons que nous ayons une structure représentant un monstre dans un jeu : ... EXPR EXPR

```
// In this game, brooms are monsters. You'll see.  
struct Broom {  
    name: String,  
    height: u32,  
    health: u32,  
    position: (f32, f32, f32),  
    intent: BroomIntent  
}  
  
// Two possible alternatives for what a `Broom` could be working on.
```

```
#[derive(Copy, Clone)]  
enum BroomIntent { FetchWater, DumpWater }
```

Le meilleur conte de fées pour les programmeurs est *L'Apprenti sorcier*: un magicien novice enchante un balai pour faire son travail à sa place, mais ne sait pas comment l'arrêter lorsque le travail est terminé. Couper le balai en deux avec une hache ne produit que deux balais, chacun de la moitié de la taille, mais en continuant la tâche avec le même dévouement aveugle que l'original:

```
// Receive the input Broom by value, taking ownership.  
fn chop(b: Broom) -> (Broom, Broom) {  
    // Initialize `broom1` mostly from `b`, changing only `height`. Since  
    // `String` is not `Copy`, `broom1` takes ownership of `b`'s name.  
    let mut broom1 = Broom { height: b.height / 2, .. b };  
  
    // Initialize `broom2` mostly from `broom1`. Since `String` is not  
    // `Copy`, we must clone `name` explicitly.  
    let mut broom2 = Broom { name: broom1.name.clone(), .. broom1 };  
  
    // Give each fragment a distinct name.  
    broom1.name.push_str(" I");  
    broom2.name.push_str(" II");  
  
    (broom1, broom2)  
}
```

Avec cette définition en place, nous pouvons créer un balai, le couper en deux et voir ce que nous obtenons:

```
let hokey = Broom {  
    name: "Hokey".to_string(),  
    height: 60,  
    health: 100,  
    position: (100.0, 200.0, 0.0),  
    intent: BroomIntent::FetchWater  
};  
  
let (hokey1, hokey2) = chop(hokey);  
assert_eq!(hokey1.name, "Hokey I");  
assert_eq!(hokey1.height, 30);  
assert_eq!(hokey1.health, 100);  
  
assert_eq!(hokey2.name, "Hokey II");
```

```
assert_eq!(hokey2.height, 30);
assert_eq!(hokey2.health, 100);
```

Le nouveau et les balais ont reçu des noms ajustés, la moitié de la hauteur et toute la santé de l'original. hokey1 hokey2

Structures de type Tuple

Le deuxième type de struct est appelé une *struct de type tuple*, car il ressemble à un tuple:

```
struct Bounds(usize, usize);
```

Vous construisez une valeur de ce type autant que vous construiriez un tuple, sauf que vous devez inclure le nom struct :

```
let image_bounds = Bounds(1024, 768);
```

Les valeurs détenues par une structure de type tuple sont appelées *éléments*, tout comme les valeurs d'un tuple. Vous y accédez comme vous le feriez pour un tuple:

```
assert_eq!(image_bounds.0 * image_bounds.1, 786432);
```

Les éléments individuels d'une structure de type tuple peuvent être publics ou non :

```
pub struct Bounds(pub usize, pub usize);
```

L'expression ressemble à un appel de fonction, et en fait c'est le cas : la définition du type définit aussi implicitement une fonction
: Bounds(1024, 768)

```
fn Bounds(elem0: usize, elem1: usize) -> Bounds { ... }
```

Au niveau le plus fondamental, les structures de champ nommé et de type tuple sont très similaires. Le choix de ce qu'il faut utiliser se résume à des questions de lisibilité, d'ambiguïté et de brièveté. Si vous utilisez l'opérateur pour obtenir beaucoup les composants d'une valeur, l'identification des champs par nom fournit au lecteur plus d'informations et est probablement plus robuste contre les fautes de frappe. Si vous utilisez

habituellement la correspondance de motifs pour trouver les éléments, les structures de type tuple peuvent bien fonctionner. .

Les structs de type tuple sont bonnes pour *les nouveaux types*, les structs avec un seul composant que vous définissez pour obtenir une vérification de type plus stricte. Par exemple, si vous travaillez avec du texte ASCII uniquement, vous pouvez définir un nouveau type comme celui-ci :

```
struct Ascii(Vec<u8>);
```

L'utilisation de ce type pour vos chaînes ASCII est bien meilleure que de simplement passer autour des tampons et d'expliquer ce qu'ils sont dans les commentaires. Le nouveau type aide Rust à détecter les erreurs lorsqu'un autre tampon d'octets est transmis à une fonction attendant du texte ASCII. Nous donnerons un exemple d'utilisation de nouveaux types pour des conversions de types efficaces au [chapitre 22](#). Vec<u8>

Structures de type unitaire

Le troisième type de struct est un peu obscur : il déclare un type de struct sans aucun élément :

```
struct Onesuch;
```

Une valeur d'un tel type n'occupe aucune mémoire, tout comme le type d'unité . Rust ne prend pas la peine de stocker des valeurs de structure de type unité en mémoire ou de générer du code pour fonctionner dessus, car il peut dire tout ce qu'il pourrait avoir besoin de savoir sur la valeur à partir de son seul type. Mais logiquement, une structure vide est un type avec des valeurs comme les autres, ou plus précisément, un type dont il n'y a qu'une seule valeur : ()

```
let o = Onesuch;
```

Vous avez déjà rencontré une structure de type unité lors de la lecture de l'opérateur de plage dans [« Champs et éléments »](#). Alors qu'une expression like est un raccourci pour la valeur struct , l'expression , une plage omettant les deux points de terminaison, est un raccourci pour la valeur struct de type unité . . . 3 .. 5 Range { start: 3, end: 5 } .. RangeFull

Les structures de type unitaire peuvent également être utiles lorsque vous travaillez avec des traits, que nous décrirons au [chapitre 11](#).

Mise en page de la structure

En mémoire, les structures de champ nommé et de type tuple sont la même chose : une collection de valeurs, de types éventuellement mixtes, disposées d'une manière particulière dans la mémoire. Par exemple, plus haut dans le chapitre, nous avons défini cette structure :

```
struct GrayscaleMap {  
    pixels: Vec<u8>,  
    size: (usize, usize)  
}
```

Une valeur est disposée en mémoire comme schématisé à [la figure 9-1](#).

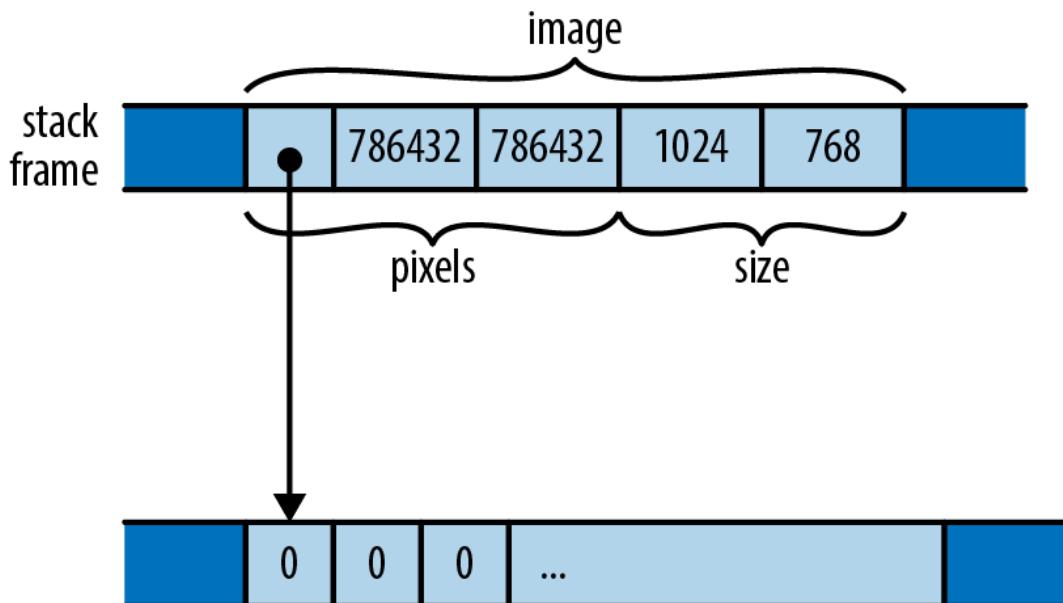


Figure 9-1. Une structure en mémoire `GrayscaleMap`

Contrairement à C et C++, Rust ne fait pas de promesses spécifiques sur la façon dont il ordonnera les champs ou les éléments d'une structure en mémoire ; ce diagramme ne montre qu'un seul arrangement possible. Cependant, Rust promet de stocker les valeurs des champs directement dans le bloc de mémoire de la structure. Alors que JavaScript, Python et Java placeraient chacun les valeurs et dans leurs propres blocs alloués au tas et que les champs de 's pointent vers eux, Rust incorpore et directement dans la valeur. Seul le tampon alloué au tas appartenant au vecteur reste dans son propre

```
bloc. pixels size GrayscaleMap pixels size GrayscaleMap pixels
```

Vous pouvez demander à Rust de disposer des structures d'une manière compatible avec C et C ++, en utilisant l'attribut. Nous aborderons cela en détail au [chapitre 23](#). #[repr(C)]

Définition des méthodes avec impl

Tout au long du livre, nous avons appelé des méthodes sur toutes sortes de valeurs. Nous avons poussé des éléments sur des vecteurs avec , récupéré leur longueur avec , vérifié les valeurs pour les erreurs avec , et ainsi de suite. Vous pouvez également définir des méthodes sur vos propres types de structure. Plutôt que d'apparaître à l'intérieur de la définition struct, comme en C++ ou Java, les méthodes Rust apparaissent dans un bloc séparé.

```
v.push(e) v.len() Result r.expect("msg") impl
```

Un bloc est simplement une collection de définitions, dont chacune devient une méthode sur le type struct nommé en haut du bloc. Ici, par exemple, on définit une structure publique , puis on lui donne deux méthodes publiques, et : impl fn Queue push pop

```
/// A first-in, first-out queue of characters.

pub struct Queue {
    older: Vec<char>, // older elements, eldest last.
    younger: Vec<char> // younger elements, youngest last.
}

impl Queue {
    /// Push a character onto the back of a queue.
    pub fn push(&mut self, c: char) {
        self.younger.push(c);
    }

    /// Pop a character off the front of a queue. Return `Some(c)` if there
    /// was a character to pop, or `None` if the queue was empty.
    pub fn pop(&mut self) -> Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }
        }
    }

    // Bring the elements in younger over to older, and put them in
    // the promised order.
}
```

```

        use std::mem::swap;
        swap(&mut self.older, &mut self.younger);
        self.older.reverse();
    }

    // Now older is guaranteed to have something. Vec's pop method
    // already returns an Option, so we're set.
    self.older.pop()
}
}

```

Les fonctions définies dans un bloc sont *appelées fonctions associées*, car elles sont associées à un type spécifique. Le contraire d'une fonction associée est une *fonction libre*, qui n'est pas définie comme l'élément d'un bloc.

```
impl impl
```

Rust passe à une méthode la valeur sur laquelle elle est appelée comme premier argument, qui doit avoir le nom spécial `.self`. Puisque le type de `self` est évidemment celui nommé en haut du bloc, ou une référence à cela, Rust vous permet d'omettre le type, et d'écrire `, self`, ou comme raccourci pour `, self`, ou `.self`. Vous pouvez utiliser les formulaires longs si vous le souhaitez, mais presque tout le code Rust utilise le raccourci, comme indiqué précédemment.

```
self self impl self &self &mut self self: Queue self:
&Queue self: &mut Queue
```

Dans notre exemple, les méthodes `push` et `pop` se réfèrent aux champs de `'comme` et `.this`. Contrairement à C++ et Java, où les membres de l'objet « `this` » sont directement visibles dans les corps de méthode en tant qu'identificateurs non qualifiés, une méthode Rust doit explicitement utiliser pour faire référence à la valeur sur laquelle elle a été appelée, de la même manière que les méthodes Python utilisent `self`, et la façon dont les méthodes JavaScript utilisent `this`.

```
.push pop Queue self.older self.younger self self this
```

Puisque `self` et `self` ont besoin de modifier le `.older` et `.younger`, ils prennent tous les deux `.self`. Cependant, lorsque vous appelez une méthode, vous n'avez pas besoin d'emprunter vous-même la référence modifiable; la syntaxe d'appel de méthode ordinaire s'en occupe implicitement. Donc, avec ces définitions en place, vous pouvez utiliser comme ceci:

```
push pop Queue &mut
self Queue
```

```
let mut q = Queue { older: Vec::new(), younger: Vec::new() };
```

```

q.push('0');
q.push('1');
assert_eq!(q.pop(), Some('0'));

q.push('∞');
assert_eq!(q.pop(), Some('1'));
assert_eq!(q.pop(), Some('∞'));
assert_eq!(q.pop(), None);

```

Le simple fait d'écrire emprunte une référence mutable à `self`, comme si vous aviez écrit `self.push(...)`, puisque c'est ce que la méthode

exige. `q.push(...)` `q(&mut q).push(...)` `push self`

Si une méthode n'a pas besoin de modifier son `self`, vous pouvez la définir pour prendre une référence partagée à la place. Par exemple: `self`

```

impl Queue {
    pub fn is_empty(&self) -> bool {
        self.older.is_empty() && self.younger.is_empty()
    }
}

```

Encore une fois, l'expression d'appel de méthode sait quel type de référence emprunter :

```

assert!(q.is_empty());
q.push('o');
assert!(!q.is_empty());

```

Ou, si une méthode veut s'approprier `self`, elle peut prendre par valeur : `self self`

```

impl Queue {
    pub fn split(self) -> (Vec<char>, Vec<char>) {
        (self.older, self.younger)
    }
}

```

L'appel de cette méthode ressemble aux autres appels de méthode : `split`

```

let mut q = Queue { older: Vec::new(), younger: Vec::new() };

q.push('P');

```

```

q.push('D');
assert_eq!(q.pop(), Some('P'));
q.push('X');

let (older, younger) = q.split();
// q is now uninitialized.
assert_eq!(older, vec!['D']);
assert_eq!(younger, vec!['X']);

```

Mais notez que, puisque prend sa valeur par, cela *déplace* le hors de , laissant non initialisé. Étant donné qu'il possède maintenant la file d'attente, il est capable d'en retirer les vecteurs individuels et de les renvoyer à l'appelant.

```
split self Queue q q split self
```

Parfois, prendre par valeur comme celle-ci, ou même par référence, ne suffit pas, donc Rust vous permet également de passer via des types de pointeurs intelligents.

```
self self
```

Se passer sous forme de boîte, de rc ou d'arc

L'argument d'une méthode peut également être un , , ou . Une telle méthode ne peut être appelée que sur une valeur du type de pointeur donné. L'appel de la méthode lui transmet la propriété du pointeur.

```
self Box<Self> Rc<Self> Arc<Self>
```

Vous n'aurez généralement pas besoin de le faire. Une méthode qui attend par référence fonctionne correctement lorsqu'elle est appelée sur l'un de ces types de pointeur :

```
self
```

```

let mut bq = Box::new(Queue::new());
// `Queue::push` expects a `&mut Queue`, but `bq` is a `Box<Queue>`.
// This is fine: Rust borrows a `&mut Queue` from the `Box` for the
// duration of the call.
bq.push('■');

```

Pour les appels de méthode et l'accès au champ, Rust emprunte automatiquement une référence à des types de pointeurs tels que , , et , donc et sont presque toujours la bonne chose dans une signature de méthode, avec les .

```
Box Rc Arc &self &mut self self
```

Mais s'il arrive qu'une méthode nécessite la propriété d'un pointeur vers , et que ses appelants ont un tel pointeur à portée de main, Rust vous laissera le passer comme argument de la méthode. Pour ce faire, vous devez

épeler le type de , comme s'il s'agissait d'un paramètre ordinaire

: Self self self

```
impl Node {  
    fn append_to(self: Rc<Self>, parent: &mut Node) {  
        parent.children.push(self);  
    }  
}
```

Fonctions associées au type

Un bloc pour un type donné peut également définir des fonctions qui ne prennent pas du tout comme argument. Ce sont toujours des fonctions associées, car elles sont dans un bloc, mais ce ne sont pas des méthodes, car elles ne prennent pas d'argument. Pour les distinguer des méthodes, nous les appelons *fonctions associées au type*. `impl self impl self`

Ils sont souvent utilisés pour fournir des fonctions de constructeur, comme ceci :

```
impl Queue {  
    pub fn new() -> Queue {  
        Queue { older: Vec::new(), younger: Vec::new() }  
    }  
}
```

Pour utiliser cette fonction, nous l'appelons : le nom du type, un double deux-points, puis le nom de la fonction. Maintenant, notre exemple de code devient un peu plus svelte: `Queue::new`

```
let mut q = Queue::new();  
  
q.push('*');  
...
```

Il est conventionnel dans Rust que les fonctions constructeur soient nommées ; nous avons déjà vu , et d'autres. Mais il n'y a rien de spécial dans le nom. Ce n'est pas un mot-clé, et les types ont souvent d'autres fonctions associées qui servent de constructeurs, comme

`.new Vec::new Box::new HashMap::new new Vec::with_capacity`

Bien que vous puissiez avoir plusieurs blocs distincts pour un seul type, ils doivent tous être dans la même caisse qui définit ce type. Cependant,

Rust vous permet d'attacher vos propres méthodes à d'autres types; nous expliquerons comment dans [le chapitre 11](#). `impl`

Si vous êtes habitué à C++ ou Java, séparer les méthodes d'un type de sa définition peut sembler inhabituel, mais il y a plusieurs avantages à le faire :

- Il est toujours facile de trouver les membres de données d'un type. Dans les grandes définitions de classe C++, vous devrez peut-être parcourir des centaines de lignes de définitions de fonction membre pour vous assurer que vous n'avez manqué aucun des membres de données de la classe ; dans Rust, ils sont tous au même endroit.
- Bien que l'on puisse imaginer adapter des méthodes dans la syntaxe pour les structs de champ nommé, ce n'est pas si soigné pour les structs de type tuple et de type unité. L'extraction de méthodes dans un bloc permet une syntaxe unique pour les trois. En fait, Rust utilise cette même syntaxe pour définir des méthodes sur des types qui ne sont pas du tout des structs, tels que les types et les types primitifs comme `.` (Le fait que n'importe quel type puisse avoir des méthodes est l'une des raisons pour lesquelles Rust n'utilise pas beaucoup le terme *objet*, préférant appeler tout une *valeur*.) `impl enum i32`
- La même syntaxe sert également à implémenter des traits, que nous aborderons dans [le chapitre 11](#). `impl`

Consts associés

Une autre caractéristique des langages comme C# et Java que Rust adopte dans son système de type est l'idée de valeurs associées à un type, plutôt qu'à une instance spécifique de ce type. Dans Rust, ceux-ci sont connus sous le nom *de consts associés*.

Comme son nom l'indique, les consts associés sont des valeurs constantes. Ils sont souvent utilisés pour spécifier les valeurs couramment utilisées d'un type. Par exemple, vous pouvez définir un vecteur bidimensionnel à utiliser en algèbre linéaire avec un vecteur unitaire associé :

```
pub struct Vector2 {
    x: f32,
    y: f32,
}

impl Vector2 {
```

```

    const ZERO: Vector2 = Vector2 { x: 0.0, y: 0.0 };
    const UNIT: Vector2 = Vector2 { x: 1.0, y: 0.0 };
}

```

Ces valeurs sont associées au type lui-même et vous pouvez les utiliser sans faire référence à une autre instance de . Tout comme les fonctions associées, on y accède en nommant le type auquel elles sont associées, suivi de leur nom : `Vector2`

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

Il n'est pas non plus nécessaire qu'un `const` associé soit du même type que le type auquel il est associé ; nous pourrions utiliser cette fonctionnalité pour ajouter des ID ou des noms aux types. Par exemple, s'il y avait plusieurs types similaires à ceux qui devaient être écrits dans un fichier, puis chargés en mémoire ultérieurement, un `const` associé pourrait être utilisé pour ajouter des noms ou des ID numériques qui pourraient être écrits à côté des données pour identifier son type : `Vector2`

```

impl Vector2 {
    const NAME: &'static str = "Vector2";
    const ID: u32 = 18;
}

```

Structures génériques

Notre définition précédente de `Character` est insatisfaisante: il est écrit pour stocker des personnages, mais il n'y a rien sur sa structure ou ses méthodes qui soit spécifique aux personnages. Si nous devions définir une autre structure qui contiendrait, disons, des valeurs, le code pourrait être identique, sauf qu'il serait remplacé par `Character`. Ce serait une perte de temps. `Queue<String>`

Heureusement, rust structs peut être *générique*, ce qui signifie que leur définition est un modèle dans lequel vous pouvez brancher les types que vous voulez. Par exemple, voici une définition pour cela peut contenir des valeurs de n'importe quel type: `Queue`

```

pub struct Queue<T> {
    older: Vec<T>,
    younger: Vec<T>
}

```

Vous pouvez lire l'in comme « pour n'importe quel type d'élément ... ». Donc, cette définition se lit comme suit: « Pour tout type , a est deux champs de type . » Par exemple, dans , est , donc et ont le type . Dans , est , et nous obtenons une structure identique à la définition spécifique avec laquelle nous avons commencé. En fait, elle-même est une structure générique, définie de cette manière.

```
<T> Queue<T> T T Queue<T> Vec<T> Queue<String> T String old  
er younger Vec<String> Queue<char> T char char Vec
```

Dans les définitions de structure génériques, les noms de type utilisés entre crochets sont appelés *paramètres de type*. Un bloc pour une structure générique ressemble à ceci :< > impl

```
impl<T> Queue<T> {  
    pub fn new() -> Queue<T> {  
        Queue { older: Vec::new(), younger: Vec::new() }  
    }  
  
    pub fn push(&mut self, t: T) {  
        self.younger.push(t);  
    }  
  
    pub fn is_empty(&self) -> bool {  
        self.olders.is_empty() && self.youngers.is_empty()  
    }  
  
    ...  
}
```

Vous pouvez lire la ligne comme quelque chose comme, « pour tout type , voici quelques fonctions associées disponibles sur . » Ensuite, vous pouvez utiliser le paramètre type comme type dans les définitions de fonction associées. impl<T> Queue<T> T Queue<T> T

La syntaxe peut sembler un peu redondante, mais le indique clairement que le bloc couvre n'importe quel type , ce qui le distingue d'un bloc écrit pour un type spécifique de , comme celui-

ci: impl<T> impl T impl Queue

```
impl Queue<f64> {  
    fn sum(&self) -> f64 {  
        ...  
    }  
}
```

```
}
```

Cet en-tête de bloc se lit comme suit : « Voici quelques fonctions associées spécifiquement pour . Cela donne une méthode, disponible sur aucun autre type de . `impl Queue<f64> Queue<f64> sum Queue`

Nous avons utilisé le raccourci de Rust pour les paramètres dans le code précédent; écrire partout devient une bouchée et une distraction. Comme un autre raccourci, chaque bloc, générique ou non, définit le paramètre de type spécial (notez le nom) comme étant le type auquel nous ajoutons des méthodes. Dans le code précédent, serait , donc nous pouvons abréger la définition de 'un peu plus

```
loin: self Queue<T> impl Self CamelCase Self Queue<T> Queue::  
new
```

```
pub fn new() -> Self {  
    Queue { older: Vec::new(), younger: Vec::new() }  
}
```

Vous avez peut-être remarqué que, dans le corps de , nous n'avions pas besoin d'écrire le paramètre type dans l'expression de construction ; le simple fait d'écrire était suffisant. C'est l'inférence de type de Rust à l'œuvre : puisqu'il n'y a qu'un seul type qui fonctionne pour la valeur de retour de cette fonction, à savoir, Rust fournit le paramètre pour nous.

Toutefois, vous devrez toujours fournir des paramètres de type dans les signatures de fonction et les définitions de type. La rouille ne les déduit pas; au lieu de cela, il utilise ces types explicites comme base à partir de laquelle il déduit des types dans les corps de fonction. `new Queue { ... }`

`Self` peut également être utilisé de cette manière; nous aurions pu écrire à la place. C'est à vous de décider ce que vous trouvez le plus facile à comprendre. `Self { ... }`

Pour les appels de fonction associés, vous pouvez fournir le paramètre type explicitement à l'aide de la notation (turbofish) `:::<>`

```
let mut q = Queue::<char>::new();
```

Mais dans la pratique, vous pouvez généralement laisser Rust le comprendre pour vous:

```

let mut q = Queue::new();
let mut r = Queue::new();

q.push("CAD"); // apparently a Queue<&'static str>
r.push(0.74); // apparently a Queue<f64>

q.push("BTC"); // Bitcoins per USD, 2019-6
r.push(13764.0); // Rust fails to detect irrational exuberance

```

En fait, c'est exactement ce que nous avons fait avec , un autre type de structure générique, tout au long du livre. `vec`

Il n'y a pas que les structs qui peuvent être génériques. Enums peuvent également prendre des paramètres de type, avec une syntaxe très similaire. Nous le montrerons en détail dans [« Enums »](#).

Structs génériques avec paramètres de durée de vie

Comme nous l'avons vu dans [« Structs Containing References »](#), si un type struct contient des références, vous devez nommer la durée de vie de ces références. Par exemple, voici une structure qui peut contenir des références aux éléments les plus et les moins importants d'une tranche :

```

struct Extrema<'elt> {
    greatest: &'elt i32,
    least: &'elt i32
}

```

Plus tôt, nous vous avons invité à penser à une déclaration comme signifiant que, compte tenu de tout type spécifique, vous pouvez faire un qui contient ce type. De même, vous pouvez penser que cela signifie que, compte tenu de toute durée de vie spécifique, vous pouvez faire un qui contient des références avec cette durée de vie. `struct`

```
Queue<T> T Queue<T> struct Extrema<'elt> 'elt Extrema<'elt>
```

Voici une fonction pour analyser une tranche et renvoyer une valeur dont les champs font référence à ses éléments : `Extrema`

```

fn find_extrema<'s>(slice: &'s [i32]) -> Extrema<'s> {
    let mut greatest = &slice[0];
    let mut least = &slice[0];
}

```

```

        for i in 1..slice.len() {
            if slice[i] < *least { least = &slice[i]; }
            if slice[i] > *greatest { greatest = &slice[i]; }
        }
        Extrema { greatest, least }
    }
}

```

Ici, puisque emprunte des éléments de , qui a la durée de vie , la structure que nous retournons utilise également comme durée de vie de ses références. Rust déduit toujours des paramètres de durée de vie pour les appels, de sorte que les appels n'ont pas besoin de les mentionner: `find_extrema slice` 's `Extrema` 's `find_extrema`

```

let a = [0, -3, 0, 15, 48];
let e = find_extrema(&a);
assert_eq!(*e.least, -3);
assert_eq!(*e.greatest, 48);

```

Parce qu'il est si courant que le type de retour utilise la même durée de vie qu'un argument, Rust nous permet d'omettre les durées de vie lorsqu'il y a un candidat évident. Nous aurions également pu écrire la signature de 'comme ceci, sans changement de sens: `find_extrema`

```

fn find_extrema(slice: &[i32]) -> Extrema {
    ...
}

```

Certes, nous *aurions pu* vouloir dire, mais c'est assez inhabituel. Rust fournit un raccourci pour le cas commun. `Extrema<'static>`

Structs génériques avec des paramètres constants

Une structure générique peut également prendre des paramètres qui sont des valeurs constantes. Par exemple, vous pouvez définir un type représentant des polynômes de degré arbitraire comme suit :

```

/// A polynomial of degree N - 1.
struct Polynomial<const N: usize> {
    /// The coefficients of the polynomial.
    ///

```

```

    /// For a polynomial a + bx + cx2 + ... + zxn-1,
    /// the `i`'th element is the coefficient of xi.
    coefficients: [f64; N]
}

```

Avec cette définition, est un polynôme quadratique, par exemple. La clause indique que le type attend une valeur comme paramètre générique, qu'il utilise pour décider du nombre de coefficients à stocker. `Polynomial<3> <const N: usize> Polynomial<N>`

Contrairement à , qui a des champs tenant sa longueur et sa capacité et stocke ses éléments dans le tas, stocke ses coefficients directement dans la valeur, et rien d'autre. La longueur est donnée par le type. (La capacité n'est pas nécessaire, car s ne peut pas croître dynamiquement.) `Vec<Polynomial<N>>`

Nous pouvons utiliser le paramètre dans les fonctions associées au type : `N`

```

impl<const N: usize> Polynomial<N> {
    fn new(coefficients: [f64; N]) -> Polynomial<N> {
        Polynomial { coefficients }
    }

    /// Evaluate the polynomial at `x`.
    fn eval(&self, x: f64) -> f64 {
        // Horner's method is numerically stable, efficient, and simple
        // c0 + x(c1 + x(c2 + x(c3 + ... x(c[n-1] + x c[n]))))
        let mut sum = 0.0;
        for i in (0..N).rev() {
            sum = self.coefficients[i] + x * sum;
        }

        sum
    }
}

```

Ici, la fonction accepte un tableau de longueur , et prend ses éléments comme coefficients d'une valeur fraîche. La méthode itère sur la plage pour trouver la valeur du polynôme en un point donné. `new N Polynomial eval 0..N x`

Comme pour les paramètres de type et de durée de vie, Rust peut souvent déduire les bonnes valeurs pour des paramètres constants:

```

use std::f64::consts::FRAC_PI_2;      // π/2

// Approximate the `sin` function: sin x ≈ x - 1/6 x3 + 1/120 x5
// Around zero, it's pretty accurate!
let sine_poly = Polynomial::new([0.0, 1.0, 0.0, -1.0/6.0, 0.0,
                                 1.0/120.0]);
assert_eq!(sine_poly.eval(0.0), 0.0);
assert!((sine_poly.eval(FRAC_PI_2) - 1.).abs() < 0.005);

```

Puisque nous passons un tableau avec six éléments, Rust sait que nous devons construire un . La méthode sait combien d'itérations la boucle doit exécuter simplement en consultant son type. Étant donné que la longueur est connue au moment de la compilation, le compilateur remplacera probablement la boucle entièrement par du code en ligne droite. `Polynomial::new Polynomial<6> eval for Self`

Un paramètre générique peut être n'importe quel type entier, , ou . Les nombres à virgule flottante, les énumérations et autres types ne sont pas autorisés. `const char bool`

Si la structure prend d'autres types de paramètres génériques, les paramètres de durée de vie doivent venir en premier, suivis des types, suivis de toutes les valeurs. Par exemple, un type qui contient un tableau de références peut être déclaré comme suit : `const`

```

struct LumpOfReferences<'a, T, const N: usize> {
    the_lump: [&'a T; N]
}

```

Les paramètres génériques constants sont un ajout relativement nouveau à Rust, et leur utilisation est quelque peu restreinte pour l'instant. Par exemple, il aurait été plus agréable de définir ainsi : `Polynomial`

```

/// A polynomial of degree N.
struct Polynomial<const N: usize> {
    coefficients: [f64; N + 1]
}

```

Cependant, Rust rejette cette définition :

```

error: generic parameters may not be used in const operations
|
6 |     coefficients: [f64; N + 1]

```

```
| ^ cannot perform const operation using `N`  
|  
= help: const parameters may only be used as standalone arguments, i.
```

Bien qu'il soit bon de le dire, un type comme est apparemment trop risqué pour Rust. Mais Rust impose cette restriction pour le moment afin d'éviter de faire face à des problèmes comme celui-ci: [f64; N] [f64; N + 1]

```
struct Ketchup<const N: usize> {  
    tomayto: [i32; N & !31],  
    tomahto: [i32; N - (N % 32)],  
}
```

Il s'avère que et sont égaux pour toutes les valeurs de , donc et ont toujours le même type. Il devrait être permis d'attribuer l'un à l'autre, par exemple. Mais enseigner au vérificateur de type de Rust l'algèbre de bit-fiddling dont il aurait besoin pour être capable de reconnaître ce fait risque d'introduire des cas de coin déroutants à un aspect du langage qui est déjà assez compliqué. Bien sûr, des expressions simples comme sont beaucoup plus bien conduites, et il y a du travail en cours pour apprendre à Rust à les gérer en douceur. N & !31 N - (N % 32) N tomayto tomahto N + 1

Étant donné que le problème ici concerne le comportement du vérificateur de types, cette restriction s'applique uniquement aux paramètres constants apparaissant dans les types, comme la longueur d'un tableau. Dans une expression ordinaire, vous pouvez utiliser comme vous le souhaitez: et sont parfaitement acceptables. N N + 1 N & !31

Si la valeur que vous souhaitez fournir pour un paramètre générique n'est pas simplement un littéral ou un identificateur unique, vous devez l'encapsuler entre accolades, comme dans . Cette règle permet à Rust de signaler les erreurs de syntaxe avec plus de précision. const Polynomial<{5 + 1}>

Dérivation de traits communs pour les types Struct

Les structs peuvent être très faciles à écrire :

```
struct Point {  
    x: f64,  
    y: f64  
}
```

Cependant, si vous deviez commencer à utiliser ce type, vous remarqueriez rapidement que c'est un peu pénible. Tel qu'il est écrit, n'est ni copiable ni clonable. Vous ne pouvez pas l'imprimer avec et il ne prend pas en charge les opérateurs et. `Point Point println!("{}:{}", point); == !=`

Chacune de ces fonctionnalités a un nom dans Rust—,, et . Ils sont *appelés traits*. Dans [le chapitre 11](#), nous montrerons comment implémenter des traits à la main pour vos propres structures. Mais dans le cas de ces traits standard, et de plusieurs autres, vous n'avez pas besoin de les implémenter à la main, sauf si vous voulez une sorte de comportement personnalisé. Rust peut les mettre en œuvre automatiquement pour vous, avec une précision mécanique. Il suffit d'ajouter un attribut à la

```
struct: Copy Clone Debug PartialEq #[derive]
```

```
#[derive(Copy, Clone, Debug, PartialEq)]  
struct Point {  
    x: f64,  
    y: f64  
}
```

Chacun de ces traits peut être implémenté automatiquement pour une structure, à condition que chacun de ses champs implémente le trait. On peut demander à Rust de dériver car ses deux champs sont tous deux de type , ce qui implémente déjà . `PartialEq Point f64 PartialEq`

Rust peut également dériver , ce qui ajouterait un support pour les opérateurs de comparaison ,,, et . Nous ne l'avons pas fait ici, car comparer deux points pour voir si l'un est « inférieur » à l'autre est en fait une chose assez étrange à faire. Il n'y a pas d'ordre conventionnel sur les points. Nous choisissons donc de ne pas soutenir ces opérateurs pour des valeurs. Des cas comme celui-ci sont l'une des raisons pour lesquelles Rust nous fait écrire l'attribut plutôt que de dériver automatiquement tous les traits qu'il peut. Une autre raison est que l'implémentation d'un trait est automatiquement une fonctionnalité publique, donc la copabilité, la clonabilité, etc. font toutes partie de l'API publique de votre structure et

```
douivent être choisies délibérément. PartialOrd < > <= >= Point #  
[derive]
```

Nous décrirons en détail les traits standard de Rust et expliquerons lesquels sont capables dans [le chapitre 13](#). #[derive]

Mutabilité intérieure

La mutabilité est comme n'importe quoi d'autre: en excès, cela cause des problèmes, mais vous en voulez souvent juste un peu. Par exemple, supposons que votre système de contrôle de robot araignée ait une structure centrale, , qui contient des paramètres et des poignées d'E/S. Il est configuré lorsque le robot démarre et les valeurs ne changent jamais

: SpiderRobot

```
pub struct SpiderRobot {  
    species: String,  
    web_enabled: bool,  
    leg_devices: [fd::FileDesc; 8],  
    ...  
}
```

Chaque système majeur du robot est géré par une structure différente, et chacun a un pointeur vers le : SpiderRobot

```
use std::rc::Rc;  
  
pub struct SpiderSenses {  
    robot: Rc<SpiderRobot>, // <-- pointer to settings and I/O  
    eyes: [Camera; 32],  
    motion: Accelerometer,  
    ...  
}
```

Les structures pour la construction de bandes, la prédation, le contrôle du flux de venin, etc. ont également toutes un pointeur intelligent. Rappelons que signifie [comptage de référence](#), et une valeur dans une boîte est toujours partagée et donc toujours immuable. `Rc<SpiderRobot>` `Rc` `Rc`

Supposons maintenant que vous souhaitez ajouter un peu de journalisation à la structure, en utilisant le type standard. Il y a un problème : un doit être . Toutes les méthodes pour y écrire nécessitent une référence. `SpiderRobot File` `File` `mut` `mut`

Ce genre de situation revient assez souvent. Ce dont nous avons besoin, c'est d'un peu de données mutables (a) à l'intérieur d'une valeur autrement immuable (la struct). C'est ce qu'on appelle *la mutabilité intérieure*. La rouille en offre plusieurs saveurs; Dans cette section, nous aborderons les deux types les plus simples : et , les deux dans le module.

```
File SpiderRobot Cell<T> RefCell<T> std::cell
```

A est une structure qui contient une seule valeur privée de type . La seule particularité de a est que vous pouvez obtenir et définir le champ même si vous n'avez pas accès à lui-même:

```
Cell<T> T Cell mut Cell
```

Cell::new(value)

Crée un nouveau , en y déplaçant le donné.

cell.get()

Renvoie une copie de la valeur dans le fichier .

cell.set(value)

Stocke la donnée dans le , en supprimant la valeur précédemment stockée.

Cette méthode prend comme non-référence: *self mut*

```
fn set(&self, value: T) // note: not `&mut self`
```

Ceci est, bien sûr, inhabituel pour les méthodes nommées . À l'heure actuelle, Rust nous a entraînés à nous attendre à ce que nous ayons besoin d'un accès si nous voulons apporter des modifications aux données. Mais de la même manière, ce détail inhabituel est tout l'intérêt de s. Ils sont simplement un moyen sûr de contourner les règles sur l'immuabilité , ni plus, ni moins.

set mut Cell

Les cellules ont également quelques autres méthodes, que vous pouvez lire [dans la documentation](#).

A serait pratique si vous ajoutiez un simple compteur à votre . Vous pourriez écrire :

```
use std::cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count: Cell<u32>,
```

```
    ...
}
```

Ensuite, même les non-méthodes de peuvent accéder à cela, en utilisant les méthodes et: `mut SpiderRobot u32 .get() .set()`

```
impl SpiderRobot {
    // Increase the error count by 1.
    pub fn add_hardware_error(&self) {
        let n = self.hardware_error_count.get();
        self.hardware_error_count.set(n + 1);
    }

    // True if any hardware errors have been reported.
    pub fn has_hardware_errors(&self) -> bool {
        self.hardware_error_count.get() > 0
    }
}
```

C'est assez facile, mais cela ne résout pas notre problème de journalisation. ne vous permet *pas* d'appeler des méthodes sur une valeur partagée. La méthode renvoie une copie de la valeur dans la cellule, elle ne fonctionne donc que si elle implémente [le caractère Copy](#). Pour la journalisation, nous avons besoin d'un mutable , et [le fichier](#) n'est pas copiable. `Cell mut .get() T File`

Le bon outil dans ce cas est un fichier . Like , est un type générique qui contient une seule valeur de type . Contrairement à , prend en charge l'emprunt de références à sa

valeur: `RefCell Cell<T> RefCell<T> T Cell RefCell T`

`RefCell::new(value)`

Crée un nouveau , se déplaçant dedans. `RefCell value`

`ref_cell.borrow()`

Renvoie un , qui est essentiellement une référence partagée à la valeur stockée dans . `Ref<T> ref_cell`

Cette méthode panique si la valeur est déjà empruntée de manière mutable; voir les détails à suivre.

`ref_cell.borrow_mut()`

Renvoie un , essentiellement une référence modifiable à la valeur dans . `RefMut<T> ref_cell`

Cette méthode panique si la valeur est déjà empruntée; voir les détails à suivre.

```
ref_cell.try_borrow(), ref_cell.try_borrow_mut()
```

Travaillez comme et , mais retournez un fichier . Au lieu de paniquer si la valeur est déjà empruntée de manière mutable, ils renvoient une valeur. borrow() borrow_mut() Result Err

Encore une fois, a quelques autres méthodes, que vous pouvez trouver [dans la documentation](#). RefCell

Les deux méthodes ne paniquent que si vous essayez d'enfreindre la règle Rust selon laquelle les références sont des références exclusives. Par exemple, cela paniquerait : borrow mut

```
use std::cell::RefCell;

let ref_cell: RefCell<String> = RefCell::new("hello".to_string());

let r = ref_cell.borrow();          // ok, returns a Ref<String>
let count = r.len();                // ok, returns "hello".len()
assert_eq!(count, 5);

let mut w = ref_cell.borrow_mut();   // panic: already borrowed
w.push_str(" world");
```

Pour éviter de paniquer, vous pouvez placer ces deux emprunts dans des blocs séparés. De cette façon, serait abandonné avant d'essayer d'emprunter . r w

Cela ressemble beaucoup au fonctionnement des références normales. La seule différence est que normalement, lorsque vous empruntez une référence à une variable, Rust vérifie *au moment de la compilation* pour s'assurer que vous utilisez la référence en toute sécurité. Si les vérifications échouent, vous obtenez une erreur du compilateur. applique la même règle à l'aide de vérifications d'exécution. Donc, si vous enfreignez les règles, vous obtenez une panique (ou un , pour et). RefCell Err try_borrow try_borrow_mut

Maintenant, nous sommes prêts à mettre au travail dans notre type: RefCell SpiderRobot

```
pub struct SpiderRobot {
```

```
    ...
```

```

    log_file: RefCell<File>,
    ...
}

impl SpiderRobot {
    // Write a line to the log file.
    pub fn log(&self, message: &str) {
        let mut file = self.log_file.borrow_mut();
        // `writeln!` is like `println!`, but sends
        // output to the given file.
        writeln!(file, "{}", message).unwrap();
    }
}

```

La variable a le type . Il peut être utilisé comme une référence mutable à un fichier . Pour plus d'informations sur l'écriture dans des fichiers, [re-portez-vous au chapitre 18.](#) file RefMut<File> File

Les cellules sont faciles à utiliser. Avoir à appeler et ou et .borrow_mut() est légèrement gênant, mais c'est juste le prix que nous payons pour contourner les règles. L'autre inconvénient est moins évident et plus grave : les cellules – et tous les types qui en contiennent – ne sont pas thread-safe. Rust [ne permettra donc pas à](#) plusieurs threads d'y accéder à la fois. Nous décrirons les saveurs de mutabilité intérieure sans danger pour le fil dans le [chapitre 19](#), lorsque nous discuterons [de « Mutex<T> », « Atomics » et « Global Variables »](#). .get() .set() .borrow()

Qu'une struct ait nommé des champs ou qu'elle soit en forme de tuple, c'est une agrégation d'autres valeurs : si j'ai une struct, alors j'ai un pointeur vers une struct partagée, et j'ai des yeux, et j'ai un accéléromètre, et ainsi de suite. Donc, l'essence d'une struct est le mot « et »: j'ai un X et un Y. Mais que se passerait-il s'il y avait un autre type construit autour du mot « ou »? C'est-à-dire que lorsque vous avez une valeur d'un tel type, vous auriez *un X ou un Y?* De tels types s'avèrent si utiles qu'ils sont omniprésents dans Rust, et ils font l'objet du chapitre suivant. [SpiderSenses Rc SpiderRobot](#)

Chapitre 10. Enums et motifs

Surprenant de voir à quel point les choses informatiques ont du sens vu comme une privation tragique de types de somme (cf. privation de lambdas).

—[Graydon Hoare](#)

Le premier sujet de ce chapitre est puissant, aussi vieux que les collines, heureux de vous aider à faire beaucoup en peu de temps (pour un prix), et connu sous de nombreux noms dans de nombreuses cultures. Mais ce n'est pas le diable. C'est une sorte de type de données défini par l'utilisateur, connu depuis longtemps par les pirates ML et Haskell comme des types de somme, des unions discriminées ou des types de données algébriques. Dans Rust, ils sont *appelés énumérations*, ou simplement *enums*. Contrairement au diable, ils sont tout à fait en sécurité, et le prix qu'ils demandent n'est pas une grande privation.

C++ et C# ont des enums ; vous pouvez les utiliser pour définir votre propre type dont les valeurs sont un ensemble de constantes nommées. Par exemple, vous pouvez définir un type nommé avec des valeurs , , , etc. Ce genre d'enum fonctionne aussi dans Rust. Mais Rust pousse les enums beaucoup plus loin. Un enum Rust peut également contenir des données, même des données de différents types. Par exemple, le type de Rust est un enum; une telle valeur est soit une valeur contenant un, soit une valeur contenant un . C'est au-delà de ce que les enums C++ et C# peuvent faire. C'est plus comme un C, mais contrairement aux unions, les enums Rust sont sans danger pour le

```
type. Color Red Orange Yellow Result<String,  
io::Error> Ok String Err io::Error union
```

Les enums sont utiles chaque fois qu'une valeur peut être une chose ou une autre. Le « prix » de leur utilisation est que vous devez accéder aux données en toute sécurité, en utilisant la correspondance de modèles, notre sujet pour la deuxième moitié de ce chapitre.

Les modèles peuvent également être familiers si vous avez utilisé le déballage en Python ou la déstructuration en JavaScript, mais Rust pousse les modèles plus loin. Les motifs de rouille sont un peu comme des expressions régulières pour toutes vos données. Ils sont utilisés pour tester si une valeur a ou non une forme souhaitée particulière. Ils peuvent extraire plusieurs champs d'une structure ou d'un tuple en variables locales

en une seule fois. Et comme les expressions régulières, elles sont concises, faisant généralement tout cela en une seule ligne de code.

Ce chapitre commence par les bases des enums, montrant comment les données peuvent être associées à des variantes enum et comment les enums sont stockés en mémoire. Ensuite, nous montrerons comment les modèles et les instructions de Rust peuvent spécifier de manière concise la logique basée sur des enums, des structs, des tableaux et des tranches. Les modèles peuvent également inclure des références, des mouvements et des conditions, ce qui les rend encore plus performants. `match if`

Enums

Les enums simples de style C sont simples:

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

Cela déclare un type avec trois valeurs possibles, *appelées variantes ou constructeurs* : , et . Cet enum particulier fait partie de la bibliothèque standard, de sorte que le code Rust peut l'importer, soit par lui-même: `use std::cmp::Ordering;`

```
fn compare(n: i32, m: i32) -> Ordering {
    if n < m {
        Ordering::Less
    } else if n > m {
        Ordering::Greater
    } else {
        Ordering::Equal
    }
}
```

ou avec tous ses constructeurs :

```
use std::cmp::Ordering::{self, *};      // `*` to import all children

fn compare(n: i32, m: i32) -> Ordering {
```

```

if n < m {
    Less
} else if n > m {
    Greater
} else {
    Equal
}
}

```

Après avoir importé les constructeurs, nous pouvons écrire à la place de , et ainsi de suite, mais comme c'est moins explicite, il est généralement considéré comme un meilleur style de *ne pas* les importer, sauf lorsque cela rend votre code beaucoup plus lisible. `Less Ordering::Less`

Pour importer les constructeurs d'un énumération déclaré dans le module actif, utilisez une importation : `self`

```

enum Pet {
    Orca,
    Giraffe,
    ...
}

use self::Pet::*;


```

En mémoire, les valeurs des enums de style C sont stockées sous forme d'entiers. Parfois, il est utile de dire à Rust quels entiers utiliser:

```

enum HttpStatus {
    Ok = 200,
    NotModified = 304,
    NotFound = 404,
    ...
}

```

Sinon, Rust vous attribuera les numéros, à partir de 0.

Par défaut, Rust stocke les énumérations de style C en utilisant le plus petit type entier intégré qui peut les accueillir. La plupart tiennent dans un seul octet:

```

use std::mem::size_of;
assert_eq!(size_of::<Ordering>(), 1);
assert_eq!(size_of::<HttpStatus>(), 2); // 404 doesn't fit in a u8

```

Vous pouvez remplacer le choix de représentation en mémoire de Rust en ajoutant un attribut à l'énumération. Pour plus d'informations, [consultez « Recherche de représentations de données communes ».](#) #[repr]

La conversion d'un enum de style C en un entier est autorisée :

```
assert_eq!(HttpStatus::Ok as i32, 200);
```

Cependant, la coulée dans l'autre sens, de l'entier à l'enum, ne l'est pas. Contrairement à C et C++, Rust garantit qu'une valeur enum n'est jamais qu'une des valeurs énoncées dans la déclaration. Une conversion non cochée d'un type entier vers un type enum pourrait briser cette garantie, elle n'est donc pas autorisée. Vous pouvez soit écrire votre propre conversion cochée : enum

```
fn http_status_from_u32(n: u32) -> Option<HttpStatus> {
    match n {
        200 => Some(HttpStatus::Ok),
        304 => Some(HttpStatus::NotModified),
        404 => Some(HttpStatus::NotFound),
        ...
        _ => None,
    }
}
```

ou utilisez [la caisse enum primitive](#). Il contient une macro qui génère automatiquement ce type de code de conversion pour vous.

Comme pour les structs, le compilateur implémentera des fonctionnalités telles que l'opérateur pour vous, mais vous devez demander: ==

```
#[derive(Copy, Clone, Debug, PartialEq, Eq)]
enum TimeUnit {
    Seconds, Minutes, Hours, Days, Months, Years,
}
```

Les enums peuvent avoir des méthodes, tout comme les structs :

```
impl TimeUnit {
    /// Return the plural noun for this time unit.
    fn plural(self) -> &'static str {
        match self {
            TimeUnit::Seconds => "seconds",
            TimeUnit::Minutes => "minutes",
            TimeUnit::Hours => "hours",
        }
    }
}
```

```

        TimeUnit::Days => "days",
        TimeUnit::Months => "months",
        TimeUnit::Years => "years",
    }
}

/// Return the singular noun for this time unit.
fn singular(self) -> &'static str {
    self.plural().trim_end_matches('s')
}
}

```

Voilà pour les enums de style C. Le type le plus intéressant de Rust enum est celui dont les variantes contiennent des données. Nous montrerons comment ceux-ci sont stockés en mémoire, comment les rendre génériques en ajoutant des paramètres de type et comment créer des structures de données complexes à partir d'enums.

Enums avec données

Certains programmes doivent toujours afficher les dates et les heures complètes jusqu'à la milliseconde, mais pour la plupart des applications, il est plus convivial d'utiliser une approximation approximative, comme « il y a deux mois ». Nous pouvons écrire un enum pour aider à cela, en utilisant l'enum défini précédemment:

```

/// A timestamp that has been deliberately rounded off, so our program
/// says "6 months ago" instead of "February 9, 2016, at 9:49 AM".
#[derive(Copy, Clone, Debug, PartialEq)]
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}

```

Deux des variantes de cet enum, et , prennent des arguments. Celles-ci sont *appelées variantes de tuple*. Comme les tuple structs, ces constructeurs sont des fonctions qui créent de nouvelles valeurs

```
: InThePast InTheFuture RoughTime
```

```

let four_score_and_seven_years_ago =
    RoughTime::InThePast(TimeUnit::Years, 4 * 20 + 7);

let three_hours_from_now =
    RoughTime::InTheFuture(TimeUnit::Hours, 3);

```

Les enums peuvent également avoir des *variantes struct*, qui contiennent des champs nommés, tout comme les structs ordinaires :

```
enum Shape {
    Sphere { center: Point3d, radius: f32 },
    Cuboid { corner1: Point3d, corner2: Point3d },
}

let unit_sphere = Shape::Sphere {
    center: ORIGIN,
    radius: 1.0,
};
```

En tout, Rust a trois types de variante enum, faisant écho aux trois types de struct que nous avons montrés dans le chapitre précédent. Les variantes sans données correspondent à des structures de type unitaire. Les variantes de tuple ressemblent et fonctionnent comme des structures de tuple. Les variantes Struct ont des accolades et des champs nommés. Un seul enum peut avoir des variantes des trois types:

```
enum RelationshipStatus {
    Single,
    InARelationship,
    ItsComplicated(Option<String>),
    ItsExtremelyComplicated {
        car: DifferentialEquation,
        cdr: EarlyModernistPoem,
    },
}
```

Tous les constructeurs et champs d'un enum partagent la même visibilité que l'enum lui-même.

Enums en mémoire

En mémoire, les énumérations avec des données sont stockées sous la forme d'une petite *balise* entière, plus suffisamment de mémoire pour contenir tous les champs de la plus grande variante. Le champ de balise est destiné à l'usage interne de Rust. Il indique quel constructeur a créé la valeur et donc quels champs il possède.

À partir de Rust 1.56, tient dans 8 octets, comme illustré à [la figure 10-1](#). RoughTime

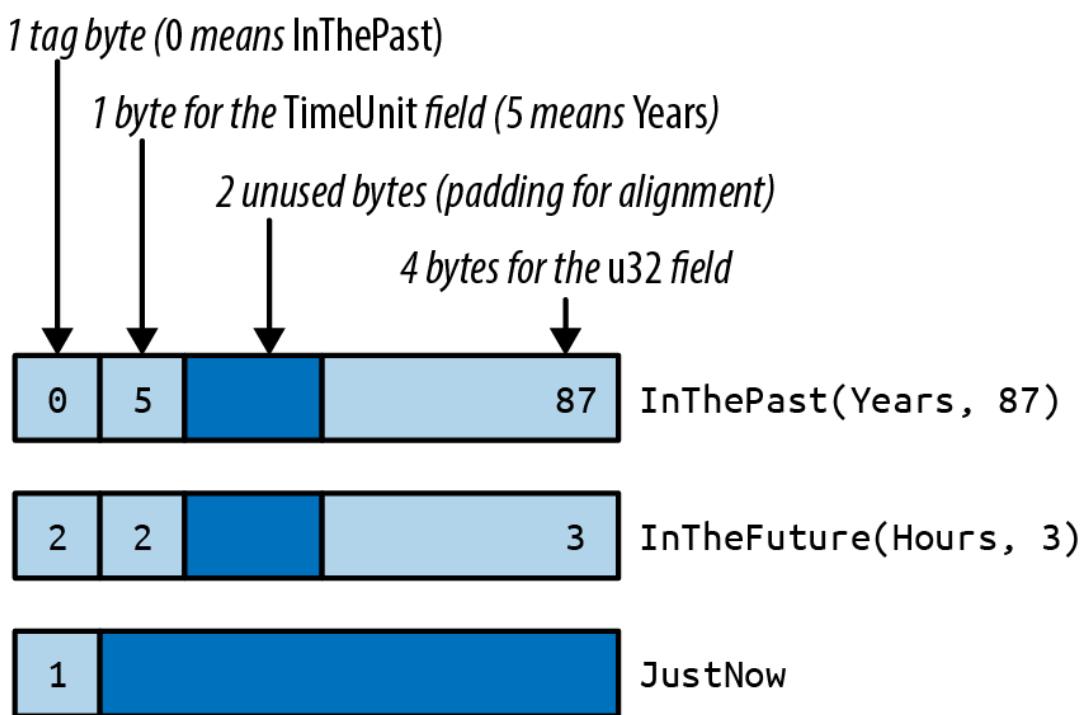


Figure 10-1. valeurs en mémoire RoughTime

Rust ne fait cependant aucune promesse sur la disposition de l'enum afin de laisser la porte ouverte à de futures optimisations. Dans certains cas, il serait possible d'emballer un enum plus efficacement que ne le suggère le chiffre. Par exemple, certaines structures génériques peuvent être stockées sans balise, comme nous le verrons plus tard.

Structures de données enrichies à l'aide d'enums

Les enums sont également utiles pour implémenter rapidement des structures de données arborescentes. Par exemple, supposons qu'un programme Rust doive fonctionner avec des données JSON arbitraires. En mémoire, tout document JSON peut être représenté sous la forme d'une valeur de ce type Rust :

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

enum Json {
    Null,
    Boolean(bool),
    Number(f64),
    String(String),
    Array(Vec<Json>),
    Object(Box<HashMap<String, Json>>),
}
```

L'explication de cette structure de données en anglais ne peut pas améliorer beaucoup le code Rust. La norme JSON spécifie les différents types de données qui peuvent apparaître dans un document JSON :,

valeurs booléennes, nombres, chaînes, tableaux de valeurs JSON et objets avec des clés de chaîne et des valeurs JSON. L'enum énonce simplement ces types. `null Json`

Ce n'est pas un exemple hypothétique. Un enum très similaire peut être trouvé dans , une bibliothèque de sérialisation pour Rust structs qui est l'une des caisses les plus téléchargées sur crates.io. `serde_json`

L'entourage de ce qui représente un `ne` sert qu'à rendre toutes les valeurs plus compactes. En mémoire, les valeurs de type occupent quatre mots machine. et les valeurs sont trois mots, et Rust ajoute un octet de balise. et les valeurs ne contiennent pas suffisamment de données pour utiliser tout cet espace, mais toutes les valeurs doivent avoir la même taille. L'espace supplémentaire n'est pas utilisé. [La figure 10-2](#) montre quelques exemples de l'apparence réelle des valeurs en mémoire.

```
Box HashMap Object Json Json String Vec Null Boolean Json
```

A est encore plus grand. Si nous devions laisser de la place pour cela dans chaque valeur, ils seraient assez grands, huit mots environ. Mais a est un seul mot : c'est juste un pointeur vers des données allouées en tas. Nous pourrions rendre encore plus compact en boxant plus de terrains.

```
HashMap Json Box<HashMap> Json
```

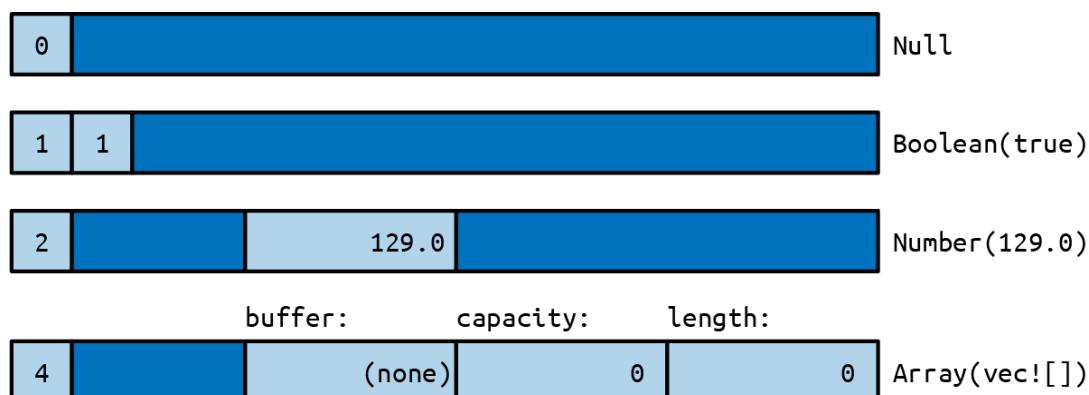


Figure 10-2. valeurs en mémoire JSON

Ce qui est remarquable ici, c'est à quel point il a été facile de mettre cela en place. En C++, on peut écrire une classe pour ceci :

```
class JSON {
private:
    enum Tag {
        Null, Boolean, Number, String, Array, Object
    };
    union Data {
        bool boolean;
        double number;
```

```

        shared_ptr<string> str;
        shared_ptr<vector<JSON>> array;
        shared_ptr<unordered_map<string, JSON>> object;

    Data() {}
    ~Data() {}
    ...
};

Tag tag;
Data data;

public:
    bool is_null() const { return tag == Null; }
    bool is_boolean() const { return tag == Boolean; }
    bool get_boolean() const {
        assert(is_boolean());
        return data.boolean;
    }
    void set_boolean(bool value) {
        this->~JSON(); // clean up string/array/object value
        tag = Boolean;
        data.boolean = value;
    }
    ...
};


```

À 30 lignes de code, nous avons à peine commencé le travail. Cette classe aura besoin de constructeurs, d'un destructeur et d'un opérateur d'affectation. Une alternative serait de créer une hiérarchie de classes avec une classe de base et des sous-classes , , et ainsi de suite. Quoi qu'il en soit, lorsque cela sera fait, notre bibliothèque JSON C ++ aura plus d'une douzaine de méthodes. Il faudra un peu de lecture pour que d'autres programmeurs le prennent et l'utilisent. L'ensemble de Rust enum est composé de huit lignes de code. JSON JSONBoolean JSONString

Enums génériques

Enums peut être générique. Deux exemples tirés de la bibliothèque standard comptent parmi les types de données les plus utilisés dans la langue :

```

enum Option<T> {
    None,
    Some(T),
}
```

```

enum Result<T, E> {
    Ok(T),
    Err(E),
}

```

Ces types sont maintenant assez familiers, et la syntaxe pour les enums génériques est la même que pour les structs génériques.

Un détail peu évident est que Rust peut éliminer le champ de balise lorsque le type est une référence, , ou un autre type de pointeur intelligent. Étant donné qu'aucun de ces types de pointeurs n'est autorisé à être nul, Rust peut représenter, disons, comme un seul mot machine: 0 pour et non nul pour pointeur. Cela rend ces types proches des analogues aux valeurs de pointeur C ou C ++ qui pourraient être nulles. La différence est que le système de type de Rust vous oblige à vérifier qu'un est avant de pouvoir utiliser son contenu. Cela élimine efficacement les déréférencements de pointeur

```
nuls. Option<T> T Box Option<Box<i32>> None Some Option Option Some
```

Les structures de données génériques peuvent être construites avec seulement quelques lignes de code :

```

// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}

```

Ces quelques lignes de code définissent un type qui peut stocker n'importe quel nombre de valeurs de type . BinaryTree T

Beaucoup d'informations sont emballées dans ces deux définitions, nous prendrons donc le temps de traduire le code mot à mot en anglais.

Chaque valeur est soit ou . Si c'est , alors il ne contient aucune donnée du tout. Si , alors il a un , un pointeur vers un tas alloué

```
. BinaryTree Empty NonEmpty Empty NonEmpty Box TreeNode
```

Chaque valeur contient un élément réel, ainsi que deux autres valeurs. Cela signifie qu'un arbre peut contenir des sous-arbres, et donc un arbre peut avoir n'importe quel nombre de descendants. `TreeNode` `BinaryTree` `NonEmpty`

Une esquisse d'une valeur de type est illustrée à [la figure 10-3](#). Comme avec , Rust élimine le champ de balise, de sorte qu'une valeur n'est qu'un mot de machine. `BinaryTree<&str>` `Option<Box<T>>` `BinaryTree`

La création d'un nœud particulier dans cette arborescence est simple :

```
use self::BinaryTree::*;

let jupiter_tree = NonEmpty(Box::new(TreeNode {
    element: "Jupiter",
    left: Empty,
    right: Empty,
}));
```

Les arbres plus grands peuvent être construits à partir de plus petits:

```
let mars_tree = NonEmpty(Box::new(TreeNode {
    element: "Mars",
    left: jupiter_tree,
    right: mercury_tree,
}));
```

Naturellement, cette affectation transfère la propriété de et vers leur nouveau nœud parent. `jupiter_node` `mercury_node`

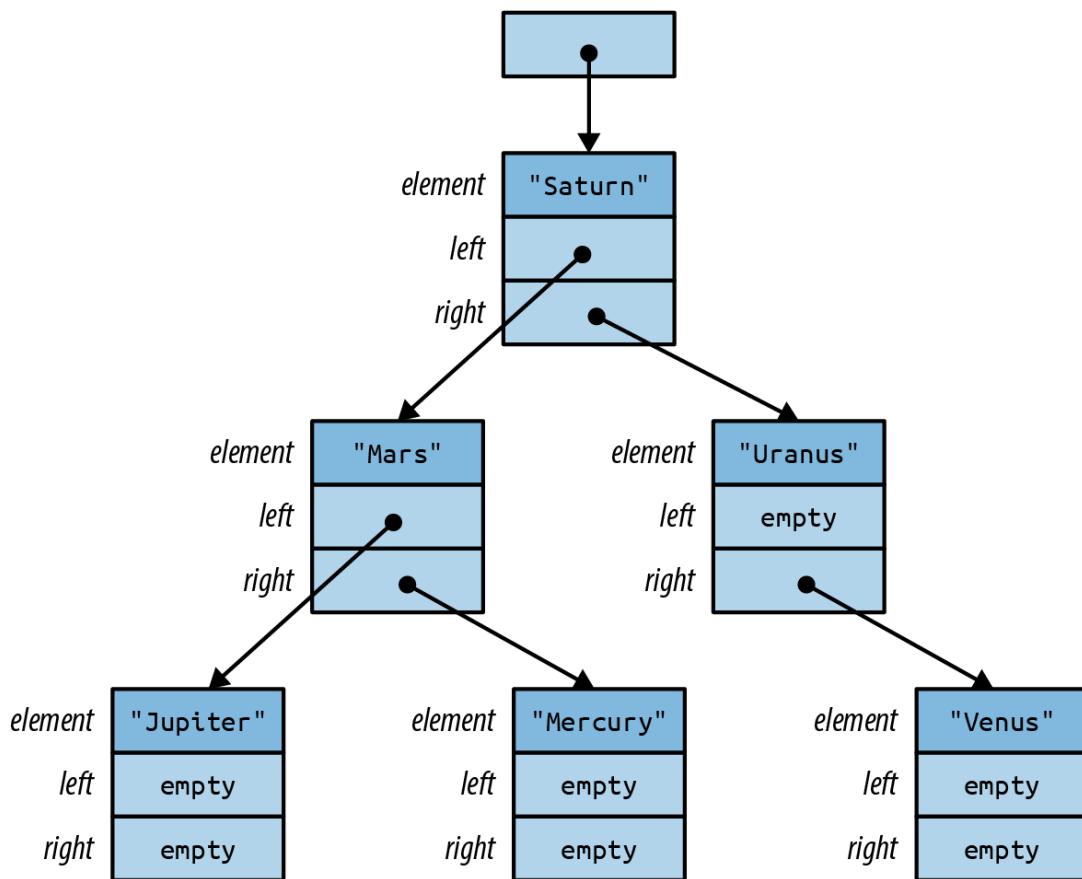


Figure 10-3. A contenant six chaînes BinaryTree

Les parties restantes de l'arbre suivent les mêmes schémas. Le nœud racine n'est pas différent des autres :

```

let tree = NonEmpty(Box::new(TreeNode {
    element: "Saturn",
    left: mars_tree,
    right: uranus_tree,
}));
```

Plus loin dans ce chapitre, nous montrerons comment implémenter une méthode sur le type afin que nous puissions écrire à la place

: add BinaryTree

```

let mut tree = BinaryTree::Empty;
for planet in planets {
    tree.add(planet);
}
```

Quelle que soit la langue d'où vous venez, la création de structures de données comme dans Rust nécessitera probablement un peu de pratique. Il ne sera pas évident au début où mettre les es. Une façon de trouver un design qui fonctionnera est de dessiner une image comme [la figure 10-3](#) qui montre comment vous voulez que les choses soient disposées en mémoire. Ensuite, revenez de l'image au code. Chaque collection de rectan-

gles est une structure ou un tuple; chaque flèche est un ou un autre pointeur intelligent. Déterminer le type de chaque champ est un peu un casse-tête, mais gérable. La récompense pour résoudre le puzzle est le contrôle de l'utilisation de la mémoire de votre programme. `BinaryTree Box Box`

Vient maintenant le « prix » que nous avons mentionné dans l'introduction. Le champ de balise d'un enum coûte un peu de mémoire, jusqu'à huit octets dans le pire des cas, mais c'est généralement négligeable. Le véritable inconvénient des enums (si on peut l'appeler ainsi) est que le code Rust ne peut pas jeter la prudence au vent et essayer d'accéder aux champs, qu'ils soient ou non réellement présents dans la valeur:

```
let r = shape.radius; // error: no field `radius` on type `Shape`
```

La seule façon d'accéder aux données dans un enum est la manière sûre: utiliser des modèles.

Modèles

Rappelez-vous la définition de notre type de plus haut dans ce chapitre: `RoughTime`

```
enum RoughTime {
    InThePast(TimeUnit, u32),
    JustNow,
    InTheFuture(TimeUnit, u32),
}
```

Supposons que vous ayez une valeur et que vous souhaitez l'afficher sur une page Web. Vous devez accéder aux champs et à l'intérieur de la valeur. Rust ne vous permet pas d'y accéder directement, en écrivant et , car après tout, la valeur pourrait être , qui n'a pas de champs. Mais alors, comment pouvez-vous extraire les données?

```
RoughTime TimeUnit u32 rough_time.0 rough_time.1 RoughTime:
:JustNow
```

Vous avez besoin d'une expression : `match`

```
1 fn rough_time_to_english(rt: RoughTime) -> String {
2     match rt {
3         RoughTime::InThePast(units, count) =>
4             format!("{} {} ago", count, units.plural()),
5         RoughTime::JustNow =>
```

```

6         format!("just now"),
7     RoughTime::InTheFuture(units, count) =>
8         format!("{} {} from now", count, units.plural()),
9     }
10 }

```

match effectue l'appariement des modèles; dans cet exemple, les *motifs* sont les pièces qui apparaissent avant le symbole sur les lignes 3, 5 et 7. Les modèles qui correspondent aux valeurs ressemblent aux expressions utilisées pour créer des valeurs. Ce n'est pas une coïncidence. Les expressions *produisent des* valeurs; les modèles *consomment* des valeurs. Les deux utilisent beaucoup de la même syntaxe. => RoughTime RoughTime

Passons en revue ce qui se passe lorsque cette expression s'exécute. Supposons que la valeur . Rust essaie d'abord de faire correspondre cette valeur au modèle de la ligne 3. Comme vous pouvez le voir à [la figure 10-4](#), il ne correspond

```
pas.match rt RoughTime::InTheFuture(TimeUnit::Months, 1)
```

```

value: RoughTime::InTheFuture(TimeUnit::Months, 1)
      ↓
      X
pattern: RoughTime::InThePast(units, count)

```

Graphique 10-4. Une valeur et un modèle qui ne correspondent pas RoughTime

Le motif correspondant à un enum, une structure ou un tuple fonctionne comme si Rust effectuait une simple analyse de gauche à droite, en vérifiant chaque composant du motif pour voir si la valeur lui correspond. Si ce n'est pas le cas, Rust passe au modèle suivant.

Les motifs des lignes 3 et 5 ne correspondent pas. Mais le modèle de la ligne 7 réussit ([Figure 10-5](#)).

```

value: RoughTime::InTheFuture(TimeUnit::Months, 1)
      ↓
      ✓
      ↓
      ✓
pattern: RoughTime::InTheFuture(
                           units, count)

```

Figure 10-5. Un match réussi

Lorsqu'un modèle contient des identificateurs simples tels que , ceux-ci deviennent des variables locales dans le code suivant le modèle. Tout ce qui est présent dans la valeur est copié ou déplacé dans les nouvelles variables. Rust stocke dans et dans , exécute la ligne 8 et renvoie la chaîne

```
.units count TimeUnit::Months units 1 count "1 months from  
now"
```

Cette sortie a un problème grammatical mineur, qui peut être résolu en ajoutant un autre bras au :match

```
RoughTime::InTheFuture(unit, 1) =>  
    format!("a {} from now", unit.singular()),
```

Ce bras ne correspond que si le champ est exactement 1. Notez que ce nouveau code doit être ajouté avant la ligne 7. Si nous l'ajoutons à la fin, Rust n'y arrivera jamais, car le motif de la ligne 7 correspond à toutes les valeurs. Le compilateur Rust vous avertira d'un « modèle inaccessible » si vous faites ce genre d'erreur. count InTheFuture

Même avec le nouveau code, présente toujours un problème: le résultat n'est pas tout à fait correct. Telle est la langue anglaise. Cela aussi peut être corrigé en ajoutant un autre bras au

```
. RoughTime::InTheFuture(TimeUnit::Hours, 1) "a hour from  
now" match
```

Comme le montre cet exemple, la correspondance de motifs fonctionne main dans la main avec les enums et peut même tester les données qu'ils contiennent, ce qui constitue un remplacement puissant et flexible de l'instruction C. Jusqu'à présent, nous n'avons vu que des modèles qui correspondent aux valeurs enum. Il y a plus que cela. Les modèles de rouille sont leur propre petit langage, résumé dans [le tableau 10-1](#). Nous passerons la majeure partie du reste du chapitre sur les fonctionnalités présentées dans ce tableau. match switch

Tableau 10-1. Modèles

Type de motif	Exemple	Notes
Littéral	100 "name"	Correspond à une valeur exacte; le nom d'un est également autorisé const
Gamme	0 ..= 100 'a' ..= 'k' 256 ..	Correspond à n'importe quelle valeur de plage, y compris la valeur finale si elle est donnée
Génériques	_	Correspond à n'importe quelle valeur et l'ignore
Variable	name mut count	J'aime mais déplace ou copie la valeur dans une nouvelle variable locale _
ref variable	ref field ref mut fie ld	Emprunte une référence à la valeur correspondante au lieu de la déplacer ou de la copier
Liaison avec sous-modèle	val @ 0 ..= 99 ref circle @ Shape::Circle e { .. }	Correspond au motif à droite de , en utilisant le nom de la variable à gauche @
Motif	Some(value)	
Enum	None Pet::Orca	
Modèle de tuple	(key, value) (r, g, b)	
Modèle de tableau	[a, b, c, d, e, f, g] [heading, column, correct ion]	

Type de motif	Exemple	Notes
Motif de tranche	[first, second] [first, _, third] [first, ..., nth] []	
Modèle de structure	Color(r, g, b) Point { x, y } Card { suit: Clubs, rank: n } Account { id, name, .. }	
Référence	&value &(k, v)	Correspond uniquement aux valeurs de référence
Ou des modèles	'a' 'A' Some("left" "right")	
Expression de garde	x if x * x <= r2	En seulement (non valide en , etc.) match let

Littéraux, variables et caractères génériques dans les modèles

Jusqu'à présent, nous avons montré des expressions travaillant avec des enums. D'autres types peuvent également être appariés. Lorsque vous avez besoin d'une instruction C, utilisez avec une valeur entière. Les littéraux entiers aiment et peuvent servir de motifs:

```
match switch match 0 1
```

```
match meadow.count_rabbits() {
  0 => {} // nothing to say
  1 => println!("A rabbit is nosing around in the clover."),
}
```

```
n => println!( "There are {} rabbits hopping about in the meadow", n)
}
```

Le motif correspond s'il n'y a pas de lapins dans le pré. correspond s'il n'y en a qu'un. S'il y a deux lapins ou plus, nous atteignons le troisième modèle, . Ce modèle n'est qu'un nom de variable. Elle peut correspondre à n'importe quelle valeur et la valeur correspondante est déplacée ou copiée dans une nouvelle variable locale. Donc, dans ce cas, la valeur de est stockée dans une nouvelle variable locale , que nous imprimons ensuite. 0 1 n meadow.count_rabbits() n

D'autres littéraux peuvent également être utilisés comme motifs, y compris les booléens, les caractères et même les chaînes:

```
let calendar = match settings.get_string("calendar") {
    "gregorian" => Calendar::Gregorian,
    "chinese" => Calendar::Chinese,
    "ethiopian" => Calendar::Ethiopian,
    other => return parse_error("calendar", other),
};
```

Dans cet exemple, sert de modèle fourre-tout comme dans l'exemple précédent. Ces modèles jouent le même rôle qu'un cas dans une instruction, correspondant à des valeurs qui ne correspondent à aucun des autres modèles. other n default switch

Si vous avez besoin d'un modèle fourre-tout, mais que vous ne vous souciez pas de la valeur correspondante, vous pouvez utiliser un seul trait de soulignement comme motif, le *modèle générique* : _

```
let caption = match photo.tagged_pet() {
    Pet::Tyrannosaur => "RRRAAAAHHHHHH",
    Pet::Samoyed => "*dog thoughts*",
    _ => "I'm cute, love me", // generic caption, works for any pet
};
```

Le modèle générique correspond à n'importe quelle valeur, mais sans le stocker n'importe où. Étant donné que Rust exige que chaque expression gère toutes les valeurs possibles, un caractère générique est souvent requis à la fin. Même si vous êtes très sûr que les cas restants ne peuvent pas se produire, vous devez au moins ajouter un bras de secours, peut-être un bras qui panique: match

```

// There are many Shapes, but we only support "selecting"
// either some text, or everything in a rectangular area.
// You can't select an ellipse or trapezoid.

match document.selection() {
    Shape::TextSpan(start, end) => paint_text_selection(start, end),
    Shape::Rectangle(rect) => paint_rect_selection(rect),
    _ => panic!("unexpected selection type"),
}

```

Modèles Tuple et Struct

Les motifs de tuple correspondent aux tuples. Ils sont utiles chaque fois que vous souhaitez impliquer plusieurs données dans un seul :match

```

fn describe_point(x: i32, y: i32) -> &'static str {
    use std::cmp::Ordering::*;

    match (x.cmp(&0), y.cmp(&0)) {
        (Equal, Equal) => "at the origin",
        (_, Equal) => "on the x axis",
        (Equal, _) => "on the y axis",
        (Greater, Greater) => "in the first quadrant",
        (Less, Greater) => "in the second quadrant",
        _ => "somewhere else",
    }
}

```

Les motifs Struct utilisent des accolades bouclées, tout comme les expressions struct. Ils contiennent un sous-modèle pour chaque champ :

```

match balloon.location {
    Point { x: 0, y: height } =>
        println!("straight up {} meters", height),
    Point { x: x, y: y } =>
        println!("at ({}, {})", x, y),
}

```

Dans cet exemple, si le premier bras correspond, alors est stocké dans la nouvelle variable locale .balloon.location.y height

Supposons que est . Comme toujours, Rust vérifie chaque composant de chaque motif à tour [de rôle Figure 10-6](#). balloon.location Point { x: 30, y: 40 }

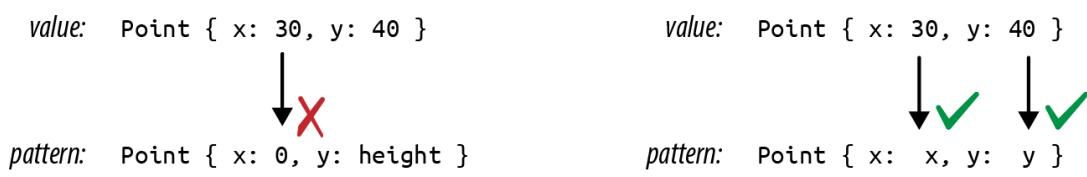


Figure 10-6. Correspondance de motifs avec des structures

Le deuxième bras correspond, de sorte que la sortie serait `.at (30m, 40m)`

Les modèles comme sont courants lors de la correspondance des structs, et les noms redondants sont un encombrement visuel, donc Rust a un raccourci pour cela: `.`. Le sens est le même. Ce modèle stocke toujours le champ d'un point dans un nouveau local et son champ dans un nouveau local. `Point { x: x, y: y }` `Point {x, y} x x y y`

Même avec le raccourci, il est fastidieux de faire correspondre une grande structure lorsque nous ne nous soucions que de quelques champs:

```
match get_account(id) {
    ...
    Some(Account {
        name, language, // <--- the 2 things we care about
        id: _, status: _, address: _, birthday: _, eye_color: _,
        pet: _, security_question: _, hashed_innermost_secret: _,
        is_adamantium_preferred_customer: _, }) =>
        language.show_custom_greeting(name),
}
```

Pour éviter cela, dites à Rust que vous ne vous souciez daucun des autres champs: `..`

```
Some(Account { name, language, .. }) =>
    language.show_custom_greeting(name),
```

Modèles de tableau et de tranche

Les modèles de tableau correspondent aux tableaux. Ils sont souvent utilisés pour filtrer certaines valeurs de cas spéciaux et sont utiles chaque fois que vous travaillez avec des tableaux dont les valeurs ont une signification différente en fonction de la position.

Par exemple, lors de la conversion des valeurs de couleur de teinte, de saturation et de luminosité (HSL) en valeurs de couleur rouge, vert, bleu (RVB), les couleurs sans luminosité ou avec une luminosité totale sont

simplement noires ou blanches. Nous pourrions utiliser une expression pour traiter ces cas simplement. `match`

```
fn hsl_to_rgb(hsl: [u8; 3]) -> [u8; 3] {
    match hsl {
        [_, _, 0] => [0, 0, 0],
        [_, _, 255] => [255, 255, 255],
        ...
    }
}
```

Les modèles de tranches sont similaires, mais contrairement aux tableaux, les tranches ont des longueurs variables, de sorte que les tapotements de tranche correspondent non seulement aux valeurs, mais également à la longueur. dans un motif de tranche correspond à un nombre quelconque d'éléments : ..

```
fn greet_people(names: &[&str]) {
    match names {
        [] => { println!("Hello, nobody.") },
        [a] => { println!("Hello, {}.", a) },
        [a, b] => { println!("Hello, {} and {}.", a, b) },
        [a, ..., b] => { println!("Hello, everyone from {} to {}.", a, b) }
    }
}
```

Modèles de référence

Les motifs de rouille prennent en charge deux fonctionnalités pour travailler avec des références. les motifs empruntent des parties d'une valeur correspondante. les modèles correspondent aux références. Nous allons d'abord couvrir les modèles. `ref` & `ref mut`

La correspondance d'une valeur non copiable déplace la valeur. En continuant avec l'exemple de compte, ce code ne serait pas valide :

```
match account {
    Account { name, language, .. } => {
        ui.greet(&name, &language);
        ui.show_settings(&account); // error: borrow of moved value: `a
    }
}
```

Ici, les champs et sont déplacés dans les variables locales et . Le reste est abandonné. C'est pourquoi nous ne pouvons pas emprunter une référence à cela par la suite.

```
account.name account.language name language account
```

Si et étaient les deux valeurs copiables, Rust copierait les champs au lieu de les déplacer, et ce code serait correct. Mais supposons que ce soient des s. Que pouvons-nous faire?

```
name language String
```

Nous avons besoin d'une sorte de modèle qui *emprunte des valeurs correspondantes* au lieu de les déplacer. C'est exactement ce que fait le mot-clé : ref

```
match account {
    Account { ref name, ref language, .. } => {
        ui.greet(name, language);
        ui.show_settings(&account); // ok
    }
}
```

Maintenant, les variables locales et sont des références aux champs correspondants dans . Étant donné qu'il n'est qu'emprunté, pas consommé, il est acceptable de continuer à appeler des méthodes dessus.

```
name language account account
```

Vous pouvez utiliser pour emprunter des références : ref mut mut

```
match line_result {
    Err(ref err) => log_error(err), // `err` is &Error (shared ref)
    Ok(ref mut line) => {          // `line` is &mut String (mut ref)
        trim_comments(line);        // modify the String in place
        handle(line);
    }
}
```

Le modèle correspond à n'importe quel résultat de réussite et emprunte une référence à la valeur de succès stockée à l'intérieur.

```
Ok(ref mut line) mut
```

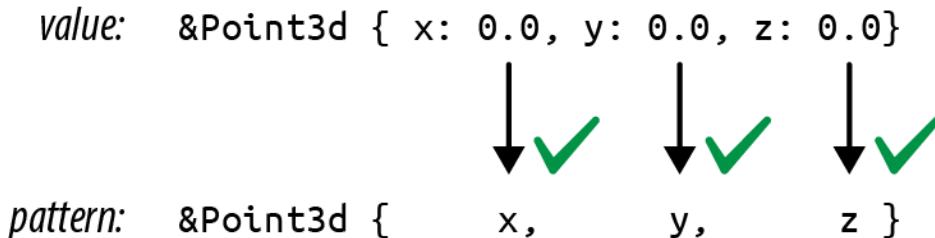
Le type opposé de modèle de référence est le modèle. Un modèle commençant par correspond à une référence : & &

```
match sphere.center() {
    &Point3d { x, y, z } => ...
}
```

Dans cet exemple, supposons qu'il renvoie une référence à un champ privé de , un motif commun dans Rust. La valeur renvoyée est l'adresse d'un fichier . Si le centre est à l'origine, renvoie

```
.sphere.center() sphere Point3d sphere.center() &Point3d {  
    x: 0.0, y: 0.0, z: 0.0 }
```

La correspondance des motifs se déroule comme illustré à [la figure 10-7](#).



Graphique 10-7. Correspondance de motifs avec des références

C'est un peu délicat car Rust suit un pointeur ici, une action que nous associons généralement à l'opérateur, pas à l'opérateur. La chose à retenir est que les motifs et les expressions sont des opposés naturels. L'expression transforme deux valeurs en un nouveau tuple, mais le motif fait le contraire : il correspond à un tuple et décompose les deux valeurs. C'est la même chose avec . Dans une expression, crée une référence. Dans un modèle, correspond à une référence. * & (x, y) (x, y) & & &

Faire correspondre une référence suit toutes les règles auxquelles nous nous attendons. Les durées de vie sont appliquées. Vous ne pouvez pas y accéder via une référence partagée. Et vous ne pouvez pas déplacer une valeur hors d'une référence, même d'une référence. Lorsque nous faisons correspondre , les variables , et recevons des copies des coordonnées, en laissant la valeur d'origine intacte. Cela fonctionne parce que ces champs sont copiables. Si nous essayons la même chose sur une structure avec des champs non copiables, nous obtiendrons une erreur

```
:mut mut &Point3d { x, y, z } x y z Point3d
```

```
match friend.borrow_car() {  
    Some(&Car { engine, .. }) => // error: can't move out of borrow  
        ...  
    None => {}  
}
```

Mettre au rebut une voiture empruntée pour des pièces n'est pas agréable, et Rust ne le supportera pas. Vous pouvez utiliser un motif pour emprunter une référence à un article. Vous ne le possédez tout simplement pas: ref

```
Some(&Car { ref engine, .. }) => // ok, engine is a reference
```

Regardons un autre exemple de modèle. Supposons que nous ayons un itérateur sur les caractères d'une chaîne, et qu'il ait une méthode qui renvoie une référence au caractère suivant, le cas échéant. (Les itérateurs peekables renvoient en fait un , comme nous le verrons au [chapitre 15.](#)) & chars chars.peek() Option<&char> Option<&ItemType>

Un programme peut utiliser un modèle pour obtenir le caractère pointu : &

```
match chars.peek() {
    Some(&c) => println!("coming up: {:?}", c),
    None => println!("end of chars"),
}
```

Gardes de match

Parfois, un bras d'allumette a des conditions supplémentaires qui doivent être remplies avant de pouvoir être considéré comme un match. Supposons que nous mettions en œuvre un jeu de société avec des espaces hexagonaux et que le joueur clique simplement pour déplacer une pièce. Pour confirmer que le clic était valide, nous pouvons essayer quelque chose comme ceci:

```
fn check_move(current_hex: Hex, click: Point) -> game::Result<Hex> {
    match point_to_hex(click) {
        None =>
            Err("That's not a game space."),
        Some(current_hex) => // try to match if user clicked the current
                             // (it doesn't work: see explanation below)
            Err("You are already there! You must click somewhere else.")
        Some(other_hex) =>
            Ok(other_hex)
    }
}
```

Cela échoue car les identificateurs dans les modèles introduisent de *nouvelles* variables. Le modèle ici crée une nouvelle variable locale , ombrageant l'argument . Rust émet plusieurs avertissements à propos de ce code, en particulier, le dernier bras du est inaccessible. Une façon de résoudre ce problème consiste simplement à utiliser une expression dans le bras

```
d'allumette: Some(current_hex) current_hex current_hex match i
f

    match point_to_hex(click) {
        None => Err("That's not a game space."),
        Some(hex) => {
            if hex == current_hex {
                Err("You are already there! You must click somewhere else")
            } else {
                Ok(hex)
            }
        }
    }
}
```

Mais Rust fournit également des *gardes d'allumettes*, des conditions supplémentaires qui doivent être vraies pour qu'un bras d'allumette s'applique, écrit comme , entre le motif et le jeton du bras: `if CONDITION =>`

```
match point_to_hex(click) {
    None => Err("That's not a game space."),
    Some(hex) if hex == current_hex =>
        Err("You are already there! You must click somewhere else"),
    Some(hex) => Ok(hex)
}
```

Si le modèle correspond, mais que la condition est fausse, la correspondance se poursuit avec le bras suivant.

Faire correspondre plusieurs possibilités

Un modèle du formulaire correspond si l'un ou l'autre des sous-modèles correspond à : `pat1 | pat2`

```
let at_end = match chars.peek() {
    Some(&'\r' | &'\n') | None => true,
    _ => false,
};
```

Dans une expression, est l'opérateur OR binaire, mais ici il fonctionne plus comme le symbole dans une expression régulière. est défini sur si est , ou un maintien d'un retour chariot ou d'un saut de ligne. | | at_end true chars.peek() None Some

Permet de faire correspondre toute une plage de valeurs. Les modèles de plage incluent les valeurs de début et de fin, ce qui correspond à tous les

```
chiffres ASCII : .. = '0' .. = '9'
```

```
match next_char {
    '0'..='9' => self.read_number(),
    'a'..='z' | 'A'..='Z' => self.read_word(),
    ' ' | '\t' | '\n' => self.skip_whitespace(),
    _ => self.handle_punctuation(),
}
```

Rust autorise également des modèles de plage tels que , qui correspondent à n'importe quelle valeur allant jusqu'à la valeur maximale du type. Cependant, les autres variétés de gammes exclusives à la fin, comme ou , et les gammes illimitées comme ne sont pas encore autorisées dans les modèles. x... x 0..100 ..100 ..

Liaison avec @ Patterns

Enfin, correspond exactement comme le donné , mais en cas de succès, au lieu de créer des variables pour des parties de la valeur correspondante, il crée une seule variable et déplace ou copie la valeur entière dans celle-ci. Par exemple, supposons que vous ayez ce code : x @ pattern pattern x

```
match self.get_selection() {
    Shape::Rect(top_left, bottom_right) => {
        optimized_paint(&Shape::Rect(top_left, bottom_right))
    }
    other_shape => {
        paint_outline(other_shape.get_outline())
    }
}
```

Notez que le premier cas décomprime une valeur, uniquement pour reconstruire une valeur identique sur la ligne suivante. Cela peut être réécrit pour utiliser un modèle : Shape::Rect Shape::Rect @

```
rect @ Shape::Rect(..) => {
    optimized_paint(&rect)
}
```

@ les motifs sont également utiles avec les plages:

```
match chars.next() {
    Some(digit @ '0'..='9') => read_number(digit, chars),
```

```
...  
},
```

Où les modèles sont autorisés

Bien que les motifs soient les plus importants dans les expressions, ils sont également autorisés à plusieurs autres endroits, généralement à la place d'un identifiant. La signification est toujours la même : au lieu de simplement stocker une valeur dans une seule variable, Rust utilise la correspondance de motif pour séparer la valeur. `match`

Cela signifie que les modèles peuvent être utilisés pour...

```
// ...unpack a struct into three new local variables  
let Track { album, track_number, title, .. } = song;  
  
// ...unpack a function argument that's a tuple  
fn distance_to((x, y): (f64, f64)) -> f64 { ... }  
  
// ...iterate over keys and values of a HashMap  
for (id, document) in &cache_map {  
    println!("Document #{:}: {}", id, document.title);  
}  
  
// ...automatically dereference an argument to a closure  
// (handy because sometimes other code passes you a reference  
// when you'd rather have a copy)  
let sum = numbers.fold(0, |a, &num| a + num);
```

Chacun d'entre eux permet d'économiser deux ou trois lignes de code standard. Le même concept existe dans d'autres langages : en JavaScript, on l'appelle *déstructuration*, tandis qu'en Python, c'est le *déballage*.

Notez que dans les quatre exemples, nous utilisons des modèles qui sont garantis pour correspondre. Le motif correspond à toutes les valeurs possibles du type struct, correspond à n'importe quelle paire, etc. Les motifs qui correspondent toujours sont spéciaux dans Rust. Ils sont *appelés modèles irréfutables*, et ce sont les seuls modèles autorisés aux quatre endroits montrés ici (après , dans les arguments de fonction, après et dans les arguments de fermeture). `Point3d { x, y, z } Point3d (x, y) (f64, f64)`

Un *modèle réfutable* est un modèle qui peut ne pas correspondre, comme , qui ne correspond pas à un résultat d'erreur, ou , qui ne correspond pas au caractère . Les motifs réfutables peuvent être utilisés dans les bras, car

ils sont conçus pour eux: si un motif ne correspond pas, il est clair ce qui se passe ensuite. Les quatre exemples précédents sont des endroits dans les programmes Rust où un modèle peut être pratique, mais le langage ne permet pas l'échec de la correspondance.

```
Ok(x) '0' ..= '9' 'Q' match match
```

Les motifs réfutables sont également autorisés dans et les expressions, qui peuvent être utilisées pour... if let while let

```
// ...handle just one enum variant specially
if let RoughTime::InTheFuture(_, _) = user.date_of_birth() {
    user.set_time_traveler(true);
}

// ...run some code only if a table lookup succeeds
if let Some(document) = cache_map.get(&id) {
    return send_cached_response(document);
}

// ...repeatedly try something until it succeeds
while let Err(err) = present_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    // let the user try again (it might still be a human)
}

// ...manually loop over an iterator
while let Some(_) = lines.peek() {
    read_paragraph(&mut lines);
}
```

Pour plus d'informations sur ces expressions, voir [« if let »](#) et [« Loops »](#).

Remplissage d'un arbre binaire

Plus tôt, nous avons promis de montrer comment implémenter une méthode, , qui ajoute un nœud à un de ce type:

```
BinaryTree::add() BinaryTree
```

```
// An ordered collection of `T`s.
enum BinaryTree<T> {
    Empty,
    NonEmpty(Box<TreeNode<T>>),
}

// A part of a BinaryTree.
struct TreeNode<T> {
```

```

    element: T,
    left: BinaryTree<T>,
    right: BinaryTree<T>,
}

```

Vous en savez maintenant assez sur les modèles pour écrire cette méthode. Une explication des arbres de recherche binaires dépasse le cadre de ce livre, mais pour les lecteurs déjà familiers avec le sujet, il vaut la peine de voir comment cela se passe dans Rust.

```

1  impl<T: Ord> BinaryTree<T> {
2      fn add(&mut self, value: T) {
3          match *self {
4              BinaryTree::Empty => {
5                  *self = BinaryTree::NonEmpty(Box::new(TreeNode {
6                      element: value,
7                      left: BinaryTree::Empty,
8                      right: BinaryTree::Empty,
9                      }))
10             }
11             BinaryTree::NonEmpty(ref mut node) => {
12                 if value <= node.element {
13                     node.left.add(value);
14                 } else {
15                     node.right.add(value);
16                 }
17             }
18         }
19     }
20 }

```

La ligne 1 indique à Rust que nous définissons une méthode sur des types ordonnés. C'est exactement la même syntaxe que nous utilisons pour définir des méthodes sur des structures génériques, [expliquée dans « Définition de méthodes avec `impl` »](#). `BinaryTree`

Si l'arbre existant est vide, c'est le cas facile. Les lignes 5 à 9 s'exécutent, changeant l'arbre en un. L'appel à ici alloue un nouveau dans le tas.

Lorsque nous avons terminé, l'arbre contient un élément. Ses sous-arbres gauche et droit sont tous deux

```
. *self Empty NonEmpty Box::new() TreeNode Empty
```

Si n'est pas vide, nous faisons correspondre le modèle de la ligne 11: `*self`

```
BinaryTree::NonEmpty(ref mut node) => {
```

Ce modèle emprunte une référence modifiable au , afin que nous puissions accéder aux données de ce nœud d’arborescence et les modifier. Cette référence est nommée , et elle est dans la portée de la ligne 12 à la ligne 16. Comme il y a déjà un élément dans ce nœud, le code doit appeler de manière récursive pour ajouter le nouvel élément à la sous-arborescence gauche ou droite. Box<TreeNode<T>> node .add()

La nouvelle méthode peut être utilisée comme ceci:

```
let mut tree = BinaryTree::Empty;
tree.add("Mercury");
tree.add("Venus");
...
...
```

Vue d’ensemble

Les enums de Rust sont peut-être nouveaux dans la programmation de systèmes, mais ils ne sont pas une idée nouvelle. Voyageant sous divers noms à consonance académique, comme les *types de données algébriques*, ils sont utilisés dans les langages de programmation fonctionnels depuis plus de quarante ans. On ne sait pas pourquoi si peu d’autres langues de la tradition C en ont jamais eu. Peut-être est-ce simplement que pour un concepteur de langage de programmation, combiner des variantes, des références, la mutabilité et la sécurité de la mémoire est extrêmement difficile. Les langages de programmation fonctionnels se passent de mutabilité. Les C, en revanche, ont des variantes, des pointeurs et une mutabilité, mais sont si spectaculairement dangereux que même en C, ils sont un dernier recours. Le vérificateur d’emprunt de Rust est la magie qui permet de combiner les quatre sans compromis. union

La programmation, c’est le traitement des données. Obtenir des données dans la bonne forme peut faire la différence entre un petit programme rapide et élégant et un enchevêtement lent et gigantesque de ruban adhésif et d’appels de méthode virtuelle.

C’est le problème que les enums d’espace abordent. Ils sont un outil de conception pour obtenir des données dans la bonne forme. Pour les cas où une valeur peut être une chose, ou une autre chose, ou peut-être rien du tout, les enums sont meilleurs que les hiérarchies de classes sur

chaque axe : plus rapides, plus sûrs, moins de code, plus faciles à documenter.

Le facteur limitant est la flexibilité. Les utilisateurs finaux d'un enum ne peuvent pas l'étendre pour ajouter de nouvelles variantes. Les variantes ne peuvent être ajoutées qu'en modifiant la déclaration enum. Et lorsque cela se produit, le code existant se brise. Chaque expression qui correspond individuellement à chaque variante de l'enum doit être revisée – elle a besoin d'un nouveau bras pour gérer la nouvelle variante. Dans certains cas, la flexibilité de trading pour la simplicité est juste du bon sens. Après tout, la structure de JSON ne devrait pas changer. Et dans certains cas, revisiter toutes les utilisations d'un enum quand il change est exactement ce que nous voulons. Par exemple, lorsque an est utilisé dans un compilateur pour représenter les différents opérateurs d'un langage de programmation, l'ajout d'un nouvel opérateur *doit* impliquer de toucher tout le code qui gère les opérateurs. `match enum`

Mais parfois, plus de flexibilité est nécessaire. Pour ces situations, Rust a des traits, le sujet de notre prochain chapitre.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 11. Traits et génériques

[Un] informaticien a tendance à être capable de traiter des structures non uniformes – cas 1, cas 2, cas 3 – tandis qu'un mathématicien aura tendance à vouloir un axiome unificateur qui régit un système entier.

—Donald Knuth

L'une des grandes découvertes de la programmation est qu'il est possible d'écrire du code qui fonctionne sur des valeurs de nombreux types différents, *même des types qui n'ont pas encore été inventés*. Voici deux exemples :

- `Vec<T>` est générique : vous pouvez créer un vecteur de n'importe quel type de valeur, y compris des types définis dans votre programme que les auteurs n'ont jamais anticipés. `Vec`
- Beaucoup de choses ont des méthodes, y compris `s` et `s`. Votre code peut prendre un scripteur par référence, n'importe quel scripteur, et lui envoyer des données. Votre code n'a pas à se soucier du type d'écriture dont il s'agit. Plus tard, si quelqu'un ajoute un nouveau type de graveur, votre code le prendra déjà en charge. `.write() File TcpStream`

Bien sûr, cette capacité n'est pas nouvelle avec Rust. C'est ce qu'on appelle le *polymorphisme*, et c'était la nouvelle technologie de langage de programmation des années 1970. À l'heure actuelle, il est effectivement universel. Rust supporte le polymorphisme avec deux caractéristiques connexes: les traits et les génériques. Ces concepts seront familiers à de nombreux programmeurs, mais Rust adopte une nouvelle approche inspirée des classes de type de Haskell.

Les traits sont la vision de Rust sur les interfaces ou les classes de base abstraites. Au début, ils ressemblent à des interfaces en Java ou C#. Le trait d'écriture d'octets est appelé `,` et sa définition dans la bibliothèque standard commence comme ceci: `std::io::Write`

```
trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()> { ... }
    ...
}
```

Ce trait offre plusieurs méthodes; nous n'avons montré que les trois premiers.

Les types standard et les deux implémentent . Il en va de même pour . Les trois types fournissent des méthodes nommées , etc. Le code qui utilise un scripteur sans se soucier de son type ressemble à ceci

```
:File TcpStream std::io::Write Vec<u8> .write() .flush()

use std::io::Write;

fn say_hello(out: &mut dyn Write) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Le type de est , ce qui signifie « une référence mutable à toute valeur qui implémente le trait ». Nous pouvons passer une référence mutable à une telle valeur: out &mut dyn Write Write say_hello

```
use std::fs::File;
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file)?; // works

let mut bytes = vec![];
say_hello(&mut bytes)?; // also works
assert_eq!(bytes, b"hello world\n");
```

Ce chapitre commence par montrer comment les traits sont utilisés, comment ils fonctionnent et comment définir les vôtres. Mais il y a plus de traits que ce que nous avons laissé entendre jusqu'à présent. Nous les utiliserons pour ajouter des méthodes d'extension aux types existants, même des types intégrés comme et . Nous expliquerons pourquoi l'ajout d'un trait à un type ne coûte pas de mémoire supplémentaire et comment utiliser les traits sans surcharge d'appel de méthode virtuelle. Nous verrons que les traits intégrés sont le crochet dans le langage que Rust fournit pour la surcharge de l'opérateur et d'autres fonctionnalités. Et nous couvrirons le type, les fonctions associées et les types associés, trois fonctionnalités que Rust a retirées de Haskell et qui résolvent élégamment les problèmes que d'autres langages abordent avec des solutions de contournement et des hacks. str bool Self

Les génériques sont l'autre saveur du polymorphisme dans Rust. Comme un modèle C++, une fonction ou un type générique peut être utilisé avec des valeurs de nombreux types différents :

```

/// Given two values, pick whichever one is less.

fn min<T: Ord>(value1: T, value2: T) -> T {
    if value1 <= value2 {
        value1
    } else {
        value2
    }
}

```

La fonction dans cette fonction signifie qu'elle peut être utilisée avec des arguments de n'importe quel type qui implémente le trait, c'est-à-dire n'importe quel type ordonné. Une exigence comme celle-ci est appelée une *limite*, car elle fixe des limites sur les types qui pourraient éventuellement l'être. Le compilateur génère du code machine personnalisé pour chaque type que vous utilisez réellement. `<T: Ord> min T Ord T T`

Les génériques et les traits sont étroitement liés : les fonctions génériques utilisent des traits dans les limites pour énoncer les types d'arguments auxquels elles peuvent être appliquées. Nous parlerons donc également de la façon dont et sont similaires, en quoi ils sont différents et comment choisir entre ces deux façons d'utiliser les traits. `&mut dyn Write <T: Write>`

Utilisation des traits

Un trait est une caractéristique qu'un type donné peut ou non prendre en charge. Le plus souvent, un trait représente une capacité : quelque chose qu'un type peut faire.

- Une valeur qui implémente peut écrire des octets. `std::io::Write`
- Une valeur qui implémente peut produire une séquence de valeurs. `std::iter::Iterator`
- Une valeur qui implémente peut faire des clones d'elle-même en mémoire. `std::clone::Clone`
- Une valeur qui implémente peut être imprimée à l'aide du spécificateur de format. `std::fmt::Debug println!() { :? }`

Ces quatre traits font tous partie de la bibliothèque standard de Rust, et de nombreux types standard les implémentent. Par exemple:

- `std::fs::File` met en œuvre le trait; il écrit des octets dans un fichier local. Il écrit dans une connexion réseau. Il implémente également .

Chaque appel sur un vecteur d'octets ajoute des données à la

```
fin. Write std::net::TcpStream Vec<u8> Write .write()
```

- Range<i32> (le type de) implémente le trait, tout comme certains types d'itérateurs associés aux tranches, aux tables de hachage, etc. 0 .. 10 Iterator
- La plupart des types de bibliothèque standard implémentent . Les exceptions sont principalement des types comme celui-ci qui représentent plus que de simples données en mémoire. clone TcpStream
- De même, la plupart des types de bibliothèque standard prennent en charge . Debug

Il existe une règle inhabituelle à propos des méthodes de trait: le trait lui-même doit être dans la portée. Sinon, toutes ses méthodes sont cachées:

```
let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // error: no method named `write_all`
```

Dans ce cas, le compilateur imprime un message d'erreur convivial qui suggère d'ajouter et en effet qui résout le problème: use std::io::Write;

```
use std::io::Write;

let mut buf: Vec<u8> = vec![];
buf.write_all(b"hello")?; // ok
```

Rust a cette règle car, comme nous le verrons plus loin dans ce chapitre, vous pouvez utiliser des traits pour ajouter de nouvelles méthodes à n'importe quel type, même les types de bibliothèque standard comme et . Les caisses tierces peuvent faire la même chose. De toute évidence, cela pourrait conduire à des conflits de noms! Mais puisque Rust vous fait importer les traits que vous prévoyez d'utiliser, les caisses sont libres de profiter de ce superpouvoir. Pour obtenir un conflit, vous devez importer deux traits qui ajoutent une méthode portant le même nom au même type. C'est rare dans la pratique. (Si vous rencontrez un conflit, vous pouvez énoncer ce que vous voulez à l'aide [d'une syntaxe de méthode complète](#), abordée plus loin dans le chapitre.) u32 str

La raison et les méthodes fonctionnent sans aucune importation spéciale est qu'elles sont toujours dans le champ d'application par défaut : elles font partie du prélude standard, des noms que Rust importe automatiquement dans chaque module. En fait, le prélude est principalement une

sélection soigneusement choisie de traits. Nous couvrirons beaucoup d'entre eux dans [le chapitre 13](#). `Clone Iterator`

Les programmeurs C++ et C# auront déjà remarqué que les méthodes de trait sont comme des méthodes virtuelles. Pourtant, les appels comme celui montré ci-dessus sont rapides, aussi rapides que n'importe quel autre appel de méthode. En termes simples, il n'y a pas de polymorphisme ici. Il est évident qu'il s'agit d'un vecteur, pas d'un fichier ou d'une connexion réseau. Le compilateur peut émettre un simple appel à . Il peut même intégrer la méthode. (C++ et C# feront souvent de même, bien que la possibilité de sous-classification l'empêche parfois.) Seuls les appels via encourent la surcharge d'une répartition dynamique, également appelée appel de méthode virtuelle, qui est indiquée par le mot-clé dans le type. est connu comme un *objet trait*; nous examinerons les détails techniques des objets traits et leur comparaison avec les fonctions génériques dans les sections suivantes.

```
buf Vec<u8>::write() &mut  
dyn Write dyn dyn Write
```

Objets Trait

Il existe deux façons d'utiliser des traits pour écrire du code polymorphe dans Rust : les objets de trait et les génériques. Nous présenterons d'abord les objets de trait et nous nous tournerons vers les génériques dans la section suivante.

La rouille n'autorise pas les variables de type : `dyn Write`

```
use std::io::Write;  
  
let mut buf: Vec<u8> = vec![];  
let writer: dyn Write = buf; // error: `Write` does not have a constant
```

La taille d'une variable doit être connue au moment de la compilation, et les types qui implémentent peuvent être de n'importe quelle taille. `Write`

Cela peut surprendre si vous venez de C# ou de Java, mais la raison est simple. En Java, une variable de type (l'interface standard Java analogue à) est une référence à tout objet qui implémente . Le fait qu'il s'agisse d'une référence va de soi. C'est la même chose avec les interfaces en C# et dans la plupart des autres langages.

```
OutputStream std::io::Write OutputStream
```

Ce que nous voulons dans Rust, c'est la même chose, mais dans Rust, les références sont explicites :

```

let mut buf: Vec<u8> = vec![];
let writer: &mut dyn Write = &mut buf; // ok

```

Une référence à un type de trait, comme , est appelée un *objet de trait*. Comme toute autre référence, un objet trait pointe vers une valeur, il a une durée de vie et il peut être partagé ou partagé. `writer mut`

Ce qui rend un objet trait différent, c'est que Rust ne connaît généralement pas le type du référent au moment de la compilation. Ainsi, un objet trait comprend un peu plus d'informations sur le type du référent. Ceci est strictement pour l'usage propre de Rust dans les coulisses: lorsque vous appelez , Rust a besoin des informations de type pour appeler dynamiquement la bonne méthode en fonction du type de . Vous ne pouvez pas interroger directement les informations de type, et Rust ne prend pas en charge le downcasting de l'objet trait vers un type concret comme `.writer.write(data)` write *writer &mut dyn Write Vec<u8>

Disposition de l'objet Trait

En mémoire, un objet trait est un pointeur gras composé d'un pointeur vers la valeur, plus un pointeur vers une table représentant le type de cette valeur. Chaque objet trait prend donc deux mots machine, comme le montre [la figure 11-1](#).

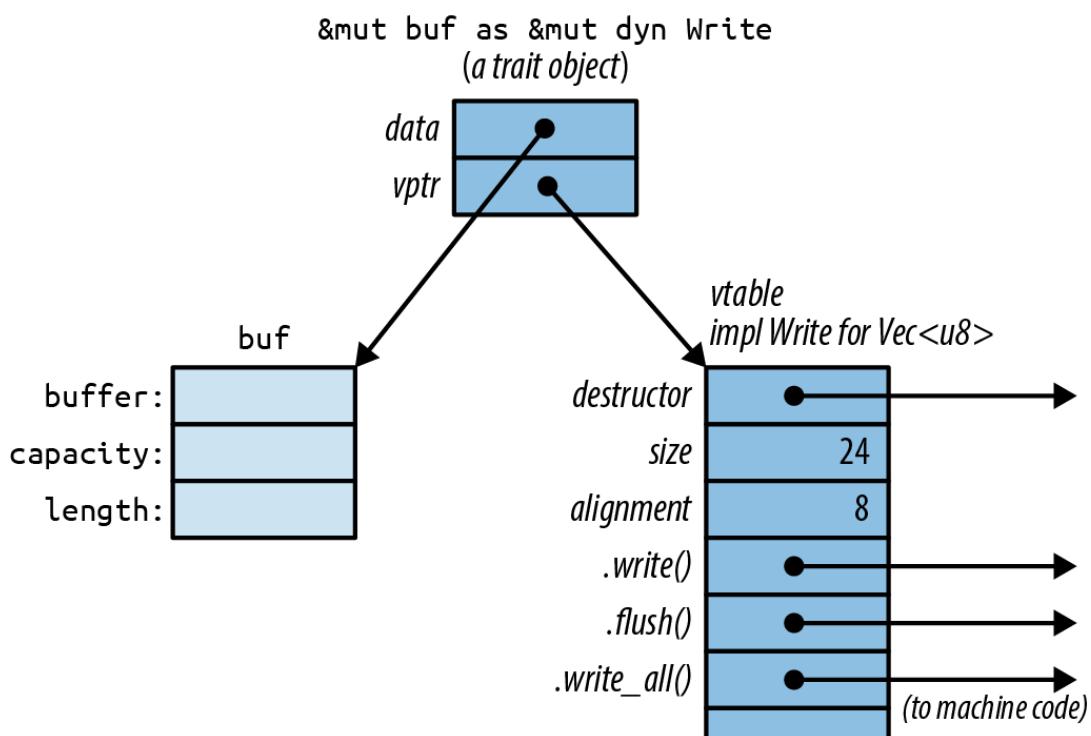


Figure 11-1. Objets traits en mémoire

C++ contient également ce type d'informations de type d'exécution. C'est ce qu'on appelle une *table virtuelle*, ou *vtable*. Dans Rust, comme en C++, le vtable est généré une fois, au moment de la compilation, et partagé par

tous les objets du même type. Tout ce qui est montré dans la teinte la plus sombre de [la figure 11-1](#), y compris le vtable, est un détail d'implémentation privé de Rust. Encore une fois, ce ne sont pas des champs et des structures de données auxquels vous pouvez accéder directement. Au lieu de cela, le langage utilise automatiquement le vtable lorsque vous appelez une méthode d'un objet trait pour déterminer l'implémentation à appeler.

Les programmeurs C++ chevronnés remarqueront que Rust et C++ utilisent la mémoire un peu différemment. En C++, le pointeur vtable, ou *vptr*, est stocké dans le cadre de la structure. Rust utilise plutôt des pointeurs de graisse. La structure elle-même ne contient rien d'autre que ses champs. De cette façon, une structure peut implémenter des dizaines de traits sans contenir des dizaines de vptrs. Même des types comme , qui ne sont pas assez grands pour accueillir un vptr, peuvent implémenter des traits. ⁱ³²

Rust convertit automatiquement les références ordinaires en objets traits en cas de besoin. C'est pourquoi nous sommes en mesure de passer à dans cet exemple: `&mut local_file say_hello`

```
let mut local_file = File::create("hello.txt")?;
say_hello(&mut local_file);
```

Le type de est , et le type de l'argument à est . Puisque a est une sorte d'écrivain, Rust le permet, convertissant automatiquement la référence simple en un objet trait. `&mut local_file &mut`
`File say_hello &mut dyn Write`

De même, Rust convertira volontiers a en , une valeur qui possède un écrivain dans le tas: `Box<File>` `Box<dyn Write>`

```
let w: Box<dyn Write> = Box::new(local_file);
```

`Box<dyn Write>` , comme , est un gros pointeur : il contient l'adresse de l'auteur lui-même et l'adresse du vtable. Il en va de même pour d'autres types de pointeurs, comme . `&mut dyn Write Rc<dyn Write>`

Ce type de conversion est le seul moyen de créer un objet trait. Ce que le compilateur fait réellement ici est très simple. Au moment où la conversion se produit, Rust connaît le vrai type du référent (dans ce cas,), il ajoute donc simplement l'adresse du vtable approprié, transformant le pointeur régulier en un gros pointeur. `File`

Fonctions génériques et paramètres de type

Au début de ce chapitre, nous avons montré une fonction qui prenait un objet trait comme argument. Réécrivons cette fonction en tant que fonction générique : `say_hello()`

```
fn say_hello<W: Write>(out: &mut W) -> std::io::Result<()> {
    out.write_all(b"hello world\n")?;
    out.flush()
}
```

Seule la signature de type a changé :

```
fn say_hello(out: &mut dyn Write) // plain function

fn say_hello<W: Write>(out: &mut W) // generic function
```

La phrase est ce qui rend la fonction générique. Il s'agit d'un *paramètre de type*. Cela signifie que dans tout le corps de cette fonction, représente un type qui implémente le trait. Les paramètres de type sont généralement des lettres majuscules simples, par convention. `<W: Write>` `W` `Write`

Le type signifie dépend de la façon dont la fonction générique est utilisée: `w`

```
say_hello(&mut local_file)?; // calls say_hello::<File>
say_hello(&mut bytes)?; // calls say_hello::<Vec<u8>>
```

Lorsque vous passez à la fonction générique, vous appelez `.` Rust génère du code machine pour cette fonction qui appelle `.` Lorsque vous passez `,`, vous appelez `.` Rust génère un code machine distinct pour cette version de la fonction, en appelant les méthodes correspondantes. Dans les deux cas, Rust déduit le type du type de l'argument. Ce processus est connu sous le nom *de monomorphisation*, et le compilateur gère tout cela automatiquement.

```
&mut local_file say_hello() say_hello::<File>()
File::write_all() File::flush() &mut bytes say_hello::<Vec<u8>>()
Vec<u8> w
```

Vous pouvez toujours épeler les paramètres de type :

```
say_hello::<File>(&mut local_file)?;
```

Ceci est rarement nécessaire, car Rust peut généralement déduire les paramètres de type en regardant les arguments. Ici, la fonction générique attend un argument, et nous lui passons un , donc Rust en déduit que

```
.say_hello &mut W &mut File W = File
```

Si la fonction générique que vous appelez n'a pas d'arguments qui four-nissent des indices utiles, vous devrez peut-être l'épeler :

```
// calling a generic method collect<C>() that takes no arguments
let v1 = (0 .. 1000).collect(); // error: can't infer type
let v2 = (0 .. 1000).collect::<Vec<i32>>(); // ok
```

Parfois, nous avons besoin de plusieurs capacités à partir d'un paramètre de type. Par exemple, si nous voulons imprimer les dix valeurs les plus courantes dans un vecteur, nous aurons besoin que ces valeurs soient imprimables :

```
use std::fmt::Debug;

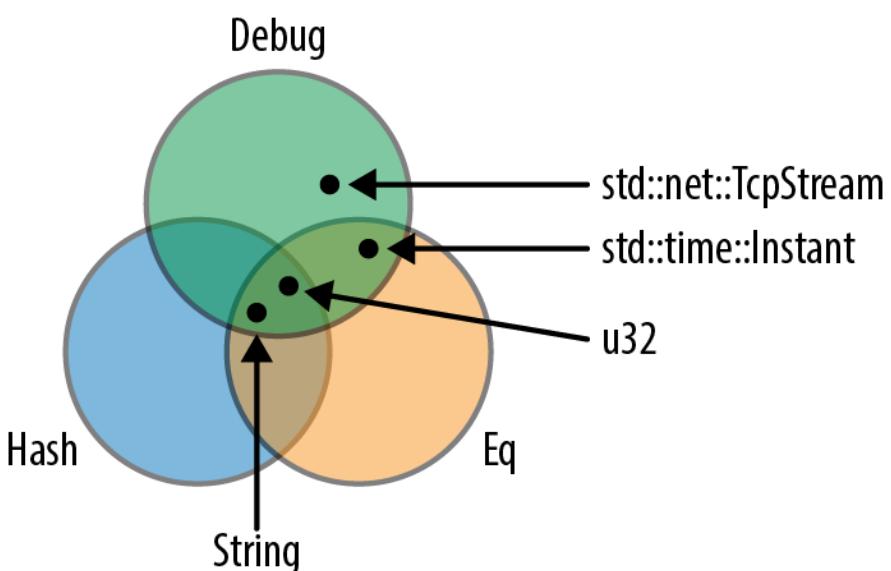
fn top_ten<T: Debug>(values: &Vec<T>) { ... }
```

Mais ce n'est pas suffisant. Comment prévoyons-nous de déterminer quelles valeurs sont les plus courantes? La méthode habituelle consiste à utiliser les valeurs comme clés dans une table de hachage. Cela signifie que les valeurs doivent soutenir les opérations. Les limites sur doivent inclure celles-ci ainsi que . La syntaxe utilise le signe : Hash Eq T Debug +

```
use std::hash::Hash;
use std::fmt::Debug;

fn top_ten<T: Debug + Hash + Eq>(values: &Vec<T>) { ... }
```

Certains types implémentent , certains implémentent , certains prennent en charge , et quelques-uns, comme et , implémentent les trois, comme le montre [la figure 11-2](#). Debug Hash Eq u32 String



Graphique 11-2. Traits en tant qu'ensembles de types

Il est également possible qu'un paramètre de type n'ait aucune limite, mais vous ne pouvez pas faire grand-chose avec une valeur si vous n'avez spécifié aucune limite pour celle-ci. Vous pouvez le déplacer. Vous pouvez le mettre dans une boîte ou un vecteur. C'est à peu près tout.

Les fonctions génériques peuvent avoir plusieurs paramètres de type :

```
/// Run a query on a large, partitioned data set.
/// See <http://research.google.com/archive/mapreduce.html>.
fn run_query<M: Mapper + Serialize, R: Reducer + Serialize>(
    data: &DataSet, map: M, reduce: R) -> Results
{ ... }
```

Comme le montre cet exemple, les limites peuvent être si longues qu'elles sont dures pour les yeux. Rust fournit une syntaxe alternative en utilisant le mot-clé : where

```
fn run_query<M, R>(data: &DataSet, map: M, reduce: R) -> Results
    where M: Mapper + Serialize,
          R: Reducer + Serialize
{ ... }
```

Les paramètres de type sont toujours déclarés à l'avance, mais les limites sont déplacées vers des lignes séparées. Ce type de clause est également autorisé sur les structs génériques, les enums, les alias de type et les méthodes, partout où les limites sont autorisées. M R where

Bien sûr, une alternative aux clauses est de rester simple: trouver un moyen d'écrire le programme sans utiliser de génériques de manière aussi intensive. where

« [Réception de références en tant qu'arguments de fonction](#) » a introduit la syntaxe des paramètres de durée de vie. Une fonction générique peut avoir à la fois des paramètres de durée de vie et des paramètres de type. Les paramètres de durée de vie viennent en premier:

```
/// Return a reference to the point in `candidates` that's
/// closest to the `target` point.
fn nearest<'t, 'c, P>(target: &'t P, candidates: &'c [P]) -> &'c P
    where P: MeasureDistance
{
    ...
}
```

Cette fonction prend deux arguments, et . Les deux sont des références, et nous leur donnons des durées de vie distinctes et (comme discuté dans [« Paramètres de durée de vie distincts »](#)). De plus, la fonction fonctionne avec n'importe quel type qui implémente le trait, nous pouvons donc l'utiliser sur des valeurs dans un programme et des valeurs dans un autre. target candidates 't 'c P MeasureDistance Point2d Point3d

Les durées de vie n'ont jamais d'impact sur le code machine. Deux appels à l'utilisation du même type, mais des durées de vie différentes, appelleront la même fonction compilée. Seuls des types différents amènent Rust à compiler plusieurs copies d'une fonction générique. nearest() P

En plus des types et des durées de vie, les fonctions génériques peuvent également prendre des paramètres constants, comme la structure que nous avons présentée dans [« Structs génériques avec des paramètres constants »](#): Polynomial

```
fn dot_product<const N: usize>(a: [f64; N], b: [f64; N]) -> f64 {
    let mut sum = 0.;
    for i in 0..N {
        sum += a[i] * b[i];
    }
    sum
}
```

Ici, la phrase indique que la fonction attend un paramètre générique , qui doit être un . Étant donné , la fonction prend deux arguments de type , et additionne les produits de leurs éléments correspondants. Ce qui distingue d'un argument ordinaire, c'est que vous pouvez l'utiliser dans les

```
types dans la signature ou le corps de . <const N:  
usize> dot_product N usize N [f64; N] N usize dot_product
```

Comme pour les paramètres de type, vous pouvez soit fournir explicitement des paramètres constants, soit laisser Rust les déduire :

```
// Explicitly provide `3` as the value for `N`.  
dot_product::<3>([0.2, 0.4, 0.6], [0., 0., 1.])  
  
// Let Rust infer that `N` must be `2`.  
dot_product([3., 4.], [-5., 1.])
```

Bien sûr, les fonctions ne sont pas le seul type de code générique dans Rust:

- Nous avons déjà couvert les types [génériques dans « Generic Structs »](#) et [« Generic Enums »](#).
- Une méthode individuelle peut être générique, même si le type sur lequel elle est définie n'est pas générique :

```
impl PancakeStack {  
    fn push<T: Topping>(&mut self, goop: T) -> PancakeResult<()> {  
        goop.pour(&self);  
        self.absorb_topping(goop)  
    }  
}
```

- Les alias de type peuvent également être génériques :

```
type PancakeResult<T> = Result<T, PancakeError>;
```

- Nous couvrirons les traits génériques plus loin dans ce chapitre.

Toutes les fonctionnalités présentées dans cette section (limites, clauses, paramètres de durée de vie, etc.) peuvent être utilisées sur tous les éléments génériques, pas seulement sur les fonctions. `where`

Lequel utiliser

Le choix d'utiliser des objets de trait ou du code générique est subtil. Étant donné que les deux caractéristiques sont basées sur des traits, elles ont beaucoup en commun.

Les objets traits sont le bon choix chaque fois que vous avez besoin d'une collection de valeurs de types mixtes, tous ensemble. Il est techniquement

possible de faire une salade générique:

```
trait Vegetable {  
    ...  
}  
  
struct Salad<V: Vegetable> {  
    veggies: Vec<V>  
}
```

Cependant, il s'agit d'une conception plutôt sévère. Chacune de ces salades se compose entièrement d'un seul type de légume. Tout le monde n'est pas fait pour ce genre de chose. L'un de vos auteurs a déjà payé 14 \$ pour un et n'a jamais tout à fait surmonté l'expérience. `Salad<IcebergLettuce>`

Comment pouvons-nous construire une meilleure salade? Étant donné que les valeurs peuvent être toutes de tailles différentes, nous ne pouvons pas demander à Rust un `: Vegetable Vec<dyn Vegetable>`

```
struct Salad {  
    veggies: Vec<dyn Vegetable> // error: `dyn Vegetable` does  
                                // not have a constant size  
}
```

Les objets traits sont la solution :

```
struct Salad {  
    veggies: Vec<Box<dyn Vegetable>>  
}
```

Chacun peut posséder n'importe quel type de légume, mais la boîte elle-même a une taille constante – deux pointeurs – adaptée au stockage dans un vecteur. Mis à part la métaphore malheureuse et mixte d'avoir des boîtes dans sa nourriture, c'est précisément ce qui est nécessaire, et cela fonctionnerait tout aussi bien pour les formes dans une application de dessin, les monstres dans un jeu, les algorithmes de routage enfichables dans un routeur réseau, etc. `Box<dyn Vegetable>`

Une autre raison possible d'utiliser des objets traits est de réduire la quantité totale de code compilé. Rust peut avoir à compiler une fonction générique plusieurs fois, une fois pour chaque type avec lequel elle est utilisée. Cela pourrait rendre le binaire grand, un phénomène appelé *gonflement du code* dans les cercles C++. De nos jours, la mémoire est abon-

dante et la plupart d'entre nous ont le luxe d'ignorer la taille du code; mais il existe des environnements contraints.

En dehors des situations impliquant des environnements à salade ou à faibles ressources, les génériques présentent trois avantages importants par rapport aux objets traits, de sorte que dans Rust, les génériques sont le choix le plus courant.

Le premier avantage est la vitesse. Notez l'absence du mot-clé dans les signatures de fonction génériques. Étant donné que vous spécifiez les types au moment de la compilation, explicitement ou par inférence de type, le compilateur sait exactement quelle méthode appeler. Le mot-clé n'est pas utilisé car il n'y a pas d'objets traits , et donc pas de répartition dynamique , impliqués. `dyn write dyn`

La fonction générique montrée dans l'introduction est aussi rapide que si nous avions écrit des fonctions séparées , , , et ainsi de suite. Le compilateur peut l'intégrer, comme n'importe quelle autre fonction, donc dans une version release, un appel à n'est probablement que deux ou trois instructions. Un appel avec des arguments constants, comme , sera encore plus rapide: Rust peut l'évaluer au moment de la compilation, de sorte qu'il n'y a aucun coût

d'exécution. `min() min_u8 min_i64 min_string min::<i32> min(5, 3)`

Ou considérez cet appel de fonction générique :

```
let mut sink = std::io::sink();
say_hello(&mut sink)?;
```

`std::io::sink()` renvoie un graveur de type qui ignore discrètement tous les octets qui y sont écrits. `Sink`

Lorsque Rust génère du code machine pour cela, il peut émettre du code qui appelle , vérifie les erreurs, puis appelle . C'est ce que le corps de la fonction générique dit de faire. `Sink::write_all Sink::flush`

Ou, Rust pourrait examiner ces méthodes et réaliser ce qui suit:

- `Sink::write_all()` ne fait rien.
- `Sink::flush()` ne fait rien.
- Aucune des deux méthodes ne renvoie jamais d'erreur.

En bref, Rust dispose de toutes les informations dont il a besoin pour optimiser complètement cet appel de fonction.

Comparez cela au comportement avec des objets traits. Rust ne sait jamais vers quel type de valeur pointe un objet trait jusqu'au moment de l'exécution. Ainsi, même si vous passez un , la surcharge d'appel de méthodes virtuelles et de vérification des erreurs s'applique toujours. Sink

Le deuxième avantage des génériques est que tous les traits ne peuvent pas prendre en charge des objets de traits. Les traits prennent en charge plusieurs fonctionnalités, telles que les fonctions associées, qui ne fonctionnent qu'avec des génériques: ils excluent complètement les objets de trait. Nous soulignerons ces caractéristiques au fur et à mesure que nous y arriverons.

Le troisième avantage des génériques est qu'il est facile de lier un paramètre de type générique avec plusieurs traits à la fois, comme notre fonction l'a fait lorsqu'elle a eu besoin de son paramètre pour implémenter . Les objets Trait ne peuvent pas faire cela : des types comme ceux-ci ne sont pas pris en charge dans Rust. (Vous pouvez contourner ce problème avec des sous-traits, définis plus loin dans ce chapitre, mais c'est un peu impliqué.) `top_ten T Debug + Hash + Eq &mut (dyn Debug + Hash + Eq)`

Définition et mise en œuvre des traits

Définir un trait est simple. Donnez-lui un nom et listez les signatures de type des méthodes de trait. Si nous écrivons un jeu, nous pourrions avoir un trait comme celui-ci:

```
/// A trait for characters, items, and scenery -  
/// anything in the game world that's visible on screen.  
trait Visible {  
    /// Render this object on the given canvas.  
    fn draw(&self, canvas: &mut Canvas);  
  
    /// Return true if clicking at (x, y) should  
    /// select this object.  
    fn hit_test(&self, x: i32, y: i32) -> bool;  
}
```

Pour implémenter un trait, utilisez la syntaxe : `impl TraitName for Type`

```
impl Visible for Broom {  
    fn draw(&self, canvas: &mut Canvas) {  
        for y in self.y - self.height - 1 .. self.y {
```

```

        canvas.write_at(self.x, y, '|');
    }

fn hit_test(&self, x: i32, y: i32) -> bool {
    self.x == x
    && self.y - self.height - 1 <= y
    && y <= self.y
}
}

```

Notez que cela contient une implémentation pour chaque méthode du trait, et rien d'autre. Tout ce qui est défini dans un trait doit en fait être une caractéristique du trait; si nous voulions ajouter une méthode d'assistance à l'appui de , nous devrions la définir dans un bloc séparé:

```
impl Visible impl Broom::draw() impl
```

```

impl Broom {
    /// Helper function used by Broom::draw() below.
    fn broomstick_range(&self) -> Range<i32> {
        self.y - self.height - 1 .. self.y
    }
}

```

Ces fonctions d'assistance peuvent être utilisées dans les blocs de traits :

```
:impl
```

```

impl Visible for Broom {
    fn draw(&self, canvas: &mut Canvas) {
        for y in self.broomstick_range() {
            ...
        }
        ...
    }
    ...
}

```

Méthodes par défaut

Le type d'enregistreur dont nous avons parlé précédemment peut être implémenté en quelques lignes de code. Tout d'abord, nous définissons le type:

```
type: Sink
```

```
/// A Writer that ignores whatever data you write to it.  
pub struct Sink;
```

Sink est une structure vide, car nous n'avons pas besoin d'y stocker de données. Ensuite, nous fournissons une implémentation du trait pour :Write Sink

```
use std::io::{Write, Result};  
  
impl Write for Sink {  
    fn write(&mut self, buf: &[u8]) -> Result<usize> {  
        // Claim to have successfully written the whole buffer.  
        Ok(buf.len())  
    }  
  
    fn flush(&mut self) -> Result<()> {  
        Ok(())  
    }  
}
```

Jusqu'à présent, cela ressemble beaucoup au trait. Mais nous avons également vu que le trait a une méthode: visible Write write_all

```
let mut out = Sink;  
out.write_all(b"hello world\n")?;
```

Pourquoi Rust nous laisse-t-il sans définir cette méthode ? La réponse est que la définition du trait de la bibliothèque standard contient une *implémentation par défaut* pour :impl Write for Sink Write write_all

```
trait Write {  
    fn write(&mut self, buf: &[u8]) -> Result<usize>;  
    fn flush(&mut self) -> Result<()>;  
  
    fn write_all(&mut self, buf: &[u8]) -> Result<()> {  
        let mut bytes_written = 0;  
        while bytes_written < buf.len() {  
            bytes_written += self.write(&buf[bytes_written..])?;  
        }  
        Ok(())  
    }  
    ...  
}
```

Les méthodes et sont les méthodes de base que chaque écrivain doit mettre en œuvre. Un rédacteur peut également implémenter , mais sinon, l'implémentation par défaut indiquée précédemment sera utilisée.`write flush write_all`

Vos propres caractéristiques peuvent inclure des implémentations par défaut utilisant la même syntaxe.

L'utilisation la plus spectaculaire des méthodes par défaut dans la bibliothèque standard est le trait, qui a une méthode requise () et des dizaines de méthodes par défaut. [Le chapitre 15](#) explique pourquoi. `Iterator .next()`

Traits et types d'autres personnes

Rust vous permet d'implémenter n'importe quel trait sur n'importe quel type, tant que le trait ou le type est introduit dans la caisse actuelle.

Cela signifie que chaque fois que vous souhaitez ajouter une méthode à n'importe quel type, vous pouvez utiliser un trait pour le faire:

```
trait IsEmoji {
    fn is_emoji(&self) -> bool;
}

/// Implement IsEmoji for the built-in character type.
impl IsEmoji for char {
    fn is_emoji(&self) -> bool {
        ...
    }
}

assert_eq!('$'.is_emoji(), false);
```

Comme toute autre méthode de trait, cette nouvelle méthode n'est visible que lorsqu'elle est dans la portée. `is_emoji IsEmoji`

Le seul but de ce trait particulier est d'ajouter une méthode à un type existant, . C'est ce qu'on appelle un *trait d'extension*. Bien sûr, vous pouvez également ajouter ce trait aux types en écrivant et ainsi de suite. `char impl IsEmoji for str { ... }`

Vous pouvez même utiliser un bloc générique pour ajouter un trait d'extension à toute une famille de types à la fois. Ce trait pourrait être mis en œuvre sur n'importe quel type: `impl`

```

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

/// Trait for values to which you can send HTML.
trait WriteHtml {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()>;
}

```

L'implémentation du trait pour tous les écrivains en fait un trait d'extension, ajoutant une méthode à tous les écrivains Rust:

```

/// You can write HTML to any std::io writer.
impl<W: Write> WriteHtml for W {
    fn write_html(&mut self, html: &HtmlDocument) -> io::Result<()> {
        ...
    }
}

```

La ligne signifie « pour chaque type qui implémente , voici une implémentation de pour » `impl<W: Write> WriteHtml for W`

La bibliothèque offre un bel exemple de l'utilité d'implémenter des traits définis par l'utilisateur sur des types standard. est une bibliothèque de sérialisation. Autrement dit, vous pouvez l'utiliser pour écrire des structures de données Rust sur le disque et les recharger ultérieurement. La bibliothèque définit un trait, , qui est implémenté pour chaque type de données pris en charge par la bibliothèque. Donc, dans le code source, il y a du code implémentant pour , , , les types de tableau et de tuple, et ainsi de suite, à travers toutes les structures de données standard comme et .
`.serde` `serde Serialize` `serde Serialize` `bool i8 i16 i32 Vec H ashMap`

Le résultat de tout cela est qu'il ajoute une méthode à tous ces types. Il peut être utilisé comme ceci: `serde .serialize()`

```

use serde::Serialize;
use serde_json;

pub fn save_configuration(config: &HashMap<String, String>)
    -> std::io::Result<()>
{
    // Create a JSON serializer to write the data to a file.
    let writer = File::create(config_filename())?;
    let mut serializer = serde_json::Serializer::new(writer);

```

```
// The serde `serialize()` method does the rest.
config.serialize(&mut serializer)?;

Ok(())
}
```

Nous avons dit plus tôt que lorsque vous implémentez un trait, le trait ou le type doit être nouveau dans la caisse actuelle. C'est ce qu'on appelle la *règle orpheline*. Il aide Rust à s'assurer que les implementations de traits sont uniques. Votre code ne peut pas , car les deux et sont définis dans la bibliothèque standard. Si Rust laissait les caisses faire cela, il pourrait y avoir plusieurs implementations de pour , dans différentes caisses, et Rust n'aurait aucun moyen raisonnable de décider quelle implementation utiliser pour un appel de méthode donné. `impl Write for u8 Write u8 Write u8`

(C++ a une restriction d'unicité similaire : la règle de définition unique. En C++ typique, il n'est pas appliqué par le compilateur, sauf dans les cas les plus simples, et vous obtenez un comportement indéfini si vous le cassez.)

Soi dans les traits

Un trait peut utiliser le mot-clé comme type. Le trait standard, par exemple, ressemble à ceci (légèrement simplifié): `Self Clone`

```
pub trait Clone {
    fn clone(&self) -> Self;
    ...
}
```

L'utilisation comme type de retour ici signifie que le type de est le même que le type de , quel qu'il soit. Si est un , alors le type de est —pas ou tout autre type

`clonable. Self x.clone() x x String x.clone() String dyn Clone`

De même, si nous définissons ce trait :

```
pub trait Spliceable {
    fn splice(&self, other: &Self) -> Self;
}
```

avec deux implementations :

```

impl Spliceable for CherryTree {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

impl Spliceable for Mammoth {
    fn splice(&self, other: &Self) -> Self {
        ...
    }
}

```

puis à l'intérieur du premier , est simplement un alias pour , et dans le second, c'est un alias pour . Cela signifie que nous pouvons épisser ensemble deux cerisiers ou deux mammouths, pas que nous pouvons créer un hybride mammouth-cerise. Le type et le type de doivent correspondre. `impl Self CherryTree Mammoth self other`

Un trait qui utilise le type est incompatible avec les objets trait : `Self`

```

// error: the trait `Spliceable` cannot be made into an object
fn splice_anything(left: &dyn Spliceable, right: &dyn Spliceable) {
    let combo = left.splice(right);
    // ...
}

```

La raison en est quelque chose que nous verrons encore et encore alors que nous creusons dans les fonctionnalités avancées des traits. Rust rejette ce code car il n'a aucun moyen de taper-vérifier l'appel . Tout l'intérêt des objets traits est que le type n'est pas connu avant l'exécution. Rust n'a aucun moyen de savoir au moment de la compilation si et sera le même type, selon les besoins. `left.splice(right)` `left right`

Les objets Trait sont vraiment destinés aux types de traits les plus simples, les types qui pourraient être implémentés à l'aide d'interfaces en Java ou de classes de base abstraites en C++. Les fonctionnalités plus avancées des traits sont utiles, mais elles ne peuvent pas coexister avec les objets traits car avec les objets traits, vous perdez les informations de type dont Rust a besoin pour vérifier votre programme.

Maintenant, si nous avions voulu un épissage génétiquement improbable, nous aurions pu concevoir un trait respectueux des objets:

```

pub trait MegaSpliceable {
    fn splice(&self, other: &dyn MegaSpliceable) -> Box<dyn MegaSpliceable>
}

```

Ce trait est compatible avec les objets traits. Il n'y a pas de problème de vérification de type des appels à cette méthode car le type de l'argument n'est pas nécessaire pour correspondre au type de , tant que les deux types sont ..splice() other self MegaSpliceable

Sous-trait

Nous pouvons déclarer qu'un trait est une extension d'un autre trait :

```

/// Someone in the game world, either the player or some other
/// pixie, gargoyle, squirrel, ogre, etc.
trait Creature: Visible {
    fn position(&self) -> (i32, i32);
    fn facing(&self) -> Direction;
    ...
}

```

La phrase signifie que toutes les créatures sont visibles. Chaque type qui implémente doit également implémenter le trait : trait Creature: Visible Creature Visible

```

impl Visible for Broom {
    ...
}

impl Creature for Broom {
    ...
}

```

Nous pouvons implémenter les deux traits dans l'un ou l'autre ordre, mais c'est une erreur à implémenter pour un type sans implémenter également . Ici, nous disons que c'est un *sous-trait* de , et c'est le *supertrait* de . Creature Visible Creature Visible Visible Creature

Les sous-trait ressemblent à des sous-interfaces en Java ou en C#, en ce sens que les utilisateurs peuvent supposer que toute valeur qui implémente un sous-trait implémente également son supertrait. Mais dans Rust, un subtrait n'hérite pas des éléments associés de son supertrait; chaque trait doit toujours être dans la portée si vous voulez appeler ses méthodes.

En fait, les sous-trait de Rust ne sont en réalité qu'un raccourci pour une liaison sur . Une définition de ce genre est exactement équivalente à celle montrée précédemment: `Self Creature`

```
trait Creature where Self: Visible {  
    ...  
}
```

Fonctions associées au type

Dans la plupart des langages orientés objet, les interfaces ne peuvent pas inclure de méthodes statiques ou de constructeurs, mais les traits peuvent inclure des fonctions associées au type, les méthodes analogiques à statiques de Rust :

```
trait StringSet {  
    /// Return a new empty set.  
    fn new() -> Self;  
  
    /// Return a set that contains all the strings in `strings`.  
    fn from_slice(strings: &[&str]) -> Self;  
  
    /// Find out if this set contains a particular `value`.  
    fn contains(&self, string: &str) -> bool;  
  
    /// Add a string to this set.  
    fn add(&mut self, string: &str);  
}
```

Chaque type qui implémente le trait doit implémenter ces quatre fonctions associées. Les deux premiers, et , ne prenez pas d'argument. Ils servent de constructeurs. Dans le code non générique, ces fonctions peuvent être appelées à l'aide de la syntaxe, comme toute autre fonction associée à un type:
`StringSet new() from_slice() self ::`

```
// Create sets of two hypothetical types that impl StringSet:  
let set1 = SortedStringSet::new();  
let set2 = HashedStringSet::new();
```

Dans le code générique, c'est la même chose, sauf que le type est souvent une variable de type, comme dans l'appel à montré ici :
`s::new()`

```
/// Return the set of words in `document` that aren't in `wordlist`.  
fn unknown_words<S: StringSet>(document: &[String], wordlist: &S) -> S {
```

```

let mut unknowns = S::new();
for word in document {
    if !wordlist.contains(word) {
        unknowns.add(word);
    }
}
unknowns
}

```

Comme les interfaces Java et C#, les objets traits ne prennent pas en charge les fonctions associées au type. Si vous souhaitez utiliser des objets de trait, vous devez modifier le trait, en ajoutant la liaison à chaque fonction associée qui ne prend pas un argument par référence : &dyn

```
StringSet where Self: Sized self
```

```

trait StringSet {
    fn new() -> Self
        where Self: Sized;

    fn from_slice(strings: &[&str]) -> Self
        where Self: Sized;

    fn contains(&self, string: &str) -> bool;

    fn add(&mut self, string: &str);
}

```

Cette liaison indique à Rust que les objets traits sont dispensés de prendre en charge cette fonction associée particulière. Avec ces ajouts, les objets traits sont autorisés; ils ne prennent toujours pas en charge ou , mais vous pouvez les créer et les utiliser pour appeler et . La même astuce fonctionne pour toute autre méthode incompatible avec les objets traits. (Nous renoncerons à l'explication technique plutôt fastidieuse de la raison pour laquelle cela fonctionne, mais le trait est couvert au [chapitre 13.](#))

```
StringSet new from_slice .contains() .add() Sized
```

Appels de méthode complets

Toutes les méthodes d'appel que nous avons vues jusqu'à présent reposent sur Rust pour combler certaines pièces manquantes pour vous. Par exemple, supposons que vous écriviez ce qui suit :

```
"hello".to_string()
```

Il est entendu que cela fait référence à la méthode du trait, dont nous appelons l'implémentation du type. Il y a donc quatre acteurs dans ce jeu : le trait, la méthode de ce trait, la mise en œuvre de cette méthode et la valeur à laquelle cette implémentation est appliquée. C'est formidable que nous n'ayons pas à épeler tout cela chaque fois que nous voulons appeler une méthode. Mais dans certains cas, vous avez besoin d'un moyen de dire exactement ce que vous voulez dire. Les appels de méthode entièrement qualifiés correspondent à la

```
facture.to_string to_string ToString str
```

Tout d'abord, il est utile de savoir qu'une méthode n'est qu'un type spécial de fonction. Ces deux appels sont équivalents :

```
"hello".to_string()  
  
str::to_string("hello")
```

Le deuxième formulaire ressemble exactement à un appel de fonction associé. Cela fonctionne même si la méthode prend un argument. Passez simplement comme premier argument de la

```
fonction.to_string self self
```

Puisqu'il s'agit d'une méthode du trait standard, il existe deux autres formes que vous pouvez utiliser: `to_string` `ToString`

```
ToString::to_string("hello")  
  
<str as ToString>::to_string("hello")
```

Ces quatre appels de méthode font exactement la même chose. Le plus souvent, vous n'écrirez que . Les autres formulaires sont des appels *de méthode qualifiés*. Ils spécifient le type ou le trait auquel une méthode est associée. Le dernier formulaire, avec les crochets d'angle, spécifie les deux : un *appel de méthode complet*. `value.method()`

Lorsque vous écrivez , à l'aide de l'opérateur, vous ne dites pas exactement quelle méthode vous appelez. Rust a un algorithme de recherche de méthode qui comprend cela, en fonction des types, des coercitions de référence, etc. Avec des appels complets, vous pouvez dire exactement quelle méthode vous voulez dire, et cela peut aider dans quelques cas étranges: "hello".`to_string()` . `to_string`

- Lorsque deux méthodes portent le même nom. L'exemple classique de hokey est celui avec deux méthodes de deux traits différents, l'une pour

le dessiner à l'écran et l'autre pour interagir avec la loi: `Outlaw .draw()`

```
outlaw.draw(); // error: draw on screen or draw pistol?  
  
Visible::draw(&outlaw); // ok: draw on screen  
HasPistol::draw(&outlaw); // ok: corral
```

Habituellement, vous feriez mieux de renommer l'une des méthodes, mais parfois vous ne pouvez pas.

- Lorsque le type de l'argument ne peut pas être déduit : `self`

```
let zero = 0; // type unspecified; could be `i8`, `u8`, ...  
  
zero.abs(); // error: can't call method `abs`  
            // on ambiguous numeric type  
  
i64::abs(zero); // ok
```

- Lorsque vous utilisez la fonction elle-même comme valeur de fonction :

```
let words: Vec<String> =  
    line.split_whitespace() // iterator produces &str values  
        .map(ToString::to_string) // ok  
        .collect();
```

- Lors de l'appel de méthodes de trait dans des macros. Nous expliquerons au [chapitre 21](#).

La syntaxe complète fonctionne également pour les fonctions associées. Dans la section précédente, nous avons écrit pour créer un nouvel ensemble dans une fonction générique. Nous aurions aussi pu écrire ou

```
.S::new() StringSet::new() <S as StringSet>::new()
```

Caractéristiques qui définissent les relations entre les types

Jusqu'à présent, chaque trait que nous avons examiné est autonome: un trait est un ensemble de méthodes que les types peuvent mettre en œuvre. Les traits peuvent également être utilisés dans des situations où plusieurs types doivent travailler ensemble. Ils peuvent décrire les relations entre les types.

- Le trait relie chaque type d'itérateur au type de valeur qu'il produit. `std::iter::Iterator`
- Le trait concerne les types qui peuvent être multipliés. Dans l'expression `a * b`, les valeurs et peuvent être soit du même type, soit de types différents. `std::ops::Mul a b`
- La caisse comprend à la fois un trait pour les générateurs de nombres aléatoires () et un trait pour les types qui peuvent être générés aléatoirement (). Les traits eux-mêmes définissent exactement comment ces types fonctionnent ensemble. `rand rand::Rng rand::Distribution`

Vous n'aurez pas besoin de créer des traits comme ceux-ci tous les jours, mais vous les rencontrerez dans toute la bibliothèque standard et dans des caisses tierces. Dans cette section, nous montrerons comment chacun de ces exemples est implémenté, en reprenant les fonctionnalités pertinentes du langage Rust au fur et à mesure que nous en avons besoin. La compétence clé ici est la capacité de lire les traits et les signatures de méthode et de comprendre ce qu'ils disent sur les types impliqués.

Types associés (ou fonctionnement des itérateurs)

Nous allons commencer par les itérateurs. À l'heure actuelle, chaque langage orienté objet dispose d'une sorte de support intégré pour les itérateurs, des objets qui représentent la traversée d'une séquence de valeurs.

La rouille a un trait standard, défini comme ceci: `Iterator`

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

La première caractéristique de ce trait, , est un *type associé*. Chaque type qui implémente doit spécifier le type d'élément qu'il produit. `type Item: Iterator`

La deuxième fonctionnalité, la méthode, utilise le type associé dans sa valeur de retour. renvoie un : , soit la valeur suivante de la séquence, soit lorsqu'il n'y a plus de valeurs à visiter. Le type est écrit comme , pas seulement simple , car il s'agit d'une caractéristique de chaque type d'itérateur, et non d'un type autonome. Comme toujours, et le type apparaît explicitement dans le code partout où leurs champs, méthodes, etc. sont

```
utilisés. next() next() Option<Self::Item> Some(item) None Self
::Item Item Item self Self
```

Voici à quoi ressemble l'implémentation d'un type : Iterator

```
// (code from the std::env standard library module)
impl Iterator for Args {
    type Item = String;

    fn next(&mut self) -> Option<String> {
        ...
    }
    ...
}
```

`std::env::Args` est le type d'itérateur renvoyé par la fonction de bibliothèque standard que nous avons utilisée au [chapitre 2](#) pour accéder aux arguments de ligne de commande. Il produit des valeurs, donc le déclare.

```
std::env::args() String impl type Item = String;
```

Le code générique peut utiliser les types associés :

```
/// Loop over an iterator, storing the values in a new vector.
fn collect_into_vector<I: Iterator>(iter: I) -> Vec<I::Item> {
    let mut results = Vec::new();
    for value in iter {
        results.push(value);
    }
    results
}
```

À l'intérieur du corps de cette fonction, Rust déduit le type de pour nous, ce qui est agréable; mais nous devons énoncer le type de retour de , et le type associé est le seul moyen de le faire. (serait tout simplement faux : on prétendrait renvoyer un vecteur d'itérateurs

```
!) value collect_into_vector Item Vec<I>
```

L'exemple précédent n'est pas du code que vous écririez vous-même, car après avoir lu [le chapitre 15](#), vous saurez que les itérateurs ont déjà une méthode standard qui fait ceci: . Regardons donc un autre exemple avant de passer à autre chose: `iter.collect()`

```
/// Print out all the values produced by an iterator
fn dump<I>(iter: I)
    where I: Iterator
```

```

{
    for (index, value) in iter.enumerate() {
        println!("{}: {:?}", index, value); // error
    }
}

```

Cela fonctionne presque. Il n'y a qu'un seul problème : il se peut qu'il ne s'agisse pas d'un type imprimable. `value`

```

error: `<I as Iterator>::Item` doesn't implement `Debug`
|
8 |     println!("{}: {:?}", index, value); // error
|           ^^^^^^
|           `<I as Iterator>::Item` cannot be formatted
|           using `{:?}` because it doesn't implement `

= help: the trait `Debug` is not implemented for `<I as Iterator>::Item`
= note: required by `std::fmt::Debug::fmt`
help: consider further restricting the associated type
|
5 |     where I: Iterator, <I as Iterator>::Item: Debug
|           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

```

Le message d'erreur est légèrement obscurci par l'utilisation par Rust de la syntaxe , qui est une façon explicite mais verbeuse de dire . Il s'agit d'une syntaxe Rust valide, mais vous aurez rarement besoin d'écrire un type de cette façon. `<I as Iterator>::Item I::Item`

L'essentiel du message d'erreur est que pour compiler cette fonction générique, nous devons nous assurer qu'elle implémente le trait, le trait de mise en forme des valeurs avec . Comme le suggère le message d'erreur, nous pouvons le faire en plaçant une limite sur

```
:I::Item Debug {:?} I::Item
```

```

use std::fmt::Debug;

fn dump<I>(iter: I)
    where I: Iterator, I::Item: Debug
{
    ...
}

```

Ou, nous pourrions écrire, « doit être un itérateur sur les valeurs »

```
:I String
```

```

fn dump<I>(iter: I)
    where I: Iterator<Item=String>
{
    ...
}

```

`Iterator<Item=String>` est lui-même un trait. Si vous considérez comme l'ensemble de tous les types d'itérateurs, alors est un sous-ensemble de : l'ensemble des types d'itérateurs qui produisent s. Cette syntaxe peut être utilisée partout où le nom d'un trait peut être utilisé, y compris les types d'objet de trait

```
: Iterator Iterator<Item=String> Iterator String
```

```

fn dump(iter: &mut dyn Iterator<Item=String>) {
    for (index, s) in iter.enumerate() {
        println!("{}: {:?}", index, s);
    }
}

```

Les traits avec des types associés, comme `, sont compatibles avec les méthodes de trait, mais seulement si tous les types associés sont épelés, comme indiqué ici. Sinon, le type de pourrait être n'importe quoi, et encore une fois, Rust n'aurait aucun moyen de taper-vérifier ce code.`. `I`terator s

Nous avons montré beaucoup d'exemples impliquant des itérateurs. Il est difficile de ne pas le faire; ils sont de loin l'utilisation la plus importante des types associés. Mais les types associés sont généralement utiles chaque fois qu'un trait doit couvrir plus que de simples méthodes:

- Dans une bibliothèque de pool de threads, un trait, représentant une unité de travail, peut avoir un type associé. `Task` `Output`
- Un trait, représentant une façon de rechercher une chaîne, peut avoir un type associé, représentant toutes les informations recueillies en faisant correspondre le motif à la chaîne : `Pattern` `Match`

```

trait Pattern {
    type Match;

    fn search(&self, string: &str) -> Option<Self::Match>;
}

/// You can search a string for a particular character.
impl Pattern for char {

```

```

/// A "match" is just the location where the
/// character was found.
type Match = usize;

fn search(&self, string: &str) -> Option<usize> {
    ...
}

```

Si vous êtes familier avec les expressions régulières, il est facile de voir comment aurait un type plus élaboré, probablement une structure qui inclurait le début et la longueur de la correspondance, les emplacements où les groupes entre parenthèses correspondaient, etc. `impl Pattern for RegExp Match`

- Une bibliothèque permettant d'utiliser des bases de données relationnelles peut avoir un trait avec des types associés représentant des transactions, des curseurs, des instructions préparées, etc. `Database Connection`

Les types associés sont parfaits pour les cas où chaque implémentation a un type spécifique connexe: chaque type de produit un type particulier de ; chaque type de recherche un type particulier de . Cependant, comme nous le verrons, certaines relations entre les types ne sont pas comme ça. `Task Output Pattern Match`

Traits génériques (ou fonctionnement de la surcharge de l'opérateur)

La multiplication dans Rust utilise ce trait:

```

/// std::ops::Mul, the trait for types that support `*`.
pub trait Mul<RHS> {
    /// The resulting type after applying the `*` operator
    type Output;

    /// The method for the `*` operator
    fn mul(self, rhs: RHS) -> Self::Output;
}

```

`Mul` est un trait générique. Le paramètre type , , est l'abréviation de *right-hand side*. `RHS`

Le paramètre type ici signifie la même chose qu'il signifie sur une structure ou une fonction: est un trait générique, et ses instances , , etc., sont

toutes des traits différents, tout comme et sont des fonctions différentes et et sont des types

```
différents. Mul<f64> Mul<String> Mul<Size> min:::  
<i32> min:::<String> Vec<i32> Vec<String>
```

Un seul type, par exemple, peut implémenter les deux et , et bien d'autres.

Vous seriez alors en mesure de multiplier a par de nombreux autres types. Chaque implémentation aurait son propre type

```
associé. WindowSize Mul<f64> Mul<i32> WindowSize Output
```

Les traits génériques bénéficient d'une dispense spéciale en ce qui concerne la règle orpheline: vous pouvez implémenter un trait étranger pour un type étranger, à condition que l'un des paramètres de type du trait soit un type défini dans la caisse actuelle. Donc, si vous vous êtes défini, vous pouvez implémenter pour , même si vous n'avez défini ni l'un ni l'autre ou . Ces implementations peuvent même être génériques, telles que . Cela fonctionne parce qu'il n'y a aucun moyen qu'une autre caisse puisse définir quoi que ce soit, et donc aucun conflit entre les implementations pourrait survenir. (Nous avons introduit la règle orpheline dans [« Traits et types d'autres personnes »](#).) C'est ainsi que les caisses définissent les opérations arithmétiques sur les

```
vecteurs. WindowSize Mul<WindowSize> f64 Mul f64 impl<T>  
Mul<WindowSize> for Vec<T> Mul<WindowSize> nalgebra
```

Le trait montré précédemment manque un détail mineur. Le vrai trait ressemble à ceci: `Mul`

```
pub trait Mul<RHS=Self> {  
    ...  
}
```

La syntaxe signifie que la valeur par défaut est . Si j'écris , sans spécifier le paramètre de type de ', cela signifie . Dans une limite, si j'écris , cela signifie . `RHS=Self` `RHS` `Self` `impl` `Mul` `for` `Complex` `Mul` `impl` `Mul<Complex>` `for` `Complex` `where T: Mul` `where T: Mul<T>`

Dans Rust, l'expression est un raccourci pour . Donc, surcharger l'opérateur dans Rust est aussi simple que de mettre en œuvre le trait. Nous montrerons des exemples dans le chapitre suivant. `lhs * rhs` `Mul::mul(lhs, rhs)` `* Mul`

Trait impl

Comme vous pouvez l'imaginer, les combinaisons de nombreux types génériques peuvent devenir désordonnées. Par exemple, la combinaison de quelques itérateurs à l'aide de combinateurs de bibliothèque standard transforme rapidement votre type de retour en une horreur :

```
use std::iter;
use std::vec::IntoIter;
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) ->
    iter::Cycle<iter::Chain<IntoIter<u8>, IntoIter<u8>>> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

Nous pourrions facilement remplacer ce type de retour poilu par un objet trait :

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> Box<dyn Iterator<Item=u8>> {
    Box::new(v.into_iter().chain(u.into_iter()).cycle())
}
```

Cependant, prendre la surcharge de l'expédition dynamique et une allocation de tas inévitable chaque fois que cette fonction est appelée juste pour éviter une signature de type laide ne semble pas être un bon échange, dans la plupart des cas.

Rust a une fonctionnalité appelée conçue précisément pour cette situation. nous permet d'« effacer » le type d'une valeur de retour, en spécifiant uniquement le ou les traits qu'elle implémente, sans envoi dynamique ni allocation de tas : `impl Trait`

```
fn cyclical_zip(v: Vec<u8>, u: Vec<u8>) -> impl Iterator<Item=u8> {
    v.into_iter().chain(u.into_iter()).cycle()
}
```

Maintenant, plutôt que de spécifier un type particulier imbriqué de structs de combinateur d'itérateur, la signature de 's indique simplement qu'elle renvoie une sorte d'itérateur sur . Le type de retour exprime l'intention de la fonction, plutôt que ses détails d'implémentation. `cyclical_zip` `u8`

Cela a définitivement nettoyé le code et l'a rendu plus lisible, mais c'est plus qu'un simple raccourci pratique. L'utilisation signifie que vous pouvez modifier le type réel renvoyé à l'avenir tant qu'il implémente toujours , et tout code appelant la fonction continuera à compiler sans problème. Cela offre beaucoup de flexibilité aux auteurs de bibliothèques, car

seules les fonctionnalités pertinentes sont codées dans la signature de type. `impl Trait impl Trait Iterator<Item=u8>`

Par exemple, si la première version d'une bibliothèque utilise des combinateurs itérateurs comme dans le précédent, mais qu'un meilleur algorithme pour le même processus est découvert, l'auteur de la bibliothèque peut utiliser différents combinateurs ou même créer un type personnalisé qui implémente , et les utilisateurs de la bibliothèque peuvent obtenir les améliorations de performances sans modifier du tout leur code. `Iterator`

Il peut être tentant d'utiliser pour approximer une version distribuée statiquement du modèle d'usine couramment utilisé dans les langages orientés objet. Par exemple, vous pouvez définir un trait comme celui-ci : `impl Trait`

```
trait Shape {  
    fn new() -> Self;  
    fn area(&self) -> f64;  
}
```

Après l'avoir implémenté pour quelques types, vous pouvez utiliser différents s en fonction d'une valeur d'exécution, comme une chaîne qu'un utilisateur entre. Cela ne fonctionne pas avec comme type de retour : `Shape impl Shape`

```
fn make_shape(shape: &str) -> impl Shape {  
    match shape {  
        "circle" => Circle::new(),  
        "triangle" => Triangle::new(), // error: incompatible types  
        "shape" => Rectangle::new(),  
    }  
}
```

Du point de vue de l'appelant, une fonction comme celle-ci n'a pas beaucoup de sens. est une forme d'envoi statique, de sorte que le compilateur doit connaître le type renvoyé par la fonction au moment de la compilation afin d'allouer la bonne quantité d'espace sur la pile et d'accéder correctement aux champs et aux méthodes sur ce type. Ici, il pourrait s'agir de , , ou , qui pourraient tous occuper différentes quantités d'espace et tous avoir des implémentations différentes de . `impl Trait Circle Triangle Rectangle area()`

Il est important de noter que Rust n'autorise pas les méthodes de trait à utiliser des valeurs de retour. Pour ce faire, il faudra apporter quelques améliorations au système de type des langues. Jusqu'à ce que ce travail soit terminé, seules les fonctions libres et les fonctions associées à des types spécifiques peuvent utiliser des retours. `impl Trait impl Trait`

`impl Trait` peut également être utilisé dans des fonctions qui prennent des arguments génériques. Par exemple, considérez cette fonction générique simple:

```
fn print<T: Display>(val: T) {  
    println!("{}", val);  
}
```

Elle est identique à cette version en utilisant : `impl Trait`

```
fn print(val: impl Display) {  
    println!("{}", val);  
}
```

Il y a une exception importante. L'utilisation de génériques permet aux appelants de la fonction de spécifier le type des arguments génériques, comme , alors que l'utilisation ne le fait pas. `print::<i32>(42) impl Trait`

Chaque argument se voit attribuer son propre paramètre de type anonyme, de sorte que pour les arguments est limité aux seules fonctions génériques les plus simples, sans relations entre les types d'arguments. `impl Trait impl Trait`

Consts associés

Comme les structs et les enums, les traits peuvent avoir des constantes associées. Vous pouvez déclarer un trait avec une constante associée en utilisant la même syntaxe que pour une struct ou un enum :

```
trait Greet {  
    const GREETING: &'static str = "Hello";  
    fn greet(&self) -> String;  
}
```

Les consts associés dans les traits ont un pouvoir spécial, cependant.

Comme les types et fonctions associés, vous pouvez les déclarer mais ne pas leur donner de valeur :

```

trait Float {
    const ZERO: Self;
    const ONE: Self;
}

```

Ensute, les implémenteurs du trait peuvent définir ces valeurs :

```

impl Float for f32 {
    const ZERO: f32 = 0.0;
    const ONE: f32 = 1.0;
}

impl Float for f64 {
    const ZERO: f64 = 0.0;
    const ONE: f64 = 1.0;
}

```

Cela vous permet d'écrire du code générique qui utilise les valeurs suivantes :

```

fn add_one<T: Float + Add<Output=T>>(value: T) -> T {
    value + T::ONE
}

```

Notez que les constantes associées ne peuvent pas être utilisées avec des objets trait, car le compilateur s'appuie sur des informations de type sur l'implémentation afin de choisir la bonne valeur au moment de la compilation.

Même un trait simple sans comportement du tout, comme , peut donner suffisamment d'informations sur un type, en combinaison avec quelques opérateurs, pour implémenter des fonctions mathématiques communes comme Fibonacci: Float

```

fn fib<T: Float + Add<Output=T>>(n: usize) -> T {
    match n {
        0 => T::ZERO,
        1 => T::ONE,
        n => fib::<T>(n - 1) + fib::<T>(n - 2)
    }
}

```

Dans les deux dernières sections, nous avons montré différentes façons dont les traits peuvent décrire les relations entre les types. Tous ces élé-

ments peuvent également être considérés comme des moyens d'éviter les frais généraux et les downcasts de méthodes virtuelles, car ils permettent à Rust de connaître des types plus concrets au moment de la compilation.

Limites de rétro-ingénierie

L'écriture de code générique peut être un véritable slog lorsqu'il n'y a pas de trait unique qui fait tout ce dont vous avez besoin. Supposons que nous ayons écrit cette fonction non générique pour faire un calcul:

```
fn dot(v1: &[i64], v2: &[i64]) -> i64 {
    let mut total = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Maintenant, nous voulons utiliser le même code avec des valeurs à virgule flottante. Nous pourrions essayer quelque chose comme ceci:

```
fn dot<N>(v1: &[N], v2: &[N]) -> N {
    let mut total: N = 0;
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Pas de chance: Rust se plaint de l'utilisation et du type de . Nous pouvons exiger d'être un type qui soutient et utilise les traits et. Notre utilisation de doit changer, cependant, parce que c'est toujours un entier dans Rust; la valeur en virgule flottante correspondante est . Heureusement, il existe un trait standard pour les types qui ont des valeurs par défaut. Pour les types numériques, la valeur par défaut est toujours 0

```
: * 0 N + * Add Mul 0 0 0.0 Default
```

```
use std::ops::{Add, Mul};

fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
}
```

```
total
}
```

C'est plus proche, mais cela ne fonctionne toujours pas tout à fait:

```
error: mismatched types
|
5 | fn dot<N: Add + Mul + Default>(v1: &[N], v2: &[N]) -> N {
|     - this type parameter
...
8 |         total = total + v1[i] * v2[i];
|                         ^^^^^^^^^^^^^^ expected type parameter `N`,
|                                         found associated type
|
|= note: expected type parameter `N`
        found associated type `<N as Mul>::Output`
help: consider further restricting this bound
|
5 | fn dot<N: Add + Mul + Default + Mul<Output = N>>(v1: &[N], v2: &[N])
|                         ^^^^^^^^^^^^^^^^^^
```

Notre nouveau code suppose que la multiplication de deux valeurs de type produit une autre valeur de type . Ce n'est pas nécessairement le cas. Vous pouvez surcharger l'opérateur de multiplication pour renvoyer le type de votre choix. Nous devons en quelque sorte dire à Rust que cette fonction générique ne fonctionne qu'avec des types qui ont la saveur normale de la multiplication, où la multiplication renvoie un . La suggestion dans le message d'erreur est *presque* juste: nous pouvons le faire en remplaçant par , et la même chose pour : N N N *

N N Mul Mul<Output=N> Add

```
fn dot<N: Add<Output=N> + Mul<Output=N> + Default>(v1: &[N], v2: &[N]) -
{
    ...
}
```

À ce stade, les limites commencent à s'accumuler, ce qui rend le code difficile à lire. Déplaçons les limites dans une clause : where

```
fn dot<N>(v1: &[N], v2: &[N]) -> N
where N: Add<Output=N> + Mul<Output=N> + Default
{
    ...
}
```

Génial. Mais Rust se plaint toujours de cette ligne de code:

```
error: cannot move out of type `<[N]>`, a non-copy slice
|
8 |         total = total + v1[i] * v2[i];
|             ^^^^^^
|             |
|             cannot move out of here
|             move occurs because `v1[_]` has type `<N>`,
|             which does not implement the `Copy` trait
```

Comme nous n'avons pas besoin d'être un type copiable, Rust interprète comme une tentative de déplacer une valeur hors de la tranche, ce qui est interdit. Mais nous ne voulons pas du tout modifier la tranche; nous voulons simplement copier les valeurs pour les exploiter. Heureusement, tous les types numériques intégrés de Rust implémentent , nous pouvons donc simplement ajouter cela à nos contraintes sur N: N v1[i] Copy

```
where N: Add<Output=N> + Mul<Output=N> + Default + Copy
```

Avec cela, le code se compile et s'exécute. Le code final ressemble à ceci :

```
use std::ops::{Add, Mul};

fn dot<N>(v1: &[N], v2: &[N]) -> N
    where N: Add<Output=N> + Mul<Output=N> + Default + Copy
{
    let mut total = N::default();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}

#[test]
fn test_dot() {
    assert_eq!(dot(&[1, 2, 3, 4], &[1, 1, 1, 1]), 10);
    assert_eq!(dot(&[53.0, 7.0], &[1.0, 5.0]), 88.0);
}
```

Cela arrive parfois dans Rust: il y a une période de disputes intenses avec le compilateur, à la fin de laquelle le code a l'air plutôt agréable, comme s'il avait été un jeu d'enfant à écrire, et fonctionne à merveille.

Ce que nous avons fait ici, c'est procéder à l'ingénierie inverse des limites sur , en utilisant le compilateur pour guider et vérifier notre travail. La raison pour laquelle c'était un peu pénible est qu'il n'y avait pas un seul trait dans la bibliothèque standard qui incluait tous les opérateurs et méthodes que nous voulions utiliser. En l'occurrence, il existe une caisse open source populaire appelée qui définit un tel trait! Si nous l'avions su, nous aurions pu ajouter à notre *Cargo.toml* et écrire: N Number num num

```
use num::Num;

fn dot<N: Num + Copy>(v1: &[N], v2: &[N]) -> N {
    let mut total = N::zero();
    for i in 0 .. v1.len() {
        total = total + v1[i] * v2[i];
    }
    total
}
```

Tout comme dans la programmation orientée objet, la bonne interface rend tout agréable, dans la programmation générique, le bon trait rend tout agréable.

Pourtant, pourquoi se donner tant de mal ? Pourquoi les concepteurs de Rust n'ont-ils pas rendu les génériques plus semblables à des modèles C++, où les contraintes sont laissées implicites dans le code, à la « duck typing »?

L'un des avantages de l'approche de Rust est la compatibilité directe du code générique. Vous pouvez modifier l'implémentation d'une fonction ou d'une méthode générique publique, et si vous n'avez pas modifié la signature, vous n'avez cassé aucun de ses utilisateurs.

Un autre avantage des limites est que lorsque vous obtenez une erreur de compilateur, au moins le compilateur peut vous dire où se trouve le problème. Les messages d'erreur du compilateur C++ impliquant des modèles peuvent être beaucoup plus longs que ceux de Rust, pointant vers de nombreuses lignes de code différentes, car le compilateur n'a aucun moyen de dire qui est à blâmer pour un problème : le modèle, ou son appelant, qui peut également être un modèle, ou l'appelant de ce modèle...

Peut-être que l'avantage le plus important d'écrire explicitement les limites est simplement qu'elles sont là, dans le code et dans la documentation. Vous pouvez regarder la signature d'une fonction générique dans Rust et voir exactement quel type d'arguments elle accepte. On ne peut

pas en dire autant des modèles. Le travail de documentation complète des types d'arguments dans les bibliothèques C++ comme Boost est encore *plus* ardu que ce que nous avons vécu ici. Les développeurs Boost n'ont pas de compilateur qui vérifie leur travail.

Traits en tant que fondation

Les traits sont l'une des principales caractéristiques d'organisation de Rust, et avec raison. Il n'y a rien de mieux pour concevoir un programme ou une bibliothèque qu'une bonne interface.

Ce chapitre était un blizzard de syntaxe, de règles et d'explications. Maintenant que nous avons jeté les bases, nous pouvons commencer à parler des nombreuses façons dont les traits et les génériques sont utilisés dans le code Rust. Le fait est que nous n'avons fait que commencer à gratter la surface. Les deux chapitres suivants couvrent les traits communs fournis par la bibliothèque standard. Les chapitres à venir couvrent les fermetures, les itérateurs, les entrées/sorties et la simultanéité. Les traits et les génériques jouent un rôle central dans tous ces sujets.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 12. Surcharge de l'opérateur

Dans le traceur d'ensemble de Mandelbrot que nous avons montré au [chapitre 2](#), nous avons utilisé le type de caisse pour représenter un nombre sur le plan complexe : num Complex

```
#[derive(Clone, Copy, Debug)]  
struct Complex<T> {  
    /// Real portion of the complex number  
    re: T,  
  
    /// Imaginary portion of the complex number  
    im: T,  
}
```

Nous avons pu ajouter et multiplier des nombres comme n'importe quel type numérique intégré, en utilisant rust et les opérateurs: Complex + *

```
z = z * z + c;
```

Vous pouvez également faire en sorte que vos propres types prennent en charge l'arithmétique et d'autres opérateurs, simplement en implémentant quelques traits intégrés. *C'est ce qu'on appelle la surcharge d'opérateur*, et l'effet est un peu similaire à la surcharge d'opérateur en C++, C#, Python et Ruby.

Les caractéristiques de surcharge de l'opérateur se répartissent en quelques catégories en fonction de la partie de la langue qu'ils prennent en charge, comme le montre [le tableau 12-1](#). Dans ce chapitre, nous couvrirons chaque catégorie. Notre objectif n'est pas seulement de vous aider à bien intégrer vos propres types dans le langage, mais aussi de vous donner une meilleure idée de la façon d'écrire des fonctions génériques comme la fonction de produit à points décrite dans [« Reverse-Engineering Bounds »](#) qui fonctionnent sur les types les plus naturellement utilisés via ces opérateurs. Le chapitre devrait également donner un aperçu de la façon dont certaines fonctionnalités du langage lui-même sont implémentées.

Tableau 12-1. Résumé des caractéristiques de la surcharge de l'opérateur

Catégorie	Trait	Opérateur
Opérateurs unaires	<code>std::ops::Ne</code> <code>g</code>	<code>-x</code>
	<code>std::ops::No</code> <code>t</code>	<code>!x</code>
Opérateurs arithmétiques	<code>std::ops::Ad</code> <code>d</code>	<code>x + y</code>
	<code>std::ops::Su</code> <code>b</code>	<code>x - y</code>
	<code>std::ops::Mu</code> <code>l</code>	<code>x * y</code>
	<code>std::ops::Di</code> <code>v</code>	<code>x / y</code>
	<code>std::ops::Re</code> <code>m</code>	<code>x % y</code>
Opérateurs binaires	<code>std::ops::Bi</code> <code>tAnd</code>	<code>x & y</code>
	<code>std::ops::Bi</code> <code>tOr</code>	<code>x y</code>
	<code>std::ops::Bi</code> <code>tXor</code>	<code>x ^ y</code>
	<code>std::ops::Sh</code> <code>l</code>	<code>x << y</code>
	<code>std::ops::Sh</code> <code>r</code>	<code>x >> y</code>

Catégorie	Trait	Opérateur
Opérateurs arithmétiques d'affectation composée	std::ops::Add Assign std::ops::Sub Assign std::ops::Mul Assign	x += y x -= y x *= y
		std::ops::Div Assign
		std::ops::Rem Assign
Opérateurs binaires d'affectation composée	std::ops::Bit AndAssign std::ops::Bit OrAssign std::ops::Bit XorAssign	x &= y x = y x ^= y
		std::ops::Shl Assign
		std::ops::Shr Assign
Comparaison	std::cmp::PartialEq	x == y, x != y
	std::cmp::PartialOrd	x < y, , , x <= y x > y x >= y
Indexation	std::ops::Index	x[y], &x[y]

Catégorie	Trait	Opérateur
	std::ops::Ind exMut	x[y] = z, &mut x [y]

Opérateurs arithmétiques et binaires

Dans Rust, l'expression est en fait un raccourci pour , un appel à la méthode du trait de la bibliothèque standard. Les types numériques standard de Rust implémentent tous . Pour que l'expression fonctionne pour les valeurs, la caisse implémente également ce trait. Des traits similaires couvrent les autres opérateurs: est un raccourci pour , une méthode du trait, couvre l'opérateur de négation du préfixe, etc. a +

```
b a.add(b) add std::ops::Add std::ops::Add a +
b Complex num Complex a *
b a.mul(b) std::ops::Mul std::ops::Neg -
```

Si vous voulez essayer d'écrire, vous devrez mettre le trait dans la portée afin que sa méthode soit visible. Cela fait, vous pouvez traiter toute l'arithmétique comme des appels de fonction: z.add(c) Add ¹

```
use std::ops::Add;

assert_eq!(4.125f32.add(5.75), 9.875);
assert_eq!(10.add(20), 10 + 20);
```

Voici la définition de : std::ops::Add

```
trait Add<Rhs = Self> {
    type Output;
    fn add(self, rhs: Rhs) -> Self::Output;
}
```

En d'autres termes, le trait est la capacité d'ajouter une valeur à vous-même. Par exemple, si vous souhaitez pouvoir ajouter des valeurs à votre type, votre type doit implémenter à la fois et . Le paramètre de type du trait est défini par défaut sur , donc si vous implémentez l'addition entre deux valeurs du même type, vous pouvez simplement écrire pour ce cas. Le type associé décrit le résultat de

l'ajout. Add<T> T i32 u32 Add<i32> Add<u32> Rhs Self Add Output

Par exemple, pour pouvoir additionner des valeurs, il faut implémenter .

Puisque nous ajoutons un type à lui-même, nous écrivons simplement

```
:Complex<i32> Complex<i32> Add<Complex<i32>> Add
```

```
use std::ops::Add;

impl Add for Complex<i32> {
    type Output = Complex<i32>;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

Bien sûr, nous ne devrions pas avoir à implémenter séparément pour ,,, et ainsi de suite. Toutes les définitions seraient exactement les mêmes, à l'exception des types impliqués, nous devrions donc être en mesure d'écrire une seule implémentation générique qui les couvre toutes, tant que le type des composants complexes eux-mêmes prend en charge l'ajout: Add Complex<i32> Complex<f32> Complex<f64>

```
use std::ops::Add;

impl<T> Add for Complex<T>
where
    T: Add<Output = T>,
{
    type Output = Self;
    fn add(self, rhs: Self) -> Self {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

En écrivant , nous nous limitons aux types qui peuvent être ajoutés à eux-mêmes, donnant une autre valeur. C'est une restriction raisonnable, mais nous pourrions assouplir encore les choses : le trait n'exige pas que les deux opérandes aient le même type, ni ne limite le type de résultat. Ainsi, une implémentation générique maximale permettrait aux opérandes gauche et droit de varier indépendamment et de produire une valeur de

n'importe quel type de composant produit par l'addition : where T:

Add<Output=T> T T Add + Complex

```
use std::ops::Add;

impl<L, R> Add<Complex<R>> for Complex<L>
where
    L: Add<R>,
{
    type Output = Complex<L::Output>;
    fn add(self, rhs: Complex<R>) -> Self::Output {
        Complex {
            re: self.re + rhs.re,
            im: self.im + rhs.im,
        }
    }
}
```

Dans la pratique, cependant, Rust a tendance à éviter de prendre en charge des opérations de type mixte. Puisque notre paramètre de type doit implémenter , il s'ensuit généralement que et va être le même type: il n'y a tout simplement pas beaucoup de types disponibles pour cette implémentation autre chose. Donc, en fin de compte, cette version générique maximale peut ne pas être beaucoup plus utile que la définition générique précédente, plus simple. L Add<R> L R L

Les traits intégrés de Rust pour les opérateurs arithmétiques et binaires se répartissent en trois groupes : les opérateurs unaires, les opérateurs binaires et les opérateurs d'affectation composée. Au sein de chaque groupe, les traits et leurs méthodes ont tous la même forme, nous allons donc couvrir un exemple de chacun.

Opérateurs unaires

Mis à part l'opérateur de déréférencement , que nous couvrirons séparément dans [« Deref et DerefMut »](#), Rust dispose de deux opérateurs unaires que vous pouvez personnaliser, illustrés dans [le tableau 12-2](#). *

Tableau 12-2. Caractéristiques intégrées pour les opérateurs unaires

Nom du trait	Expression	Expression équivalente
--------------	------------	------------------------

std::ops::Neg	-x	x.neg()
---------------	----	---------

std::ops::Not	!x	x.not()
---------------	----	---------

Tous les types numériques signés de Rust implémentent , pour l'opérateur de négation unaire ; les types entiers et implémenter , pour l'opérateur de complément unaire . Il existe également des implementations pour les références à ces types. `std::ops::Neg` – `bool std::ops::Not` !

Notez que complète les valeurs et effectue un complément binaire (c'est-à-dire inverse les bits) lorsqu'il est appliqué à des entiers; il joue le rôle à la fois des opérateurs et des opérateurs de C et C++. ! `bool ! ~`

Les définitions de ces traits sont simples :

```
trait Neg {
    type Output;
    fn neg(self) -> Self::Output;
}

trait Not {
    type Output;
    fn not(self) -> Self::Output;
}
```

La négation d'un nombre complexe annule simplement chacun de ses composants. Voici comment nous pourrions écrire une implémentation générique de la négation pour les valeurs : `Complex`

```
use std::ops::Neg;

impl<T> Neg for Complex<T>
where
    T: Neg<Output = T>,
{
    type Output = Complex<T>;
    fn neg(self) -> Complex<T> {
        Complex {
            re: -self.re,
            im: -self.im,
        }
    }
}
```

Opérateurs binaires

Les opérateurs arithmétiques binaires et binaires de Rust et leurs caractéristiques intégrées correspondantes apparaissent dans [le tableau 12-3](#).

Tableau 12-3. Caractéristiques intégrées pour les opérateurs binaires

Catégorie	Nom du trait	Expression	Expression équivalente
Opérateurs arithmétiques	<code>std::ops::A</code> dd	<code>x + y</code>	<code>x.add(y)</code>
	<code>std::ops::S</code> ub	<code>x - y</code>	<code>x.sub(y)</code>
	<code>std::ops::M</code> ul	<code>x * y</code>	<code>x.mul(y)</code>
	<code>std::ops::D</code> iv	<code>x / y</code>	<code>x.div(y)</code>
	<code>std::ops::R</code> em	<code>x % y</code>	<code>x.rem(y)</code>
Opérateurs binaires	<code>std::ops::B</code> itAnd	<code>x & y</code>	<code>x.bitand(y)</code>
	<code>std::ops::B</code> itOr	<code>x y</code>	<code>x.bitor(y)</code>
	<code>std::ops::B</code> itXor	<code>x ^ y</code>	<code>x.bitxor(y)</code>
	<code>std::ops::S</code> hl	<code>x << y</code>	<code>x.shl(y)</code>
	<code>std::ops::S</code> hr	<code>x >> y</code>	<code>x.shr(y)</code>

Tous les types numériques de Rust implémentent les opérateurs arithmétiques. Les types entiers de Rust et implémentent les opérateurs binaires. Il existe également des implementations qui acceptent les références à ces types en tant qu'opérandes ou les deux. `bool`

Tous les traits ici ont la même forme générale. La définition de , pour l'opérateur, ressemble à ceci : `std::ops::BitXor ^`

```

trait BitXor<Rhs = Self> {
    type Output;
    fn bitxor(self, rhs: Rhs) -> Self::Output;
}

```

Au début de ce chapitre, nous avons également montré , un autre trait dans cette catégorie, ainsi que plusieurs exemples d'implémentations. `std::ops::Add`

Vous pouvez utiliser l'opérateur pour concaténer a avec une tranche ou un autre . Cependant, Rust ne permet pas à l'opérande gauche d'être un , pour décourager la construction de longues cordes en concaténant à plusieurs reprises de petits morceaux sur la gauche. (Cela fonctionne mal, nécessitant un temps quadratique dans la longueur finale de la chaîne.) Généralement, la macro est meilleure pour construire des cordes pièce par pièce; nous montrons comment faire cela dans [« Ajout et insertion de texte »](#). + String &str String + &str write!

Opérateurs d'affectation composée

Une expression d'affectation composée est une expression semblable ou : elle prend deux opérandes, effectue une opération sur eux comme l'addition ou un AND binaire, et stocke le résultat dans l'opérande gauche. Dans Rust, la valeur d'une expression d'affectation composée est toujours , jamais la valeur stockée. `x += y` `x &= y ()`

De nombreuses langues ont des opérateurs comme ceux-ci et les définissent généralement comme un raccourci pour des expressions comme ou . Cependant, Rust n'adopte pas cette approche. Au lieu de cela, est un raccourci pour l'appel de méthode , où est la seule méthode du trait: `x = x + y` `x = x & y` `x += y` `x.add_assign(y)` `add_assign` `std::ops::AddAssign`

```

trait AddAssign<Rhs = Self> {
    fn add_assign(&mut self, rhs: Rhs);
}

```

[Le tableau 12-4](#) montre tous les opérateurs d'affectation de composés de Rust et les caractéristiques intégrées qui les implémentent.

Tableau 12-4. Caractéristiques intégrées pour les opérateurs d'affectation de composés

Catégorie	Nom du trait	Expression	Expression équivalente
Opérateurs arithmétiques	<code>std::ops::AddAssign</code>	<code>x += y</code>	<code>x.add_assign(y)</code>
	<code>std::ops::SubAssign</code>	<code>x -= y</code>	<code>x.subtract_assign(y)</code>
	<code>std::ops::MulAssign</code>	<code>x *= y</code>	<code>x.multiply_assign(y)</code>
	<code>std::ops::DivAssign</code>	<code>x /= y</code>	<code>x.divide_assign(y)</code>
	<code>std::ops::RemAssign</code>	<code>x %= y</code>	<code>x.remainder_assign(y)</code>
	<code>std::ops::BitAndAssign</code>	<code>x &= y</code>	<code>x.bitand_assign(y)</code>
	<code>std::ops::BitOrAssign</code>	<code>x = y</code>	<code>x.bitor_assign(y)</code>
	<code>std::ops::BitXorAssign</code>	<code>x ^= y</code>	<code>x.bitxor_assign(y)</code>
	<code>std::ops::ShlAssign</code>	<code>x <= y</code>	<code>x.shl_assign(y)</code>
	<code>std::ops::ShrAssign</code>	<code>x >= y</code>	<code>x.shr_assign(y)</code>

Tous les types numériques de Rust implémentent les opérateurs d'affectation composée arithmétique. Les types entiers de Rust et implémentent les opérateurs d'affectation composée binaire. `bool`

Une implémentation générique de pour notre type est simple: `AddAssign Complex`

```

use std::ops::AddAssign;

impl<T> AddAssign for Complex<T>
where
    T: AddAssign<T>,
{
    fn add_assign(&mut self, rhs: Complex<T>) {
        self.re += rhs.re;
        self.im += rhs.im;
    }
}

```

Le trait intégré pour un opérateur d'affectation de composé est complètement indépendant du trait intégré pour l'opérateur binaire correspondant. La mise en œuvre n'implémente pas automatiquement ; si vous voulez que Rust autorise votre type comme opérande gauche d'un opérateur, vous devez l'implémenter vous-même.

```
std::ops::Add std::ops::AddAssign += AddAssign
```

Comparaisons d'équivalence

Les opérateurs d'égalité de Rust, et , sont des raccourcis pour les appels aux traits et aux méthodes:

```
== != std::cmp::PartialEq eq ne
```

```

assert_eq!(x == y, x.eq(&y));
assert_eq!(x != y, x.ne(&y));

```

Voici la définition de :

```
std::cmp::PartialEq
```

```

trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    fn eq(&self, other: &Rhs) -> bool;
    fn ne(&self, other: &Rhs) -> bool {
        !self.eq(other)
    }
}

```

Étant donné que la méthode a une définition par défaut, il vous suffit de définir pour implémenter le trait, voici donc une implémentation complète pour :

```
ne eq PartialEq Complex
```

```

impl<T: PartialEq> PartialEq for Complex<T> {
    fn eq(&self, other: &Complex<T>) -> bool {
        self.re == other.re && self.im == other.im
    }
}

```

En d'autres termes, pour tout type de composant qui peut lui-même être comparé pour l'égalité, cela implémente la comparaison pour . En supposant que nous ayons également implémenté quelque part le long de la ligne, nous pouvons maintenant

écrire: T Complex<T> std::ops::Mul Complex

```

let x = Complex { re: 5, im: 2 };
let y = Complex { re: 2, im: 5 };
assert_eq!(x * y, Complex { re: 0, im: 29 });

```

Les implémentations de sont presque toujours de la forme montrée ici: elles comparent chaque champ de l'opérande gauche au champ correspondant de droite. Ceux-ci deviennent fastidieux à écrire, et l'égalité est une opération courante à prendre en charge, donc si vous demandez, Rust générera automatiquement une implémentation de pour vous.

Ajoutez simplement à l'attribut de la définition de type comme suit

```
:PartialEq PartialEq PartialEq derive
```

```

#[derive(Clone, Copy, Debug, PartialEq)]
struct Complex<T> {
    ...
}

```

L'implémentation générée automatiquement par Rust est essentiellement identique à notre code manuscrit, comparant chaque champ ou élément du type à tour de rôle. Rust peut également dériver des implémentations pour les types. Naturellement, chacune des valeurs que le type détient (ou pourrait contenir, dans le cas d'un) doit elle-même implémenter

```
.PartialEq enum enum PartialEq
```

Contrairement aux traits arithmétiques et binaires, qui prennent leurs opérandes par valeur, prennent ses opérandes par référence. Cela signifie que la comparaison de valeurs non telles que s, s ou s ne les déplace pas, ce qui serait gênant: PartialEq Copy String Vec HashMap

```

let s = "d\x6fv\x65t\x61i\x6c".to_string();
let t = "\x64o\x76e\x74a\x691".to_string();
assert!(s == t); // s and t are only borrowed...

```

```
// ... so they still have their values here.
assert_eq!(format!("{} {}", s, t), "dovetail dovetail");
```

Cela nous amène à la liaison du trait sur le paramètre type, qui est d'un type que nous n'avons jamais vu auparavant: Rhs

```
where
Rhs: ?Sized,
```

Cela assouplit l'exigence habituelle de Rust selon laquelle les paramètres de type doivent être des types de taille, ce qui nous permet d'écrire des traits comme ou . Les méthodes et prennent des paramètres de type , et comparer quelque chose avec a ou a est tout à fait raisonnable. Puisque implémente , les assertions suivantes sont équivalentes

```
:PartialEq<str> PartialEq<[T]> eq ne &Rhs &str &
[T] str PartialEq<str>

assert!("ungula" != "ungulate");
assert!("ungula".ne("ungulate"));
```

Ici, les deux et seraient le type non dimensionné , faisant de 's et paramètres les deux valeurs. Nous discuterons des types de taille, des types non dimensionnés et du trait en détail dans

« Taille ». Self Rhs str ne self rhs &str Sized

Pourquoi ce trait s'appelle-t-il ? La définition mathématique traditionnelle d'une *relation d'équivalence*, dont l'égalité est un exemple, impose trois exigences. Pour toutes les valeurs et : `PartialEq` x y

- Si est vrai, alors doit être vrai aussi. En d'autres termes, l'échange des deux côtés d'une comparaison d'égalité n'affecte pas le résultat. $x == y$ $y == x$
- Si et , alors il doit être le cas que . Étant donné toute chaîne de valeurs, chacune égale à la suivante, chaque valeur de la chaîne est directement égale à toutes les autres. L'égalité est contagieuse. $x == y$ $y == z$ $x == z$
- Il doit toujours être vrai que . $x == x$

Cette dernière exigence peut sembler trop évidente pour valoir la peine d'être énoncée, mais c'est exactement là que les choses tournent mal. Rust et sont des valeurs à virgule flottante standard IEEE. Selon cette norme, les expressions comme et d'autres sans valeur appropriée doivent produire des valeurs spéciales *non numériques*, généralement appelées

valeurs NaN. La norme exige en outre qu'une valeur NaN soit traitée comme inégale à toutes les autres valeurs, y compris elle-même. Par exemple, la norme requiert tous les comportements suivants

```
:f32 f64 0.0/0.0
```

```
assert!(f64::is_nan(0.0 / 0.0));
assert_eq!(0.0 / 0.0 == 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 != 0.0 / 0.0, true);
```

De plus, toute comparaison ordonnée avec une valeur NaN doit renvoyer false :

```
assert_eq!(0.0 / 0.0 < 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 > 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 <= 0.0 / 0.0, false);
assert_eq!(0.0 / 0.0 >= 0.0 / 0.0, false);
```

Ainsi, bien que l'opérateur de Rust réponde aux deux premières exigences en matière de relations d'équivalence, il ne répond clairement pas à la troisième lorsqu'il est utilisé sur des valeurs en virgule flottante IEEE. C'est ce qu'on appelle une *relation d'équivalence partielle*, de sorte que Rust utilise le nom du trait intégré de l'opérateur. Si vous écrivez du code générique avec des paramètres de type connus uniquement pour être , vous pouvez supposer que les deux premières exigences sont valables, mais vous ne devez pas supposer que les valeurs sont toujours égales à elles-mêmes. == PartialEq == PartialEq

Cela peut être un peu contre-intuitif et peut conduire à des bugs si vous n'êtes pas vigilant. Si vous préférez que votre code générique nécessite une relation d'équivalence complète, vous pouvez plutôt utiliser le trait comme une liaison, ce qui représente une relation d'équivalence complète : si un type implémente , alors doit être pour chaque valeur de ce type. Dans la pratique, presque tous les types de mise en œuvre devraient également être mis en œuvre; et sont les seuls types de la bibliothèque standard qui sont mais pas . std::cmp::Eq Eq x ==
x true x PartialEq Eq f32 f64 PartialEq Eq

La bibliothèque standard définit comme une extension de , n'ajoutant aucune nouvelle méthode : Eq PartialEq

```
trait Eq: PartialEq<Self> {}
```

Si votre type l'est et que vous souhaitez qu'il le soit également, vous devez explicitement implémenter , même si vous n'avez pas besoin de définir de nouvelles fonctions ou de nouveaux types pour le faire. La mise en œuvre pour notre type est donc rapide: `PartialEq Eq Eq Eq Complex`

```
impl<T: Eq> Eq for Complex<T> {}
```

Nous pourrions l'implémenter encore plus succinctement en incluant simplement dans l'attribut sur la définition du type: `Eq derive Complex`

```
#[derive(Clone, Copy, Debug, Eq, PartialEq)]
struct Complex<T> {
    ...
}
```

Les implémentations dérivées sur un type générique peuvent dépendre des paramètres de type. Avec l'attribut, implémenterait , car fait, mais n'implémenterait que , puisque n'implémente pas

```
.derive Complex<i32> Eq i32 Complex<f32> PartialEq f32 Eq
```

Lorsque vous vous implémentez vous-même, Rust ne peut pas vérifier que vos définitions pour les méthodes et se comportent réellement comme requis pour une équivalence partielle ou complète. Ils pourraient faire tout ce que vous voulez. Rust vous prend simplement sur parole que vous avez mis en œuvre l'égalité d'une manière qui répond aux attentes des utilisateurs du trait. `std::cmp::PartialEq eq ne`

Bien que la définition de fournisse une définition par défaut pour , vous pouvez fournir votre propre implémentation si vous le souhaitez. Cependant, vous devez vous assurer que et sont des compléments exacts les uns des autres. Les utilisateurs du trait supposeront qu'il en est ainsi. `PartialEq ne ne eq PartialEq`

Comparaisons ordonnées

Rust spécifie le comportement des opérateurs de comparaison ordonnés , , et le tout en termes d'un seul trait, : < >

```
<= >= std::cmp::PartialOrd
```

```
trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;
```

```

fn lt(&self, other: &Rhs) -> bool { ... }
fn le(&self, other: &Rhs) -> bool { ... }
fn gt(&self, other: &Rhs) -> bool { ... }
fn ge(&self, other: &Rhs) -> bool { ... }

}

```

Note qui s'étend : vous ne pouvez faire des comparaisons ordonnées que sur des types que vous pouvez également comparer pour l'égalité. `PartialOrd<Rhs>` `PartialEq<Rhs>`

La seule méthode que vous devez mettre en œuvre vous-même est .

Lorsque renvoie , indique alors la relation de ' à

```
:PartialOrd partial_cmp partial_cmp Some(o) o self other
```

```

enum Ordering {
    Less,          // self < other
    Equal,         // self == other
    Greater,       // self > other
}

```

Mais si les retours , cela signifie et sont non ordonnés les uns par rapport aux autres: ni l'un ni l'autre n'est plus grand que l'autre, ni égal. Parmi tous les types primitifs de Rust, seules les comparaisons entre les valeurs en virgule flottante reviennent : plus précisément, la comparaison d'une valeur NaN (pas un nombre) avec quoi que ce soit d'autre renvoie. Nous donnons plus de détails sur les valeurs NaN dans [« Comparaisons d'équivalence »](#). `partial_cmp None self other None None`

Comme les autres opérateurs binaires, pour comparer des valeurs de deux types et , doit implémenter . Les expressions telles que ou sont des raccourcis pour les appels à des méthodes, comme indiqué dans [le tableau 12-5](#). `Left Right Left PartialOrd<Right> x < y x >= y PartialOrd`

Tableau 12-5. Opérateurs et méthodes de comparaison ordonnés PartialOrd

Expression	Appel de méthode équivalent	Définition par défaut
<code>x < y</code>	<code>x.lt(y)</code>	<code>x.partial_cmp(&y) == Some(Less)</code>
<code>x > y</code>	<code>x.gt(y)</code>	<code>x.partial_cmp(&y) == Some(Greater)</code>
<code>x <= y</code>	<code>x.le(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Less Equal))</code>
<code>x >= y</code>	<code>x.ge(y)</code>	<code>matches!(x.partial_cmp(&y), Some(Greater Equal))</code>

Comme dans les exemples précédents, le code d'appel de méthode équivalent affiché suppose que et sont dans la portée. `std::cmp::PartialOrd std::cmp::Ordering`

Si vous savez que les valeurs de deux types sont toujours ordonnées l'une par rapport à l'autre, vous pouvez implémenter le trait le plus strict: `std::cmp::Ord`

```
trait Ord: Eq + PartialOrd<Self> {
    fn cmp(&self, other: &Self) -> Ordering;
}
```

La méthode ici renvoie simplement un , au lieu d'un like : déclare toujours ses arguments égaux ou indique leur ordre relatif. Presque tous les types qui implémentent devraient également implémenter . Dans la bibliothèque standard, et sont les seules exceptions à cette règle. `cmp Ordering Option<Ordering> partial_cmp cmp PartialOrd Ord f32 f64`

Comme il n'y a pas d'ordre naturel sur les nombres complexes, nous ne pouvons pas utiliser notre type des sections précédentes pour afficher un exemple d'implémentation de . Au lieu de cela, supposons que vous travaillez avec le type suivant, représentant l'ensemble des nombres se situant dans un intervalle semi-ouvert donné : `Complex PartialOrd`

```

#[derive(Debug, PartialEq)]
struct Interval<T> {
    lower: T, // inclusive
    upper: T, // exclusive
}

```

Vous souhaitez que les valeurs de ce type soient partiellement ordonnées : un intervalle est inférieur à un autre s'il tombe entièrement avant l'autre, sans chevauchement. Si deux intervalles inégaux se chevauchent, ils ne sont pas ordonnés : un élément de chaque côté est inférieur à un élément de l'autre. Et deux intervalles égaux sont tout simplement égaux.

La mise en œuvre suivante de ces règles : PartialOrd

```

use std::cmp::{Ordering, PartialOrd};

impl<T: PartialOrd> PartialOrd<Interval<T>> for Interval<T> {
    fn partial_cmp(&self, other: &Interval<T>) -> Option<Ordering> {
        if self == other {
            Some(Ordering::Equal)
        } else if self.lower >= other.upper {
            Some(Ordering::Greater)
        } else if self.upper <= other.lower {
            Some(Ordering::Less)
        } else {
            None
        }
    }
}

```

Une fois cette implémentation en place, vous pouvez écrire ce qui suit :

```

assert!(Interval { lower: 10, upper: 20 } < Interval { lower: 20, upper
assert!(Interval { lower: 7, upper: 8 } >= Interval { lower: 0, upper
assert!(Interval { lower: 7, upper: 8 } <= Interval { lower: 7, upper

// Overlapping intervals aren't ordered with respect to each other.
let left = Interval { lower: 10, upper: 30 };
let right = Interval { lower: 20, upper: 40 };
assert!(!(left < right));
assert!(!(left >= right));

```

Bien que ce soit ce que vous verrez habituellement, les ordres totaux définis avec sont nécessaires dans certains cas, tels que les méthodes de tri implémentées dans la bibliothèque standard. Par exemple, les intervalles de tri ne sont pas possibles avec seulement une implémentation. Si vous

voulez les trier, vous devrez combler les lacunes des cas non ordonnés.

Vous voudrez peut-être trier par limite supérieure, par exemple, et il est facile de le faire avec : `PartialOrd Ord PartialOrd sort_by_key`

```
intervals.sort_by_key(| i| i.upper);
```

Le type wrapper en tire parti en implémentant avec une méthode qui inverse simplement tout ordre. Pour tout type qui implémente , implémente aussi, mais avec un ordre inversé. Par exemple, le tri de nos intervalles de haut en bas par la limite inférieure est

simple: `Reverse Ord T Ord std::cmp::Reverse<T> Ord`

```
use std::cmp::Reverse;
intervals.sort_by_key(| i| Reverse(i.lower));
```

Index et IndexMut

Vous pouvez spécifier comment une expression d'indexation comme fonctionne sur votre type en implémentant les traits et. Les tableaux supportent directement l'opérateur, mais sur tout autre type, l'expression est normalement un raccourci pour , où est une méthode du trait. Cependant, si l'expression est attribuée ou empruntée de manière mutable, il s'agit plutôt d'un raccourci pour , un appel à la méthode du

```
trait a[i] std::ops::Index std::ops::IndexMut [] a[i] *a.ind
ex(i) index std::ops::Index *a.index_mut(i) std::ops::Index
Mut
```

Voici les définitions des traits :

```
trait Index<Idx> {
    type Output: ?Sized;
    fn index(&self, index: Idx) -> &Self::Output;
}

trait IndexMut<Idx>: Index<Idx> {
    fn index_mut(&mut self, index: Idx) -> &mut Self::Output;
}
```

Notez que ces traits prennent le type de l'expression d'index comme paramètre. Vous pouvez indexer une tranche avec un seul , faisant référence à un seul élément, car les tranches implémentent . Mais vous pouvez vous référer à une sous-section avec une expression comme parce

qu'ils implémentent également . Cette expression est un raccourci pour

```
:usize Index<usize> a[i..j] Index<Range<usize>>
```

```
*a.index(std::ops::Range { start: i, end: j })
```

Rust's et collections vous permettent d'utiliser n'importe quel type hashable ou ordonné comme index. Le code suivant fonctionne car implémente : HashMap BTreeMap HashMap<&str, i32> Index<&str>

```
use std::collections::HashMap;
let mut m = HashMap::new();
m.insert("十", 10);
m.insert("百", 100);
m.insert("千", 1000);
m.insert("万", 1_0000);
m.insert("億", 1_0000_0000);

assert_eq!(m["十"], 10);
assert_eq!(m["千"], 1000);
```

Ces expressions d'indexation sont équivalentes à :

```
use std::ops::Index;
assert_eq!(*m.index("十"), 10);
assert_eq!(*m.index("千"), 1000);
```

Le type associé au trait spécifie le type produit par une expression d'indexation : pour notre , le type de l'implémentation est

```
.Index Output HashMap Index Output i32
```

Le trait s'étend avec une méthode qui prend une référence mutable à , et renvoie une référence mutable à une valeur. Rust sélectionne automatiquement lorsque l'expression d'indexation se produit dans un contexte où cela est nécessaire. Par exemple, supposons que nous écrivions ce qui suit : IndexMut Index index_mut self Output index_mut

```
let mut desserts =
    vec![ "Howalon".to_string(), "Soan papdi".to_string()];
desserts[0].push_str(" (fictional)");
desserts[1].push_str(" (real)");
```

Étant donné que la méthode fonctionne sur , ces deux dernières lignes sont équivalentes à : push_str &mut self

```

use std::ops::IndexMut;
(*desserts.index_mut(0)).push_str(" (fictional)");
(*desserts.index_mut(1)).push_str(" (real)");

```

L'une des limites est que, de par sa conception, il doit renvoyer une référence modifiable à une certaine valeur. C'est pourquoi vous ne pouvez pas utiliser une expression comme insérer une valeur dans le : la table devrait créer une entrée pour d'abord, avec une valeur par défaut, et renvoyer une référence modifiable à cela. Mais tous les types n'ont pas de valeurs par défaut bon marché, et certains peuvent être coûteux à abandonner; ce serait un gaspillage de créer une telle valeur pour être immédiatement abandonné par la cession. (Il est prévu d'améliorer cela dans les versions ultérieures de la langue.) `IndexMut m["+ "+"] = 10; HashMap m "+"`

L'utilisation la plus courante de l'indexation concerne les collections. Par exemple, supposons que nous travaillions avec des images bitmap, comme celles que nous avons créées dans le traceur d'ensemble Mandelbrot au [chapitre 2](#). Rappelons que notre programme contenait du code comme celui-ci :

```

pixels[row * bounds.0 + column] = ...;

```

Il serait plus agréable d'avoir un type qui agit comme un tableau bidimensionnel, nous permettant d'accéder aux pixels sans avoir à écrire toute l'arithmétique: `Image<u8>`

```

image[row][column] = ...;

```

Pour ce faire, nous devrons déclarer une struct:

```

struct Image<P> {
    width: usize,
    pixels: Vec<P>,
}

impl<P: Default + Copy> Image<P> {
    // Create a new image of the given size.
    fn new(width: usize, height: usize) -> Image<P> {
        Image {
            width,
            pixels: vec![P::default(); width * height],
        }
    }
}

```

```
}
```

Et voici les implémentations de et cela correspondrait à la
facture: `Index` `IndexMut`

```
impl<P> std::ops::Index<usize> for Image<P> {
    type Output = [P];
    fn index(&self, row: usize) -> &[P] {
        let start = row * self.width;
        &self.pixels[start..start + self.width]
    }
}

impl<P> std::ops::IndexMut<usize> for Image<P> {
    fn index_mut(&mut self, row: usize) -> &mut [P] {
        let start = row * self.width;
        &mut self.pixels[start..start + self.width]
    }
}
```

Lorsque vous indexez dans un , vous récupérez une tranche de pixels ;
l'indexation de la tranche vous donne un pixel individuel. `Image`

Notez que lorsque nous écrivons , if est hors limites, notre méthode es-
saiera d'indexer hors de portée, déclenchant une panique. Voici comment
et les implémentations sont censées se comporter : l'accès hors limites est
détecté et provoque une panique, comme lorsque vous indexez un
tableau, une tranche ou un vecteur hors limites. `image[row]`
`[column] row .index() self.pixels` `Index` `IndexMut`

Autres opérateurs

Tous les opérateurs ne peuvent pas être surchargés dans Rust. À partir de
Rust 1.56, l'opérateur de vérification des erreurs ne fonctionne qu'avec
quelques autres types de bibliothèque standard, mais des travaux sont en
cours pour l'étendre également aux types définis par l'utilisateur. De
même, les opérateurs logiques et sont limités aux valeurs booléennes
uniquement. Les opérateurs et créent toujours une structure représen-
tant les limites de la plage, l'opérateur emprunte toujours des références
et l'opérateur déplace ou copie toujours des valeurs. Aucun d'entre eux
ne peut être surchargé. `? Result && | |= & =`

L'opérateur de déréférencement, , et l'opérateur de point pour accéder aux champs et aux méthodes d'appel, comme dans et , peuvent être surchargés à l'aide des traits Deref et DerefMut, qui sont couverts dans le chapitre suivant. (Nous ne les avons pas inclus ici parce que ces traits font plus que simplement surcharger quelques opérateurs.) *val val.field val.method()

Rust ne prend pas en charge la surcharge de l'opérateur d'appel de fonction, . Au lieu de cela, lorsque vous avez besoin d'une valeur appelable, vous écrivez généralement simplement une fermeture. Nous expliquerons comment cela fonctionne et couvrirons les , , et les traits spéciaux dans le chapitre 14. f(x) Fn FnMut FnOnce

- 1** Les programmeurs Lisp se réjouissent ! L'expression est l'opérateur sur , capturé en tant que valeur de fonction. <i32 as Add>::add + i32

Chapitre 13. Caractéristiques d'utilité

La science n'est rien d'autre que la recherche de l'unité dans la variété sauvage de la nature – ou, plus exactement, dans la variété de notre expérience. La poésie, la peinture, les arts sont la même recherche, selon l'expression de Coleridge, de l'unité dans la variété.

—Jacob Bronowski

Ce chapitre décrit ce que nous appelons les traits « utilitaires » de Rust, un sac à main de divers traits de la bibliothèque standard qui ont suffisamment d'impact sur la façon dont Rust est écrit pour que vous deviez les connaître afin d'écrire du code idiomatique et de concevoir des interfaces publiques pour vos caisses que les utilisateurs jugeront correctement « rustiques ». Ils se répartissent en trois grandes catégories :

Traits d'extension de la langue

Tout comme les traits de surcharge d'opérateur que nous avons couverts dans le chapitre précédent vous permettent d'utiliser les opérateurs d'expression de Rust sur vos propres types, il existe plusieurs autres traits de bibliothèque standard qui servent de points d'extension Rust, vous permettant d'intégrer vos propres types plus étroitement avec le langage. Ceux-ci incluent `,`, `et`, `,` et les traits de conversion `et`. Nous les décrirons dans ce

chapitre. `Drop` `Deref` `DerefMut` `From` `Into`

Traits de marqueur

Ce sont des traits principalement utilisés pour lier des variables de type génériques afin d'exprimer des contraintes que vous ne pouvez pas capturer autrement. Ceux-ci incluent `et`. `Sized` `Copy`

Traits de vocabulaire public

Ceux-ci n'ont pas d'intégration de compilateur magique; vous pouvez définir des traits équivalents dans votre propre code. Mais ils servent l'objectif important de définir des solutions conventionnelles pour des problèmes communs. Ceux-ci sont particulièrement précieux dans les interfaces publiques entre les caisses et les modules: en réduisant les variations inutiles, ils rendent les interfaces plus faciles à comprendre, mais ils augmentent également la probabilité que les fonctionnalités de différentes caisses puissent simplement être branchées ensemble directement, sans code de colle standard ou personnalisé. Ceux-ci comprennent `,`, les traits d'emprunt de référence `,`, `et`; les caractères de conversion faillibles `et`; et le trait, une généralisation de

```
.Default AsRef AsMut Borrow BorrowMut TryFrom TryInto T
oOwned Clone
```

Celles-ci sont résumées dans [le tableau 13-1.](#)

Tableau 13-1. Résumé des traits d'utilité

Trait	Description
<u>Drop</u>	Destructeurs. Code de nettoyage que Rust exécute automatiquement chaque fois qu'une valeur est supprimée.
<u>Sized</u>	Trait de marqueur pour les types avec une taille fixe connue au moment de la compilation, par opposition aux types (tels que les tranches) qui sont dimensionnés dynamiquement.
<u>Clone</u>	Types prenant en charge les valeurs de clonage.
<u>Copy</u>	Trait de marqueur pour les types qui peuvent être clonés simplement en effectuant une copie octet par octet de la mémoire contenant la valeur.
<u>Deref</u>	Caractéristiques des types de pointeurs intelligents.
<u>et Deref</u>	
<u>fMut</u>	
<u>Defau</u>	Types qui ont une « valeur par défaut » raisonnable.
<u>lt</u>	
<u>AsRef</u>	Caractéristiques de conversion pour emprunter un type de référence à un autre.
<u>et AsMu</u>	
<u>t</u>	
<u>Empru</u>	Traits de conversion, tels que /, mais garantissant en outre
<u>nter et</u>	un hachage, un ordre et une égalité cohérents. AsRef AsM
<u>Emprun</u>	ut
<u>terMut</u>	
<u>De et</u>	Caractéristiques de conversion pour transformer un type
<u>vers</u>	de valeur en un autre.

Trait	Description
<u>TryFrom</u> <u>TryInto</u>	Caractéristiques de conversion pour transformer un type de valeur en un autre, pour les transformations qui pourraient échouer.
<u>ToOwned</u> <u>ed</u>	Caractéristique de conversion pour convertir une référence en une valeur possédée.

Il existe également d'autres caractéristiques de bibliothèque standard importantes. Nous couvrirons et dans [le chapitre 15](#). Le trait, pour le calcul des codes de hachage, est couvert au [chapitre 16](#). Et une paire de traits qui marquent les types sans fil, et , sont couverts au [chapitre 19](#).

Goutte

Lorsque le propriétaire d'une valeur disparaît, nous disons que Rust *baisse* la valeur. La suppression d'une valeur implique la libération des autres valeurs, du stockage en tas et des ressources système que la valeur possède. Les baisses se produisent dans diverses circonstances : lorsqu'une variable sort de sa portée; à la fin d'une instruction d'expression ; lorsque vous tronquez un vecteur, en supprimant des éléments de son extrémité ; et ainsi de suite.

Pour la plupart, Rust gère automatiquement la suppression des valeurs pour vous. Par exemple, supposons que vous définissiez le type suivant :

```
struct Appellation {
    name: String,
    nicknames: Vec<String>
}
```

Un stockage de tas possède le contenu des chaînes et le tampon d'éléments du vecteur. Rust s'occupe de nettoyer tout cela chaque fois qu'un est tombé, sans aucun codage supplémentaire nécessaire de votre part. Cependant, si vous le souhaitez, vous pouvez personnaliser la façon dont Rust supprime les valeurs de votre type en implémentant le trait:

```
Appellation: std::ops::Drop
```

```

trait Drop {
    fn drop(&mut self);
}

```

Une implémentation de est analogue à un destructeur en C++, ou à un finaliseur dans d'autres langages. Lorsqu'une valeur est supprimée, si elle est implémentée , Rust appelle sa méthode, avant de procéder à la suppression des valeurs propres à ses champs ou éléments, comme il le ferait normalement. Cette invocation implicite de est la seule façon d'appeler cette méthode ; si vous essayez de l'invoquer explicitement vous-même, Rust signale cela comme une erreur. `Drop std::ops::Drop drop drop`

Étant donné que Rust appelle une valeur avant de supprimer ses champs ou éléments, la valeur reçue par la méthode est toujours entièrement initialisée. Une implémentation de pour notre type peut tirer pleinement parti de ses champs: `Drop::drop Drop Appellation`

```

impl Drop for Appellation {
    fn drop(&mut self) {
        print!("Dropping {}", self.name);
        if !self.nicknames.is_empty() {
            print!(" (AKA {})".format(self.nicknames.join(", ")));
        }
        println!("");
    }
}

```

Compte tenu de cette implémentation, nous pouvons écrire ce qui suit:

```

{
    let mut a = Appellation {
        name: "Zeus".to_string(),
        nicknames: vec!["cloud collector".to_string(),
                        "king of the gods".to_string()]
    };

    println!("before assignment");
    a = Appellation { name: "Hera".to_string(), nicknames: vec![] };
    println!("at end of block");
}

```

Lorsque nous affectons la seconde à , la première est abandonnée, et lorsque nous quittons la portée de , la seconde est abandonnée. Ce code imprime les éléments suivants : Appellation a a

```
before assignment
Dropping Zeus (AKA cloud collector, king of the gods)
at end of block
Dropping Hera
```

Puisque notre implémentation pour ne fait qu'imprimer un message, comment, exactement, sa mémoire est-elle nettoyée ? Le type implémente , en supprimant chacun de ses éléments, puis en libérant le tampon alloué au tas qu'ils occupaient. A utilise un interne pour tenir son texte, donc pas besoin de s'implémenter lui-même; il lui permet de s'occuper de libérer les personnages. Le même principe s'étend aux valeurs : lorsque l'on est lâché, c'est finalement l'implémentation de qui se charge en fait de libérer le contenu de chacune des chaînes, et enfin de libérer le tampon contenant les éléments du vecteur. Quant à la mémoire qui détient la valeur elle-même, elle a aussi un propriétaire, peut-être une variable locale ou une structure de données, qui est responsable de la

```
libérer. std::ops::Drop Appellation Vec Drop String Vec<u8> String Drop Vec Appellation Vec Drop Appellation
```

Si la valeur d'une variable est déplacée ailleurs, de sorte que la variable n'est pas initialisée lorsqu'elle sort de la portée, Rust n'essaiera pas de supprimer cette variable: il n'y a pas de valeur à supprimer.

Ce principe s'applique même lorsqu'une variable peut ou non avoir vu sa valeur déplacée, en fonction du flux de contrôle. Dans des cas comme celui-ci, Rust garde une trace de l'état de la variable avec un indicateur invisible indiquant si la valeur de la variable doit être supprimée ou non :

```
let p;
{
    let q = Appellation { name: "Cardamine hirsuta".to_string(),
                          nicknames: vec![ "shotweed".to_string(),
                                           "bittercress".to_string() ] }
    if complicated_condition() {
        p = q;
    }
}
println!("Sproing! What was that?");
```

Selon que l'autre retourne ou , soit ou finira par posséder le , avec l'autre non initialisé. L'endroit où il atterrit détermine s'il est abandonné avant ou après le , puisqu'il sort du champ d'application avant le , et après. Bien qu'une valeur puisse être déplacée d'un endroit à l'autre, Rust ne la laisse tomber qu'une seule

```
fois. complicated_condition true false p q Appellation println!  
n! q println! p
```

Vous n'aurez généralement pas besoin d'implémenter à moins que vous ne définissiez un type qui possède des ressources que Rust ne connaît pas déjà. Par exemple, sur les systèmes Unix, la bibliothèque standard de Rust utilise le type suivant en interne pour représenter un descripteur de fichier du système d'exploitation : `std::ops::Drop`

```
struct FileDesc {  
    fd: c_int,  
}
```

Le champ de `a` est simplement le numéro du descripteur de fichier qui doit être fermé lorsque le programme en a terminé; `c` est un alias pour `.La`. La bibliothèque standard implémente ce qui suit

```
: fd FileDesc c_int i32 Drop FileDesc
```

```
impl Drop for FileDesc {  
    fn drop(&mut self) {  
        let _ = unsafe { libc::close(self.fd) };  
    }  
}
```

Voici le nom Rust pour la fonction de la bibliothèque C. Le code Rust peut appeler des fonctions C uniquement dans les blocs, de sorte que la bibliothèque en utilise une ici. `libc::close` `close` `unsafe`

Si un type implémente `Drop`, il ne peut pas implémenter le trait. Si un type est `Copy`, cela signifie qu'une simple duplication octet par octet est suffisante pour produire une copie indépendante de la valeur. Mais c'est généralement une erreur d'appeler la même méthode plus d'une fois sur les mêmes données. `Drop` `Copy` `Copy` `drop`

Le prélude standard inclut une fonction pour supprimer une valeur, mais sa définition est tout sauf magique : `drop`

```
fn drop<T>(_x: T) { }
```

En d'autres termes, il reçoit son argument par valeur, prenant la propriété de l'appelant, puis n'en fait rien. La rouille diminue la valeur du moment où elle sort de son champ d'application, comme elle le ferait pour toute autre variable. `_x`

Taille

Un *type de taille* est un type dont les valeurs ont toutes la même taille en mémoire. Presque tous les types de Rust sont dimensionnés: chaque prend huit octets, chaque tuple douze. Même les enums sont dimensionnés : quelle que soit la variante réellement présente, un enum occupe toujours suffisamment d'espace pour contenir sa plus grande variante. Et bien qu'un possède un tampon alloué au tas dont la taille peut varier, la valeur elle-même est un pointeur vers le tampon, sa capacité et sa longueur, de même qu'un type de taille. `u64` (`f32`, `f32`, `f32`) `Vec<T>` `Vec` `Vec<T>`

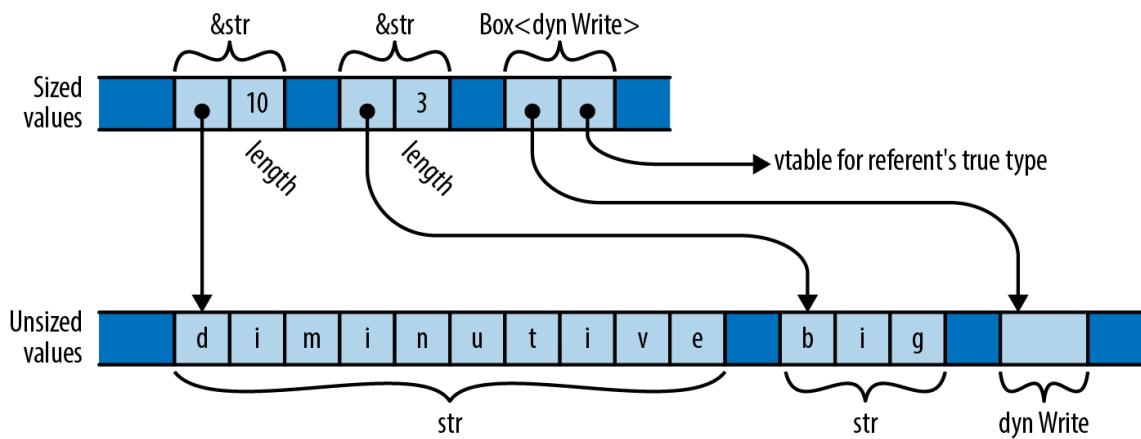
Tous les types de taille implémentent le trait, qui n'a pas de méthodes ou de types associés. Rust l'implémente automatiquement pour tous les types auxquels il s'applique; vous ne pouvez pas le mettre en œuvre vous-même. La seule utilisation pour est comme une limite pour les variables de type: un comme lié nécessite d'être un type dont la taille est connue au moment de la compilation. Les traits de ce type sont *appelés traits marqueurs*, car le langage Rust lui-même les utilise pour marquer certains types comme ayant des caractéristiques

d'intérêt. `std::marker::Sized` `Sized T: Sized`

Cependant, Rust a également *quelques types non dimensionnés* dont les valeurs ne sont pas toutes de la même taille. Par exemple, le type de tranche de chaîne (note, sans) n'est pas dimensionné. Les littéraux de chaîne et sont des références à des tranches qui occupent dix et trois octets. Les deux sont illustrés à [la figure 13-1](#). Les types de tranches de tableau comme (encore une fois, sans) sont également non dimensionnés : une référence partagée peut pointer vers une tranche de n'importe quelle taille. Étant donné que les et types désignent des ensembles de valeurs de tailles variables, il s'agit de types non

dimensionnés. str & "diminutive" "big" str [T] & &[u8]

[u8] str [T]



Graphique 13-1. Références à des valeurs non dimensionnées

L'autre type commun de type non dimensionné dans Rust est un type, le référent d'un objet trait. Comme nous l'avons expliqué dans [« Trait Objects »](#), un objet trait est un pointeur vers une valeur qui implémente un trait donné. Par exemple, les types et sont des pointeurs vers une valeur qui implémente le trait. Le référent peut être un fichier ou un socket réseau ou un type de votre propre pour lequel vous avez implémenté . Étant donné que l'ensemble des types qui implémentent est ouvert, considéré comme un type est non dimensionné: ses valeurs ont différentes tailles. `dyn &dyn std::io::Write Box<dyn std::io::Write> Write Write Write dyn Write`

Rust ne peut pas stocker des valeurs non dimensionnées dans des variables ou les transmettre comme arguments. Vous ne pouvez les traiter qu'à travers des pointeurs comme ou , qui sont eux-mêmes dimensionnés. Comme le montre [la figure 13-1](#), un pointeur vers une valeur non dimensionnée est toujours un *pointeur gras*, large de deux mots : un pointeur vers une tranche porte également la longueur de la tranche et un objet trait porte également un pointeur vers un vtable d'implémentations de méthode. `&str Box<dyn Write>`

Les objets traits et les pointeurs vers les tranches sont bien symétriques. Dans les deux cas, le type manque d'informations nécessaires pour l'utiliser : vous ne pouvez pas indexer a sans connaître sa longueur, ni appeler une méthode sur a sans connaître l'implémentation de approprié à la valeur spécifique à laquelle il se réfère. Et dans les deux cas, le pointeur de graisse remplit les informations manquantes dans le type, portant un pointeur de longueur ou de vtable. Les informations statiques omises

sont remplacées par des informations dynamiques. [u8] Box<dyn

Write> Write

Étant donné que les types non dimensionnés sont si limités, la plupart des variables de type génériques devraient être limitées aux types. En fait, cela est nécessaire si souvent que c'est la valeur implicite par défaut dans Rust: si vous écrivez , Rust vous comprend comme signifiant . Si vous ne voulez pas contraindre de cette façon, vous devez explicitement vous désinscrire, en écrivant . La syntaxe est spécifique à ce cas et signifie « pas nécessairement ». Par exemple, si vous écrivez struct S<T: ?Size> { b: Box<T> } , puis Rust vous permettra d'écrire et , où la boîte devient un gros pointeur, ainsi que et , où la boîte est un pointeur ordinaire. Sized struct S<T> { ... } struct S<T: Sized> { ... } T struct S<T: ?Sized> { ... } ?Sized Sized S<str> S<dyn Write> S<i32> S<String>

Malgré leurs restrictions, les types non dimensionnés rendent le système de type de Rust plus fluide. En lisant la documentation standard de la bibliothèque, vous rencontrerez parfois une liaison sur une variable de type; cela signifie presque toujours que le type donné est uniquement pointé vers et permet au code associé de fonctionner avec des tranches et des objets de trait ainsi que des valeurs ordinaires. Lorsqu'une variable de type a la limite, les gens dis-le souvent est *de taille douteuse*: elle peut l'être ou non. ?Sized ?Sized Sized

Mis à part les tranches et les objets traits, il existe un autre type de type non dimensionné. Le dernier champ d'un type struct (mais seulement son dernier) peut être dédimensionné, et un tel struct est lui-même non dimensionné. Par exemple, un pointeur compté en références est implémenté en interne en tant que pointeur vers le type privé , qui stocke le nombre de références à côté du fichier . Voici une définition simplifiée de : Rc<T> RcBox<T> T RcBox

```
struct RcBox<T: ?Sized> {
    ref_count: usize,
    value: T,
}
```

Le champ est celui auquel on compte les références; déréférence à un pointeur vers ce champ. Le champ contient le nombre de références. value T Rc<T> Rc<T> ref_count

Le réel n'est qu'un détail d'implémentation de la bibliothèque standard et n'est pas disponible pour un usage public. Mais supposons que nous travaillions avec la définition précédente. Vous pouvez l'utiliser avec des types de taille, comme ; le résultat est un type de structure de taille. Ou vous pouvez l'utiliser avec des types non dimensionnés, comme (où est le trait pour les types qui peuvent être formatés par et des macros similaires); est un type de structure non

```
RcBox RcBox RcBox<String> RcBox<dyn  
std::fmt::Display> Display println! RcBox<dyn Display>
```

Vous ne pouvez pas créer une valeur directement. Au lieu de cela, vous devez d'abord créer un ordinaire, de taille dont le type implémente , comme . La rouille permet alors de convertir une référence en référence grasse : RcBox<dyn

```
Display> RcBox value Display RcBox<String> &RcBox<String> &  
RcBox<dyn Display>
```

```
let boxed_lunch: RcBox<String> = RcBox {  
    ref_count: 1,  
    value: "lunch".to_string()  
};  
  
use std::fmt::Display;  
let boxed_displayable: &RcBox<dyn Display> = &boxed_lunch;
```

Cette conversion se produit implicitement lors de la transmission de valeurs à des fonctions, de sorte que vous pouvez passer un à une fonction qui attend un : &RcBox<String> &RcBox<dyn Display>

```
fn display(boxed: &RcBox<dyn Display>) {  
    println!("For your enjoyment: {}", &boxed.value);  
}  
  
display(&boxed_lunch);
```

Cela produirait les résultats suivants :

```
For your enjoyment: lunch
```

Clone

Le trait est pour les types qui peuvent faire des copies d'eux-mêmes. est défini comme suit : `std::clone::Clone`

```
trait Clone: Sized {
    fn clone(&self) -> Self;
    fn clone_from(&mut self, source: &Self) {
        *self = source.clone()
    }
}
```

La méthode doit construire une copie indépendante et la renvoyer. Étant donné que le type de retour de cette méthode est et que les fonctions peuvent ne pas renvoyer de valeurs non dimensionnées, le trait lui-même étend le trait : cela a pour effet de limiter les types des implémentations à être `.clone` `self` `Self` `Sized` `Self` `Sized`

Le clonage d'une valeur implique généralement l'attribution de copies de tout ce qu'elle possède, de sorte qu'un peut être coûteux, à la fois en temps et en mémoire. Par exemple, le clonage a non seulement copié le vecteur, mais copié également chacun de ses éléments. C'est pourquoi Rust ne se contente pas de cloner automatiquement les valeurs, mais vous oblige plutôt à effectuer un appel de méthode explicite. Les types de pointeurs comptés par référence sont des exceptions : le clonage de l'un d'entre eux incrémentera simplement le nombre de références et vous remet un nouveau

```
pointeur.clone Vec<String> String Rc<T> Arc<T>
```

La méthode se transforme en une copie de . La définition par défaut de simplement cloner, puis déplace cela dans . Cela fonctionne toujours, mais pour certains types, il existe un moyen plus rapide d'obtenir le même effet. Par exemple, supposons et sont s. L'instruction doit cloner , supprimer l'ancienne valeur de , puis déplacer la valeur clonée dans ; il s'agit d'une allocation de tas et d'une désallocation de tas. Mais si le tampon de tas appartenant à l'original a une capacité suffisante pour contenir le contenu, aucune allocation ou désallocation n'est nécessaire: vous pouvez simplement copier le texte de 's dans le tampon de 's et ajuster la longueur. Dans le code générique, vous devez utiliser autant que possible pour tirer parti des implémentations optimisées lorsqu'elles sont présentes.

```
clone_from self source clone_from source *self s t S
string s = t.clone(); t s s s t t s clone_from
```

Si votre implémentation s'applique simplement à chaque champ ou élément de votre type, puis construit une nouvelle valeur à partir de ces clones, et que la définition par défaut de est assez bonne, alors Rust l'implémentera pour vous: mettez simplement au-dessus de votre définition de type. `Clone` `clone` `clone_from` `#[derive(Clone)]`

Presque tous les types de la bibliothèque standard qui ont du sens pour copier les implémentations. Les types primitifs aiment et font. Les types de conteneurs comme , et le font aussi. Certains types n'ont pas de sens à copier, comme ; ceux-ci n'implémentent pas . Certains types peuvent être copiés, mais la copie peut échouer si le système d'exploitation ne dispose pas des ressources nécessaires ; ces types n'implémentent pas , car ils doivent être infaillibles. Au lieu de cela, fournit une méthode qui renvoie un , qui peut signaler un

```
échec. Clone bool i32 String Vec<T> HashMap std::sync::Mutex C  
lone std::fs::File Clone std::fs::File try_clone std:  
:io::Result<File>
```

Copier

Au [chapitre 4](#), nous avons expliqué que, pour la plupart des types, l'affectation déplace les valeurs plutôt que de les copier. Le déplacement des valeurs rend beaucoup plus simple le suivi des ressources qu'ils possèdent. Mais dans [« Copy Types: The Exception to Moves »](#), nous avons souligné l'exception: les types simples qui ne possèdent aucune ressource peuvent être des types, où l'affectation fait une copie de la source, plutôt que de déplacer la valeur et de laisser la source non initialisée. `Copy`

À ce moment-là, nous avons laissé vague ce qui était exactement, mais maintenant nous pouvons vous dire: un type est s'il implémente le trait marqueur, qui est défini comme suit: `Copy` `Copy` `std::marker::Copy`

```
trait Copy: Clone { }
```

Ceci est certainement facile à mettre en œuvre pour vos propres types:

```
impl Copy for MyType { }
```

Mais parce qu'il s'agit d'un trait marqueur ayant une signification particulière pour le langage, Rust permet à un type de l'implémenter unique-

ment si une copie superficielle octet pour octet est tout ce dont il a besoin. Les types qui possèdent d'autres ressources, telles que les tampons de tas ou les descripteurs de système d'exploitation, ne peuvent pas implémenter . Copy Copy Copy

Tout type qui implémente le trait ne peut pas être . Rust suppose que si un type a besoin d'un code de nettoyage spécial, il doit également nécessiter un code de copie spécial et ne peut donc pas être . Drop Copy Copy

Comme avec , vous pouvez demander à Rust de dériver pour vous, en utilisant . Vous verrez souvent les deux dérivés à la fois, avec . Clone Copy #[derive(Copy)] #[derive(Copy, Clone)]

Réfléchissez bien avant de faire un type . Bien que cela rende le type plus facile à utiliser, il impose de lourdes restrictions à sa mise en œuvre. Les copies implicites peuvent également être coûteuses. Nous expliquons ces facteurs en détail dans [« Types de copie: l'exception aux déplacements »](#). Copy

Deref et DerefMut

Vous pouvez spécifier comment les opérateurs de déréférencement aiment et se comportent sur vos types en implementant les traits et. Les types de pointeurs aiment et implémentent ces traits afin qu'ils puissent se comporter comme le font les types de pointeurs intégrés de Rust. Par exemple, si vous avez une valeur , alors fait référence à la valeur qui pointe vers, et fait référence à sa composante réelle. Si le contexte attribue ou emprunte une référence mutable au référent, Rust utilise le trait (« déréférer mutable ») ; sinon, l'accès en lecture seule suffit et utilise

```
.* . std::ops::Deref std::ops::DerefMut Box<T> Rc<T> Box<Complex> b *b Complex b b.re DerefMut Deref
```

Les traits sont définis comme suit:

```
trait Deref {  
    type Target: ?Sized;  
    fn deref(&self) -> &Self::Target;  
}  
  
trait DerefMut: Deref {  
    fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

```
fn deref_mut(&mut self) -> &mut Self::Target;  
}
```

Les méthodes et prennent une référence et renvoient une référence. devrait être quelque chose qui contient, possède ou fait référence à: pour le type est . Notez que cela s'étend : si vous pouvez déréférencer quelque chose et le modifier, vous devriez certainement pouvoir emprunter une référence partagée à celui-ci également. Étant donné que les méthodes renvoient une référence ayant la même durée de vie que , reste empruntée aussi longtemps que la référence renvoyée

```
vit.deref deref_mut &Self &Self::Target Target Self Box<Complex> Target Complex DerefMut Deref &self self
```

Les et traits jouent également un autre rôle. Puisque prend une référence et renvoie une référence, Rust l'utilise pour convertir automatiquement les références du premier type en seconde. En d'autres termes, si l'insertion d'un appel empêche une incompatibilité de type, Rust en insère un pour vous. L'implémentation active la conversion correspondante pour les références modifiables. C'est ce qu'on appelle les *coercitions deref*: un type est « constraint » à se comporter comme un

```
autre. Deref DerefMut deref &Self &Self::Target deref DerefMut
```

Bien que les coercitions de deref ne soient pas quelque chose que vous ne pourriez pas écrire explicitement vous-même, elles sont pratiques:

- Si vous avez une certaine valeur et que vous voulez l'appliquer, vous pouvez simplement écrire , au lieu de : l'appel de méthode emprunte implicitement , et contraint à , car implémente

```
.Rc<String> r String::find r.find('?')  
(*r).find('?') r &Rc<String> &String Rc<T> Deref<Target=T>
```

- Vous pouvez utiliser des méthodes telles que sur les valeurs, même s'il s'agit d'une méthode du type de tranche, car implémente . Il n'est pas nécessaire de réimplémenter toutes les méthodes de , car vous pouvez contraindre un fichier

```
.split_at String split_at str String Deref<Target=str> String str &str &String
```

- Si vous avez un vecteur d'octets et que vous souhaitez le passer à une fonction qui attend une tranche d'octets , vous pouvez simplement

passer comme argument, puisque implémente .v &

```
[u8] &v Vec<T> Deref<Target=[T]>
```

Rust appliquera plusieurs coercitions deref successivement si nécessaire. Par exemple, en utilisant les coercitions mentionnées précédemment, vous pouvez appliquer directement à un , puisque déréférences à , qui déréférence à , qui a la

```
méthode. split_at Rc<String> &Rc<String> &String &str split_at
```

Par exemple, supposons que vous ayez le type suivant :

```
struct Selector<T> {
    // Elements available in this `Selector`.
    elements: Vec<T>,
    // The index of the "current" element in `elements`. A `Selector`
    // behaves like a pointer to the current element.
    current: usize
}
```

Pour que le comportement comme les revendications de commentaire de document, vous devez implémenter et pour le type

```
: Selector Deref DerefMut
```

```
use std::ops::{Deref, DerefMut};

impl<T> Deref for Selector<T> {
    type Target = T;
    fn deref(&self) -> &T {
        &self.elements[self.current]
    }
}

impl<T> DerefMut for Selector<T> {
    fn deref_mut(&mut self) -> &mut T {
        &mut self.elements[self.current]
    }
}
```

Compte tenu de ces implémentations, vous pouvez utiliser un comme ceci: Selector

```

let mut s = Selector { elements: vec!['x', 'y', 'z'],
                      current: 2 };

// Because `Selector` implements `Deref`, we can use the `*` operator
// refer to its current element.
assert_eq!(*s, 'z');

// Assert that 'z' is alphabetic, using a method of `char` directly or
// `Selector`, via deref coercion.
assert!(s.is_alphabetic());

// Change the 'z' to a 'w', by assigning to the `Selector`'s referent.
*s = 'w';

assert_eq!(s.elements, ['x', 'y', 'w']);

```

Les et traits sont conçus pour implémenter des types de pointeurs intelligents, tels que , et , et des types qui servent de versions propriétaires de quelque chose que vous utiliseriez également fréquemment par référence, de la manière et servir de versions propriétaires de et . Vous ne devez pas implémenter et pour un type juste pour faire apparaître automatiquement les méthodes du type dessus, la façon dont les méthodes d'une classe de base C ++ sont visibles sur une sous-classe. Cela ne fonctionnera pas toujours comme vous vous y attendez et peut être déroutant lorsque cela tourne

mal. Deref DerefMut Box Rc Arc Vec<T> String [T] str Deref De
refMut Target

Les coercitions deref viennent avec une mise en garde qui peut causer une certaine confusion: Rust les applique pour résoudre les conflits de type, mais pas pour satisfaire les limites sur les variables de type. Par exemple, le code suivant fonctionne correctement :

```

let s = Selector { elements: vec!["good", "bad", "ugly"],
                   current: 2 };

fn show_it(thing: &str) { println!("{}", thing); }
show_it(&s);

```

Dans l'appel , Rust voit un argument de type et un paramètre de type , trouve l'implémentation et réécrit l'appel en tant que , selon les

```
besoins. show_it(&s) &Selector<&str> &str Deref<Target=str> s  
how_it(s.deref())
```

Cependant, si vous passez à une fonction générique, Rust n'est soudainement plus coopératif: `show_it`

```
use std::fmt::Display;  
fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }  
show_it_generic(&s);
```

Rust se plaint:

```
error: `Selector<&str>` doesn't implement `std::fmt::Display`  
|  
31 |     show_it_generic(&s);  
|         ^^  
|  
|             `Selector<&str>` cannot be formatted with  
|             the default formatter  
|             help: consider adding dereference here: `&*s`  
|  
|  
note: required by a bound in `show_it_generic`  
|  
30 |     fn show_it_generic<T: Display>(thing: T) { println!("{}", thing); }  
|             ^^^^^^ required by this bound  
|             in `show_it_generic`
```

Cela peut être déroutant : comment rendre une fonction générique pourra-t-elle introduire une erreur ? Certes, ne s'implémente pas lui-même, mais il fait référence à , ce qui est certainement le cas. `Selector<&str> Display &str`

Étant donné que vous passez un argument de type et que le type de paramètre de la fonction est , la variable de type doit être . Ensuite, Rust vérifie si la liaison est satisfaite : comme elle n'applique pas de contraintes de ref pour satisfaire les limites sur les variables de type, cette vérification échoue. `&Selector<&str> &T T Selector<&str> T: Display`

Pour contourner ce problème, vous pouvez épeler la coercition à l'aide de l'opérateur : `as`

```
show_it_generic(&s as &str);
```

Ou, comme le suggère le compilateur, vous pouvez forcer la coercition avec : &*

```
show_it_generic(&*s);
```

Faire défaut

Certains types ont une valeur par défaut raisonnablement évidente : le vecteur ou la chaîne par défaut est vide, le nombre par défaut est zéro, la valeur par défaut est , et ainsi de suite. Des types comme celui-ci peuvent implémenter le trait: Option None std::default::Default

```
trait Default {  
    fn default() -> Self;  
}
```

La méthode renvoie simplement une nouvelle valeur de type . est simple: default Self String Default

```
impl Default for String {  
    fn default() -> String {  
        String::new()  
    }  
}
```

Tous les types de collection de Rust—, , , etc. : implémentez , avec des méthodes qui renvoient une collection vide. Ceci est utile lorsque vous devez créer une collection de valeurs, mais que vous souhaitez laisser votre appelant décider exactement du type de collection à créer. Par exemple, la méthode du trait divise les valeurs produites par l'itérateur en deux collections, en utilisant une fermeture pour décider où va chaque valeur

```
: Vec HashMap BinaryHeap Default default Iterator partition
```

```
use std::collections::HashSet;  
let squares = [4, 9, 16, 25, 36, 49, 64];  
let (powers_of_two, impure): (HashSet<i32>, HashSet<i32>) =  
    squares.iter().partition(|&n| n & (n-1) == 0);  
  
assert_eq!(powers_of_two.len(), 3);  
assert_eq!(impure.len(), 4);
```

La fermeture utilise un peu de bricolage pour reconnaître les nombres qui sont des puissances de deux, et l'utilise pour produire deux s. Mais bien sûr, n'est pas spécifique à s; vous pouvez l'utiliser pour produire n'importe quel type de collection que vous aimez, tant que le type de collection implémente , pour produire une collection vide pour commencer, et , pour ajouter un à la collection. implémente et , afin que vous puissiez écrire : | &n| n & (n-1) ==

```
0 partition HashSet partition HashSet Default Extend<T> T St  
ring Default Extend<char>
```

```
let (upper, lower): (String, String)  
    = "Great Teacher Onizuka".chars().partition(|&c| c.is_uppercase())  
assert_eq!(upper, "GTO");  
assert_eq!(lower, "reat eacher nizuka");
```

Une autre utilisation courante de est de produire des valeurs par défaut pour les structs qui représentent une grande collection de paramètres, dont la plupart n'auront généralement pas besoin de modifier. Par exemple, la caisse fournit des liaisons Rust pour la bibliothèque graphique OpenGL puissante et complexe. La structure comprend 24 champs, chacun contrôlant un détail différent de la façon dont OpenGL doit rendre un peu de graphiques. La fonction attend une struct comme argument.

Puisque implémente , vous pouvez en créer un à passer à , en mentionnant uniquement les champs que vous souhaitez

```
modifier: Default glum glum::DrawParameters glum draw Draw  
Parameters DrawParameters Default draw
```

```
let params = glum::DrawParameters {  
    line_width: Some(0.02),  
    point_size: Some(0.02),  
    .. Default::default()  
};  
  
target.draw(..., &params).unwrap();
```

Cela appelle à créer une valeur initialisée avec les valeurs par défaut de tous ses champs, puis utilise la syntaxe des structs pour en créer une nouvelle avec les champs et modifiés, prêt à vous permettre de le passer à .Default::default() DrawParameters .. line_width point_size t arget.draw

Si un type implémente , la bibliothèque standard implémente automatiquement pour , , , , et . La valeur par défaut du type , par exemple, est un pointage vers la valeur par défaut du type

```
. T Default Default Rc<T> Arc<T> Box<T> Cell<T> RefCell<T> Co  
w<T> Mutex<T> RwLock<T> Rc<T> Rc T
```

Si tous les types d'éléments d'un type de tuple implémentent , alors le type de tuple le fait aussi, par défaut à un tuple contenant la valeur par défaut de chaque élément. `Default`

Rust n'implémente pas implicitement pour les types struct, mais si tous les champs d'une struct implémentent , vous pouvez implémenter automatiquement pour la struct à l'aide de `.Default Default Default # [derive(Default)]`

AsRef et AsMut

Lorsqu'un type implémente , cela signifie que vous pouvez en emprunter un efficacement. est l'analogue des références modifiables. Leurs définitions sont les suivantes : `AsRef<T>` &`T` `AsMut`

```
trait AsRef<T: ?Sized> {  
    fn as_ref(&self) -> &T;  
}  
  
trait AsMut<T: ?Sized> {  
    fn as_mut(&mut self) -> &mut T;  
}
```

Ainsi, par exemple, implémente , et implémente . Vous pouvez également emprunter le contenu d'un 's sous la forme d'un tableau d'octets, donc implémente

également. `Vec<T>` `AsRef<[T]>` `String` `AsRef<str>` `String` `String` `A
sRef<[u8]>`

`AsRef` est généralement utilisé pour rendre les fonctions plus flexibles dans les types d'arguments qu'elles acceptent. Par exemple, la fonction est déclarée comme suit : `std::fs::File::open`

```
fn open<P: AsRef<Path>>(path: P) -> Result<File>
```

Ce que l'on veut vraiment, c'est un , le type représentant un chemin d'accès au système de fichiers. Mais avec cette signature, accepte tout ce qu'il peut emprunter, c'est-à-dire tout ce qui implémente . Ces types incluent et , les types de chaîne d'interface du système d'exploitation et , et bien sûr et ; consultez la documentation de la bibliothèque pour la liste complète.

C'est ce qui permet de passer des littéraux de chaîne à

```
:open &Path open &Path AsRef<Path> String str OsString OsSt  
r PathBuf Path open
```

```
let dot_emacs = std::fs::File::open("/home/jimb/.emacs")?;
```

Toutes les fonctions d'accès au système de fichiers de la bibliothèque standard acceptent les arguments de chemin d'accès de cette façon. Pour les appelants, l'effet ressemble à celui d'une fonction surchargée en C++, bien que Rust adopte une approche différente pour établir quels types d'arguments sont acceptables.

Mais cela ne peut pas être toute l'histoire. Un littéral de chaîne est un , mais le type qui implémente est , sans un . Et comme nous l'avons expliqué dans [« Deref et DerefMut »](#), Rust n'essaie pas de dissuader les contraintes de ref pour satisfaire les limites des variables de type, donc elles n'aideront pas non plus ici. &str AsRef<Path> str &

Heureusement, la bibliothèque standard inclut l'implémentation générale

:

```
impl<'a, T, U> AsRef<U> for &'a T  
where T: AsRef<U>,  
      T: ?Sized, U: ?Sized  
{  
    fn as_ref(&self) -> &U {  
        (*self).as_ref()  
    }  
}
```

En d'autres termes, pour tous les types et , si , alors aussi bien: suivez simplement la référence et procédez comme avant. En particulier, depuis , alors aussi. Dans un sens, c'est un moyen d'obtenir une forme limitée de coercition deref dans la vérification des limites sur les variables de type. T U T: AsRef<U> &T: AsRef<U> str: AsRef<Path> &str:
AsRef<Path> AsRef

Vous pouvez supposer que si un type implémente , il doit également implémenter . Cependant, il y a des cas où ce n'est pas approprié. Par exemple, nous avons mentionné que les implementations ; cela a du sens, car chacun a certainement un tampon d'octets auquel il peut être utile d'accéder en tant que données binaires. Cependant, garantit en outre que ces octets sont un codage UTF-8 bien formé du texte Unicode; s'il était implémenté, cela permettrait aux appelants de changer les octets de 's en ce qu'ils voulaient, et vous ne pourriez plus faire confiance à un UTF-8 bien formé. Il n'est logique pour un type d'implémenter que si la modification du donné ne peut pas violer les invariants du

```
type. AsRef<T> AsMut<T> String AsRef<[u8]> String String String String AsMut<[u8]> String String AsMut<T> T
```

Bien que et soient assez simples, fournir des traits standard et génériques pour la conversion de référence évite la prolifération de traits de conversion plus spécifiques. Vous devriez éviter de définir vos propres traits lorsque vous pourriez simplement implémenter

```
. AsRef AsMut AsFoo AsRef<Foo>
```

Emprunter et EmprunterMut

Le trait est similaire à : si un type implémente , alors sa méthode lui emprunte efficacement un. Mais impose plus de restrictions: un type ne doit être implémenté que lorsqu'un hachage et compare de la même manière que la valeur à laquelle il est emprunté. (La rouille n'applique pas cela; c'est juste l'intention documentée du trait.) Cela est utile dans le traitement des clés dans les tables de hachage et les arbres ou lorsqu'il s'agit de valeurs qui seront hachées ou comparées pour une autre

```
raison. std::borrow::Borrow AsRef Borrow<T> borrow &T Borrow Borrow B orrow<T> &T Borrow
```

Cette distinction est importante lorsque vous empruntez à s, par exemple: implémente , et , mais ces trois types cibles auront généralement des valeurs de hachage différentes. Seule la tranche est garantie de hachage comme l'équivalent , donc implémente uniquement

```
. String String AsRef<str> AsRef<[u8]> AsRef<Path> &str String String Borrow<str>
```

Borrow est identique à celle de ; seuls les noms ont été modifiés : AsRef

```

trait Borrow<Borrowed: ?Sized> {
    fn borrow(&self) -> &Borrowed;
}

```

Borrow est conçu pour répondre à une situation spécifique avec des tables de hachage génériques et d'autres types de collections associatives. Par exemple, supposons que vous ayez un , mappant des chaînes à des nombres. Les clés de cette table sont s; chaque entrée en possède une. Quelle doit être la signature de la méthode qui recherche une entrée dans ce tableau ? Voici une première tentative : std::collections ::HashMap<String, i32> String

```

impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: K) -> Option<&V> { ... }
}

```

Cela a du sens : pour rechercher une entrée, vous devez fournir une clé du type approprié pour la table. Mais dans ce cas, est ; cette signature vous obligerait à passer un by value à chaque appel à , ce qui est clairement du gaspillage. Vous avez vraiment besoin d'une référence à la clé: K String String get

```

impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get(&self, key: &K) -> Option<&V> { ... }
}

```

C'est légèrement mieux, mais maintenant vous devez passer la clé comme un , donc si vous voulez rechercher une chaîne constante, vous devriez écrire: &String

```
 hashtable.get(&"twenty-two".to_string())
```

C'est ridicule: il alloue un tampon sur le tas et y copie le texte, juste pour pouvoir l'emprunter en tant que , le passer à , puis le déposer. String &String get

Il devrait être assez bon pour passer tout ce qui peut être haché et comparé à notre type de clé; a devrait être parfaitement adéquat, par exem-

ple. Voici donc l'itération finale, qui est ce que vous trouverez dans la bibliothèque standard: `&str`

```
impl<K, V> HashMap<K, V> where K: Eq + Hash
{
    fn get<Q: ?Sized>(&self, key: &Q) -> Option<&V>
        where K: Borrow<Q>,
              Q: Eq + Hash
    { ... }
}
```

En d'autres termes, si vous pouvez emprunter la clé d'une entrée en tant que clé et que la référence résultante hache et compare exactement la clé elle-même, alors clairement devrait être un type de clé acceptable.

Puisque implémente et , cette version finale de vous permet de passer soit ou comme une clé, selon les

```
besoins. &Q &Q String Borrow<str> Borrow<String> get &String &
str
```

`Vec<T>` et mettre en œuvre . Chaque type de chaîne permet d'emprunter son type de tranche correspondant : implémente, implémente, etc. Et tous les types de collections associatives de la bibliothèque standard utilisent pour décider quels types peuvent être transmis à leurs fonctions de recherche. [T:

```
N] Borrow<[T]> String Borrow<str> PathBuf Borrow<Path> Borrow
```

La bibliothèque standard comprend une implémentation générale afin que chaque type puisse être emprunté à lui-même: . Cela garantit qu'il s'agit toujours d'un type acceptable pour la recherche d'entrées dans un fichier . T T: Borrow<T> &K HashMap<K, V>

Pour plus de commodité, chaque type implémente également , renvoyant une référence partagée comme d'habitude. Cela vous permet de transmettre des références modifiables aux fonctions de recherche de collection sans avoir à réemprunter une référence partagée, émulant ainsi la coercition implicite habituelle de Rust des références mutables aux références partagées. `&mut T Borrow<T> &T`

Le trait est l'analogue de pour les références mutables: `BorrowMut` `Borrow`

```
trait BorrowMut<Borrowed: ?Sized>: Borrow<Borrowed> {
    fn borrow_mut(&mut self) -> &mut Borrowed;
}
```

Les mêmes attentes décrivent également Borrow et BorrowMut

De et vers

Les et traits représentent les conversions qui consomment une valeur d'un type et renvoient une valeur d'un autre. Alors que les et traits empruntent une référence d'un type à un autre, et s'approprient leur argument, le transforment, puis renvoient la propriété du résultat à l'appelant.

```
std::convert::From std::convert::Into AsRef AsMut From Into
```

Leurs définitions sont joliment symétriques :

```
trait Into<T>: Sized {
    fn into(self) -> T;
}

trait From<T>: Sized {
    fn from(other: T) -> Self;
}
```

La bibliothèque standard implémente automatiquement la conversion triviale de chaque type à lui-même : chaque type implémente et

```
.T From<T> Into<T>
```

Bien que les traits fournissent simplement deux façons de faire la même chose, ils se prêtent à des utilisations différentes.

Vous utilisez généralement pour rendre vos fonctions plus flexibles dans les arguments qu'ils acceptent. Par exemple, si vous écrivez : Into

```
use std::net::Ipv4Addr;
fn ping<A>(address: A) -> std::io::Result<bool>
    where A: Into<Ipv4Addr>
{
    let ipv4_address = address.into();
    ...
}
```

alors peut accepter non seulement un comme argument, mais aussi un ou un tableau, puisque ces deux types se trouvent commodément à implémenter . (Il est parfois utile de traiter une adresse IPv4 comme une valeur unique de 32 bits ou un tableau de 4 octets.) Parce que la seule chose que l'on sait, c'est qu'il implémente , il n'est pas nécessaire de spécifier le type que vous voulez lorsque vous appelez ; il n'y en a qu'un qui pourrait éventuellement fonctionner, alors l'inférence de type le remplit pour vous.

```
ping Ipv4Addr u32 [u8;
```

```
4] Into<Ipv4Addr> ping address Into<Ipv4Addr> into
```

Comme dans la section précédente, l'effet est un peu similaire à celui de la surcharge d'une fonction en C++. Avec la définition d'avant, nous pouvons faire n'importe lequel de ces appels: `AsRef ping`

```
println!("{:?}", ping(Ipv4Addr::new(23, 21, 68, 141))); // pass an Ipv4Addr
println!("{:?}", ping([66, 146, 219, 98]));           // pass a [u8; 4]
println!("{:?}", ping(0xd076eb94_u32));                // pass a u32
```

Le trait, cependant, joue un rôle différent. La méthode sert de constructeur générique pour produire une instance d'un type à partir d'une autre valeur unique. Par exemple, plutôt que d'avoir deux méthodes nommées `new` et `new_v2`, il implémente simplement `new` et nous permettant d'écrire

```
: From from Ipv4Addr from_array from_u32 From<[u8; 4]> From<u32>
```

```
let addr1 = Ipv4Addr::from([66, 146, 219, 98]);
let addr2 = Ipv4Addr::from(0xd076eb94_u32);
```

Nous pouvons laisser l'inférence de type déterminer quelle implémentation s'applique.

Avec une implémentation appropriée, la bibliothèque standard implémente automatiquement le trait correspondant. Lorsque vous définissez votre propre type, s'il a des constructeurs à argument unique, vous devez les écrire en tant qu'implémentations des types appropriés ; vous obtiendrez les implémentations correspondantes gratuitement. `From Into From<T> Into`

Étant donné que les méthodes de conversion prennent possession de leurs arguments, une conversion peut réutiliser les ressources de la

valeur d'origine pour construire la valeur convertie. Par exemple, supposons que vous écriviez : `from` `into`

```
let text = "Beautiful Soup".to_string();
let bytes: Vec<u8> = text.into();
```

L'implémentation de `for` prend simplement le tampon de tas de `'s` et le réutilise, inchangé, en tant que tampon d'élément du vecteur renvoyé. La conversion n'a pas besoin d'allouer ou de copier le texte. C'est un autre cas où les mouvements permettent des implémentations efficaces.

```
Into<Vec<u8>> String String
```

Ces conversions offrent également un bon moyen de détendre une valeur d'un type contraint en quelque chose de plus flexible, sans affaiblir les garanties du type contraint. Par exemple, `a` garantit que son contenu est toujours valide UTF-8; ses méthodes de mutation sont soigneusement limitées pour s'assurer que rien de ce que vous pouvez faire n'introduira jamais un mauvais UTF-8. Mais cet exemple « rétrograde » efficacement un bloc d'octets simples avec lequel vous pouvez faire tout ce que vous voulez: peut-être que vous allez le compresser, ou le combiner avec d'autres données binaires qui ne sont pas UTF-8. Parce que prend son argument par valeur, n'est plus initialisé après la conversion, ce qui signifie que nous pouvons accéder librement au tampon du premier sans pouvoir corrompre aucun existant.

```
String String into text String String
```

Cependant, les conversions bon marché ne font pas partie du contrat de `et`. Alors que les conversions sont censées être bon marché, et les conversions peuvent allouer, copier ou traiter le contenu de la valeur. Par exemple, `implémente`, qui copie la tranche de chaîne dans un nouveau tampon alloué au tas pour le `.` Et `implémente`, qui compare et réorganise les éléments en fonction des exigences de son

```
algorithme. Into From AsRef AsMut From Into String From<&str> String std::collections::BinaryHeap<T> From<Vec<T>>
```

L'opérateur `utilise` et aide à nettoyer le code dans les fonctions qui pourraient échouer de plusieurs façons en convertissant automatiquement des types d'erreur spécifiques en types d'erreur généraux en cas de besoin.

```
? From Into
```

Par exemple, imaginez un système qui a besoin de lire des données binaires et d'en convertir une partie à partir de nombres de base 10 écrits

sous forme de texte UTF-8. Cela signifie utiliser et l'implémentation pour , qui peut chacun renvoyer des erreurs de différents types. En supposant que nous utilisions les types que nous avons définis au [chapitre 7](#) lorsque nous discutons de la gestion des erreurs, l'opérateur effectuera la conversion pour

```
nous: std::str::from_utf8 FromStr i32 GenericError GenericResult ?
```

```
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>
type GenericResult<T> = Result<T, GenericError>

fn parse_i32_bytes(b: &[u8]) -> GenericResult<i32> {
    Ok(std::str::from_utf8(b)?.parse::<i32>()?)
}
```

Comme la plupart des types d'erreur, et implémentez le trait, et la bibliothèque standard nous donne une couverture pour convertir de tout ce qui implémente en un , qui utilise

```
automatiquement: Utf8Error ParseIntError Error From
impl Error Box<dyn Error> ?
```

```
impl<'a, E: Error + Send + Sync + 'a> From<E>
for Box<dyn Error + Send + Sync + 'a> {
    fn from(err: E) -> Box<dyn Error + Send + Sync + 'a> {
        Box::new(err)
    }
}
```

Cela transforme ce qui aurait été une fonction assez grande avec deux instructions en une seule ligne. `match`

Avant et ont été ajoutés à la bibliothèque standard, le code Rust était plein de traits de conversion ad hoc et de méthodes de construction, chacun spécifique à un seul type. et codifiez les conventions que vous pouvez suivre pour rendre vos types plus faciles à utiliser, puisque vos utilisateurs les connaissent déjà. D'autres bibliothèques et le langage lui-même peuvent également s'appuyer sur ces traits comme moyen canonique et standardisé d'encoder les conversions. `From Into From Into`

`From` et sont des traits infaillibles : leur API exige que les conversions n'échouent pas. Malheureusement, de nombreuses conversions sont plus complexes que cela. Par exemple, les grands entiers comme peuvent

stocker des nombres beaucoup plus grands que , et convertir un nombre comme en n'a pas beaucoup de sens sans quelques informations supplémentaires. Faire une simple conversion binaire, dans laquelle les 32 premiers bits sont jetés, ne donne pas souvent le résultat que nous espérions: Into i64 i32 2_000_000_000_000i64 i32

```
let huge = 2_000_000_000_000i64;
let smaller = huge as i32;
println!("{}", smaller); // -1454759936
```

Il existe de nombreuses options pour gérer cette situation. Selon le contexte, une telle conversion « wrapping » peut être appropriée. D'autre part, des applications telles que le traitement du signal numérique et les systèmes de contrôle peuvent souvent se contenter d'une conversion « saturante », dans laquelle les nombres supérieurs à la valeur maximale possible sont limités à ce maximum.

TryFrom et TryInto

Comme il n'est pas clair comment une telle conversion devrait se comporter, Rust n'implémente pas pour , ou toute autre conversion entre types numériques qui perdrait des informations. Au lieu de cela, implémente . et sont les cousins faillibles de et sont réciproques de la même manière; la mise en œuvre signifie que cela est également mis en œuvre. From<i64> i32 i32 TryFrom<i64> TryFrom TryInto From I nto TryFrom TryInto

Leurs définitions ne sont qu'un peu plus complexes que et . From Into

```
pub trait TryFrom<T>: Sized {
    type Error;
    fn try_from(value: T) -> Result<Self, Self::Error>;
}

pub trait TryInto<T>: Sized {
    type Error;
    fn try_into(self) -> Result<T, Self::Error>;
}
```

La méthode nous donne un , afin que nous puissions choisir quoi faire dans le cas exceptionnel, comme un nombre trop grand pour tenir dans

le type résultant: `try_into()` `Result`

```
// Saturate on overflow, rather than wrapping
let smaller: i32 = huge.try_into().unwrap_or(i32::MAX);
```

Si nous voulons également traiter le cas négatif, nous pouvons utiliser la méthode de : `unwrap_or_else()` `Result`

```
let smaller: i32 = huge.try_into().unwrap_or_else(|_| {
    if huge >= 0 {
        i32::MAX
    } else {
        i32::MIN
    }
});
```

La mise en œuvre de conversions faillibles pour vos propres types est également facile. Le type peut être aussi simple, ou aussi complexe, qu'une application particulière l'exige. La bibliothèque standard utilise une structure vide, ne fournissant aucune information au-delà du fait qu'une erreur s'est produite, car la seule erreur possible est un débordement. D'autre part, les conversions entre types plus complexes peuvent vouloir renvoyer plus d'informations : `Error`

```
impl TryInto<LinearShift> for Transform {
    type Error = TransformError;

    fn try_into(self) -> Result<LinearShift, Self::Error> {
        if !self.normalized() {
            return Err(TransformError::NotNormalized);
        }
        ...
    }
}
```

Où et relier les types avec des conversions simples, et étendre la simplicité de et les conversions avec la gestion expressive des erreurs offerte par . Ces quatre traits peuvent être utilisés ensemble pour relier de nombreux types dans une seule

caisse. `From` `Into` `TryFrom` `TryInto` `From` `Into` `Result`

ToOwned

Étant donné une référence, la façon habituelle de produire une copie propre de son référent est d'appeler , en supposant que le type implémente . Mais que se passe-t-il si vous voulez cloner un ou un ? Ce que vous voulez probablement, c'est un ou un , mais la définition de ne le permet pas : par définition, le clonage d'un doit toujours renvoyer une valeur de type , et sont non dimensionnés ; ce ne sont même pas des types qu'une fonction pourrait renvoyer.

```
clone std::clone::Clone &str &
[i32] String Vec<i32> Clone &T T str [u8]
```

Le trait fournit un moyen légèrement plus lâche de convertir une référence en une valeur possédée: std::borrow::ToOwned

```
trait ToOwned {
    type Owned: Borrow<Self>;
    fn to_owned(&self) -> Self::Owned;
}
```

Contrairement à , qui doit retourner exactement , peut retourner tout ce que vous pourriez emprunter à: le type doit implémenter . Vous pouvez emprunter à à a , donc peut implémenter , tant que implémente , afin que nous puissions copier les éléments de la tranche dans le vecteur. De même, implémente , implémente , et ainsi de

```
suite.
```

```
clone Self to_owned &Self Owned Borrow<Self> &
[T] Vec<T>
[T] ToOwned<Owned=Vec<T>> T Clone str ToOwned<Owned=String
> Path ToOwned<Owned=PathBuf>
```

Emprunter et posséder au travail : l'humble vache

Faire bon usage de Rust implique de réfléchir à des questions de propriété, comme si une fonction doit recevoir un paramètre par référence ou par valeur. Habituellement, vous pouvez choisir l'une ou l'autre approche, et le type du paramètre reflète votre décision. Mais dans certains cas, vous ne pouvez pas décider d'emprunter ou de posséder tant que le programme n'est pas en cours d'exécution; le type (pour « clone on write ») fournit un moyen de le faire. std::borrow::Cow

Sa définition est présentée ici :

```
enum Cow<'a, B: ?Sized>
    where B: ToOwned
{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

A emprunte une référence partagée à a ou possède une valeur à partir de laquelle nous pourrions emprunter une telle référence. Puisque implémente , vous pouvez appeler des méthodes dessus comme s'il s'agissait d'une référence partagée à un : si c'est , il emprunte une référence partagée à la valeur possédée ; et si c'est , il distribue simplement la référence qu'il détient. Cow B Cow Deref B Owned Borrowed

Vous pouvez également obtenir une référence modifiable à la valeur d'un 'en appelant sa méthode, qui renvoie un fichier . Si le se trouve être , appelle simplement la méthode de la référence pour obtenir sa propre copie du référent, change le en un , et emprunte une référence mutable à la valeur nouvellement détenue. Il s'agit du comportement « clone on write » auquel le nom du type fait référence. Cow to_mut &mut
B Cow Cow::Borrowed to_mut to_owned Cow Cow::Owned

De même, a une méthode qui promeut la référence à une valeur possédée, si nécessaire, puis la renvoie, déplaçant la propriété à l'appelant et consommant le dans le processus. Cow into_owned Cow

Une utilisation courante consiste à renvoyer une constante de chaîne allouée statiquement ou une chaîne calculée. Par exemple, supposons que vous deviez convertir un énumérateur d'erreur en message. La plupart des variantes peuvent être gérées avec des chaînes fixes, mais certaines d'entre elles ont des données supplémentaires qui doivent être incluses dans le message. Vous pouvez retourner un : Cow Cow<'static, str>

```
use std::path::PathBuf;
use std::borrow::Cow;
fn describe(error: &Error) -> Cow<'static, str> {
    match *error {
        Error::OutOfMemory => "out of memory".into(),
        Error::StackOverflow => "stack overflow".into(),
        Error::MachineOnFire => "machine on fire".into(),
    }
}
```

```

Error::Unfathomable => "machine bewildered".into(),
Error::NotFound(ref path) => {
    format!("file not found: {}", path.display()).into()
}
}
}

```

Ce code utilise l'implémentation de pour construire les valeurs. La plupart des bras de cette instruction renvoient une référence à une chaîne allouée statiquement. Mais lorsque nous obtenons une variante, nous utilisons pour construire un message incorporant le nom de fichier donné. Ce bras de l'instruction produit une

```
valeur. Cow Into match Cow::Borrowed NotFound format! match Cow::Owned
```

Les appels qui n'ont pas besoin de changer la valeur peuvent simplement traiter le comme un : describe Cow &str

```
println!("Disaster has struck: {}", describe(&error));
```

Les appels qui ont besoin d'une valeur possédée peuvent facilement en produire une:

```
let mut log: Vec<String> = Vec::new();
...
log.push(describe(&error).into_owned());
```

L'utilisation des aides et de ses appels reporte l'allocation jusqu'au moment où elle devient nécessaire. Cow describe

Chapitre 14. Fermetures

Sauvez l'environnement! Créez une clôture dès aujourd'hui!

—Cormac Flanagan

Le tri d'un vecteur d'entiers est facile :

```
integers.sort();
```

Il est donc triste de constater que lorsque nous voulons que certaines données soient triées, ce n'est presque jamais un vecteur d'entiers. Nous avons généralement des enregistrements d'une sorte ou d'une autre, et la méthode intégrée ne fonctionne généralement pas: `sort`

```
struct City {
    name: String,
    population: i64,
    country: String,
    ...
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort(); // error: how do you want them sorted?
}
```

Rust se plaint de ne pas mettre en œuvre . Nous devons spécifier l'ordre de tri, comme ceci: `City std::cmp::Ord`

```
/// Helper function for sorting cities by population.
fn city_population_descending(city: &City) -> i64 {
    -city.population
}

fn sort_cities(cities: &mut Vec<City>) {
    cities.sort_by_key(city_population_descending); // ok
}
```

La fonction d'assistance, , prend un enregistrement et extrait la *clé*, le champ par lequel nous voulons trier nos données. (Il renvoie un nombre négatif parce qu'il organise les nombres dans l'ordre croissant, et nous voulons un ordre décroissant: la ville la plus peuplée d'abord.) La méthode prend cette fonction clé comme paramètre. `city_population_descending` `City` `sort` `sort_by_key`

Cela fonctionne bien, mais il est plus concis d'écrire la fonction d'assistance comme une *fermeture*, une expression de fonction anonyme:

```
fn sort_cities(cities: &mut Vec<City>) {  
    cities.sort_by_key(|city| -city.population);  
}
```

La fermeture ici est . Il prend un argument et renvoie . Rust déduit le type d'argument et le type de retour de la façon dont la fermeture est utilisée. `|city| -city.population` `city -city.population`

Voici d'autres exemples de fonctionnalités de bibliothèque standard qui acceptent les fermetures :

- `Iterator` méthodes telles que `et` , pour travailler avec des données séquentielles. Nous couvrirons ces méthodes au [chapitre 15.](#) `map` `filter`
- API de threading comme `join` , qui démarre un nouveau thread système. La simultanéité consiste à déplacer le travail vers d'autres threads, et les fermetures représentent commodément les unités de travail. Nous aborderons ces fonctionnalités au [chapitre 19.](#) `thread::spawn`
- Certaines méthodes qui doivent conditionnellement calculer une valeur par défaut, comme la méthode des entrées. Cette méthode obtient ou crée une entrée dans un `HashMap` , et elle est utilisée lorsque la valeur par défaut est coûteuse à calculer. La valeur par défaut est transmise en tant que fermeture appelée uniquement si une nouvelle entrée doit être créée. `or_insert_with` `HashMap` `HashMap`

Bien sûr, les fonctions anonymes sont partout de nos jours, même dans des langages comme Java, C #, Python et C ++ qui ne les avaient pas à l'origine. À partir de maintenant, nous supposerons que vous avez déjà vu des fonctions anonymes et nous nous concentrerons sur ce qui rend les fermetures de Rust un peu différentes. Dans ce chapitre, vous apprendrez les trois types de fermetures, comment utiliser les fermetures avec des méthodes de bibliothèque standard, comment une fermeture peut « capturer » des variables dans son étendue, comment écrire vos propres fonctions et méthodes qui prennent les fermetures comme arguments, et comment stocker les fermetures pour une utilisation ultérieure comme rappels. Nous expliquerons également comment les fermetures Rust sont mises en œuvre et pourquoi elles sont plus rapides que prévu.

Capture de variables

Une fermeture peut utiliser des données appartenant à une fonction en-globante. Par exemple:

```
/// Sort by any of several different statistics.  
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {  
    cities.sort_by_key(|city| -city.get_statistic(stat));  
}
```

La fermeture utilise ici `stat`, qui appartient à la fonction d'enceinte, `.get_statistic`. On dit que la fermeture « capture ». C'est l'une des caractéristiques classiques des fermetures, donc naturellement, Rust le soutient; mais dans Rust, cette fonctionnalité est livrée avec une ficelle attachée. `stat sort_by_statistic stat`

Dans la plupart des langues avec fermetures, le ramassage des ordures joue un rôle important. Par exemple, considérez ce code JavaScript :

```
// Start an animation that rearranges the rows in a table of cities.  
function startSortingAnimation(cities, stat) {  
    // Helper function that we'll use to sort the table.  
    // Note that this function refers to stat.  
    function keyfn(city) {  
        return city.get_statistic(stat);  
    }  
  
    if (pendingSort)  
        pendingSort.cancel();  
  
    // Now kick off an animation, passing keyfn to it.  
    // The sorting algorithm will call keyfn later.  
    pendingSort = new SortingAnimation(cities, keyfn);  
}
```

La fermeture est stockée dans le nouvel objet. Il est destiné à être appelé après les retours. Maintenant, normalement, lorsqu'une fonction est renvoyée, toutes ses variables et tous ses arguments sortent de la portée et sont ignorés. Mais ici, le moteur JavaScript doit rester d'une manière ou d'une autre, puisque la fermeture l'utilise. La plupart des moteurs JavaScript le font en allouant dans le tas et en laissant le garbage collector le récupérer plus

tard. `keyfn SortingAnimation startSortingAnimation stat stat`

Rust n'a pas de ramassage des ordures. Comment cela fonctionnera-t-il? Pour répondre à cette question, nous allons examiner deux exemples.

Fermetures qui empruntent

Tout d'abord, répétons l'exemple d'ouverture de cette section:

```
// Sort by any of several different statistics.
fn sort_by_statistic(cities: &mut Vec<City>, stat: Statistic) {
    cities.sort_by_key(|city| -city.get_statistic(stat));
}
```

Dans ce cas, lorsque Rust crée la fermeture, il emprunte automatiquement une référence à . Cela va de soi : la fermeture fait référence à , elle doit donc y avoir une référence. stat stat

Le reste est simple. La clôture est assujettie aux règles sur les emprunts et les durées de vie que nous avons décrites au [chapitre 5](#). En particulier, puisque la fermeture contient une référence à , Rust ne la laissera pas survivre . Étant donné que la fermeture n'est utilisée que pendant le tri, cet exemple convient parfaitement. stat stat

En bref, Rust assure la sécurité en utilisant des durées de vie au lieu de la collecte des ordures. Le chemin de Rust est plus rapide: même une allocation GC rapide sera plus lente que le stockage sur la pile, comme rust le fait dans ce cas. stat

Fermetures qui volent

Le deuxième exemple est plus délicat :

```
use std::thread;

fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };

    thread::spawn(|| {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

Cela ressemble un peu plus à ce que faisait notre exemple JavaScript : prendre une fermeture et l'appeler dans un nouveau thread système. Notez qu'il s'agit de la liste d'arguments vide de la fermeture. thread::spawn ||

Le nouveau thread s'exécute en parallèle avec l'appelant. Lorsque la fermeture revient, le nouveau thread se ferme. (La valeur de retour de la fermeture est renvoyée au thread appelant en tant que valeur. Nous couvrirons cela au [chapitre 19](#).) JoinHandle

Encore une fois, la fermeture contient une référence à . Mais cette fois, Rust ne peut pas garantir que la référence est utilisée en toute sécurité. Rust rejette donc ce programme : key_fn stat

```
error: closure may outlive the current function, but it borrows `stat`,
      which is owned by the current function
|
33 | let key_fn = |city: &City| -> i64 { -city.get_statistic(stat) };
|          ^^^^^^^^^^^^^^^^^^^^^^
|          |
|          |                                     `stat` is borro
|          may outlive borrowed value `stat`
```

En fait, il y a deux problèmes ici, parce qu'il est également partagé de manière dangereuse. Tout simplement, on ne peut pas s'attendre à ce que le nouveau thread créé par finisse son travail avant et soit détruit à la fin de la fonction. cities thread::spawn cities stat

La solution aux deux problèmes est la même : dire à Rust de *déménager* et d'entrer dans les fermetures qui les utilisent au lieu d'emprunter des références à celles-ci. cities stat

```
fn start_sorting_thread(mut cities: Vec<City>, stat: Statistic)
    -> thread::JoinHandle<Vec<City>>
{
    let key_fn = move |city: &City| -> i64 { -city.get_statistic(stat) }

    thread::spawn(move || {
        cities.sort_by_key(key_fn);
        cities
    })
}
```

La seule chose que nous avons changée est d'ajouter le mot-clé avant chacune des deux fermetures. Le mot-clé indique à Rust qu'une fermeture n'emprunte pas les variables qu'elle utilise : elle les vole. move move

La première fermeture, , prend possession de . Ensuite, la deuxième fermeture prend possession des deux et . key_fn stat cities key_fn

Rust offre donc deux façons pour les fermetures d'obtenir des données à partir des scopes d'enclos: les déménagements et les emprunts. En réalité,

il n'y a rien de plus à dire que cela; les fermetures suivent les mêmes règles concernant les déménagements et les emprunts que celles que nous avons déjà abordées aux chapitres [4](#) et [5](#). Quelques exemples :

- Comme partout ailleurs dans la langue, si une fermeture est une valeur de type copiable, comme , elle copie la valeur à la place. Donc, s'il s'agissait d'un type copiable, nous pourrions continuer à l'utiliser même après avoir créé une fermeture qui
`l'utilise. move i32 Statistic stat move`
- Les valeurs de types non pouvant être copiés, comme , sont vraiment déplacées : le code précédent est transféré vers le nouveau thread, par le biais de la fermeture. Rust ne nous permettrait pas d'accéder par nom après la création de la
`fermeture. Vec<City> cities move cities`
- En l'occurrence, ce code n'a pas besoin d'être utilisé après le point où la fermeture le déplace. Si nous le faisions, cependant, la solution de contournement serait facile: nous pourrions dire à Rust de cloner et de stocker la copie dans une variable différente. La fermeture ne volerait qu'une seule des copies, quelle que soit celle à laquelle elle se réfère.
`cities cities`

Nous obtenons quelque chose d'important en acceptant les règles strictes de Rust : la sécurité des fils. C'est précisément parce que le vecteur est déplacé, plutôt que d'être partagé entre les threads, que nous savons que l'ancien thread ne libérera pas le vecteur pendant que le nouveau thread le modifie.

Types de fonction et de fermeture

Tout au long de ce chapitre, nous avons vu des fonctions et des fermetures utilisées comme valeurs. Naturellement, cela signifie qu'ils ont des types. Par exemple:

```
fn city_population_descending(city: &City) -> i64 {  
    -city.population  
}
```

Cette fonction prend un argument (a) et renvoie un . Il a le type

```
. &City i64 fn(&City) -> i64
```

Vous pouvez faire toutes les mêmes choses avec des fonctions qu'avec d'autres valeurs. Vous pouvez les stocker dans des variables. Vous pouvez

utiliser toute la syntaxe Rust habituelle pour calculer les valeurs de fonction :

```
let my_key_fn: fn(&City) -> i64 =  
    if user.prefs.by_population {  
        city_population_descending  
    } else {  
        city_monster_attack_risk_descending  
    };  
  
cities.sort_by_key(my_key_fn);
```

Les structures peuvent avoir des champs typés de fonction. Les types génériques comme peuvent stocker des scads de fonctions, tant qu'ils partagent tous le même type. Et les valeurs de fonction sont minuscules : une valeur est l'adresse mémoire du code machine de la fonction, tout comme un pointeur de fonction en C++. `Vec fn fn`

Une fonction peut prendre une autre fonction comme argument. Par exemple:

```
/// Given a list of cities and a test function,  
/// return how many cities pass the test.  
fn count_selected_cities(cities: &Vec<City>,  
                         test_fn: fn(&City) -> bool) -> usize  
{  
    let mut count = 0;  
    for city in cities {  
        if test_fn(city) {  
            count += 1;  
        }  
    }  
    count  
}  
  
/// An example of a test function. Note that the type of  
/// this function is `fn(&City) -> bool`, the same as  
/// the `test_fn` argument to `count_selected_cities`.  
fn has_monster_attacks(city: &City) -> bool {  
    city.monster_attack_risk > 0.0  
}  
  
// How many cities are at risk for monster attack?  
let n = count_selected_cities(&my_cities, has_monster_attacks);
```

Si vous êtes familier avec les pointeurs de fonction en C/C++, vous verrez que les valeurs de fonction de Rust sont exactement la même chose.

Après tout cela, il peut être surprenant que les fermetures n'aient *pas* le même type que les fonctions:

```
let limit = preferences.acceptable_monster_risk();
let n = count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // error: type mismatch
```

Le deuxième argument provoque une erreur de type. Pour prendre en charge les fermetures, nous devons modifier la signature de type de cette fonction. Il doit ressembler à ceci:

```
fn count_selected_cities<F>(cities: &Vec<City>, test_fn: F) -> usize
    where F: Fn(&City) -> bool
{
    let mut count = 0;
    for city in cities {
        if test_fn(city) {
            count += 1;
        }
    }
    count
}
```

Nous n'avons changé que la signature de type de , pas le corps. La nouvelle version est générique. Il faut un de n'importe quel type aussi longtemps que met en œuvre le trait spécial . Ce trait est automatiquement implémenté par toutes les fonctions et la plupart des fermetures qui prennent un seul comme argument et renvoient une valeur booléenne

```
:count_selected_cities test_fn F F Fn(&City) -> bool &City

fn(&City) -> bool      // fn type (functions only)
Fn(&City) -> bool      // Fn trait (both functions and closures)
```

Cette syntaxe spéciale est intégrée au langage. Le et le type de retour sont facultatifs; s'il est omis, le type de retour est . -> ()

La nouvelle version de accepte soit une fonction, soit une fermeture

```
:count_selected_cities
```

```
count_selected_cities(
    &my_cities,
```

```

    has_monster_attacks); // ok

count_selected_cities(
    &my_cities,
    |city| city.monster_attack_risk > limit); // also ok

```

Pourquoi notre première tentative n'a-t-elle pas fonctionné ? Eh bien, une fermeture est appellable, mais ce n'est pas un . La fermeture a son propre type qui n'est pas un type. fn |city| city.monster_attack_risk > limit fn

En fait, chaque fermeture que vous écrivez a son propre type, car une fermeture peut contenir des données : des valeurs empruntées ou volées dans des étendues englobantes. Il peut s'agir de n'importe quel nombre de variables, dans n'importe quelle combinaison de types. Ainsi, chaque fermeture a un type ad hoc créé par le compilateur, suffisamment grand pour contenir ces données. Il n'y a pas deux fermetures qui ont exactement le même type. Mais chaque fermeture met en œuvre un trait; la fermeture dans notre exemple implémente . Fn Fn(&City) -> i64

Étant donné que chaque fermeture a son propre type, le code qui fonctionne avec les fermetures doit généralement être générique, comme . C'est un peu maladroit d'épeler les types génériques à chaque fois, mais pour voir les avantages de cette conception, lisez la suite. count_selected_cities

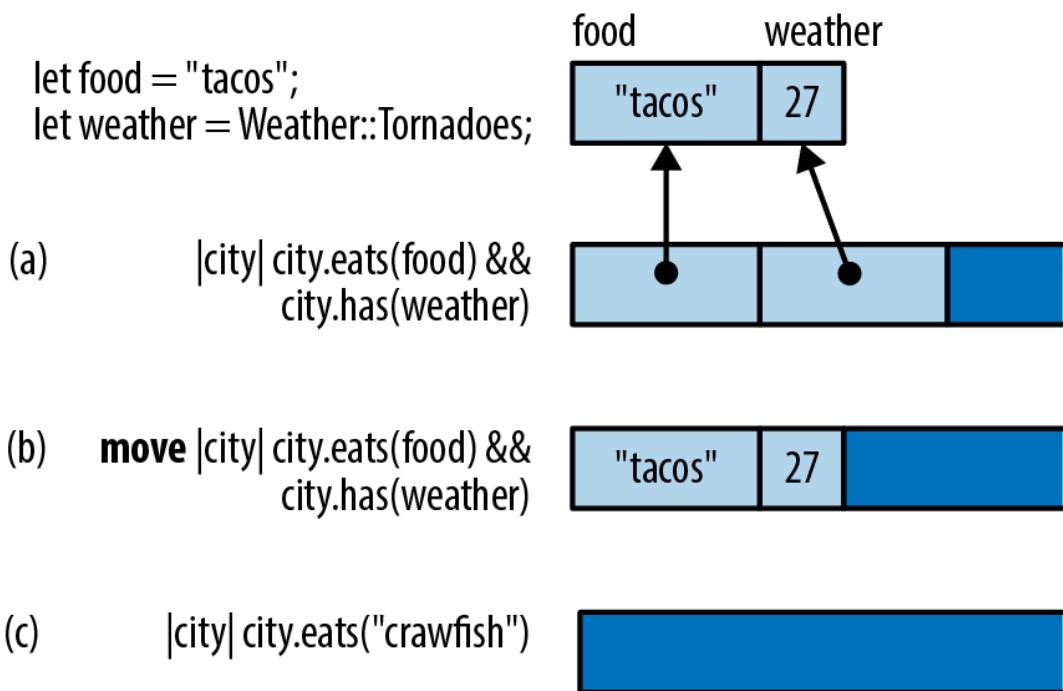
Performance de fermeture

Les fermetures de Rust sont conçues pour être rapides : plus rapides que les pointeurs de fonction, suffisamment rapides pour que vous puissiez les utiliser même dans un code chaud et sensible aux performances. Si vous êtes familier avec les lambdas C++, vous constaterez que les fermetures Rust sont tout aussi rapides et compactes, mais plus sûres.

Dans la plupart des langues, les fermetures sont allouées dans le tas, distribuées dynamiquement et les ordures collectées. Ainsi, créer, appeler et collecter chacun d'eux coûte un tout petit peu de temps CPU supplémentaire. Pire encore, les fermetures ont tendance à *exclure les contours*, une technique clé utilisée par les compilateurs pour éliminer la surcharge d'appel de fonction et permettre une série d'autres optimisations. Tout compte fait, les fermetures sont suffisamment lentes dans ces langues pour qu'il puisse être utile de les retirer manuellement des boucles internes serrées.

Les fermetures antirouille n'ont aucun de ces inconvénients de performance. Ce ne sont pas des ordures ramassées. Comme tout le reste dans Rust, ils ne sont pas alloués sur le tas à moins que vous ne les mettiez dans un , ou un autre conteneur. Et puisque chaque fermeture a un type distinct, chaque fois que le compilateur Rust connaît le type de fermeture que vous appelez, il peut insérer le code pour cette fermeture particulière. Cela permet d'utiliser des fermetures en boucles serrées, et les programmes Rust le font souvent, avec enthousiasme, comme vous le verrez au [chapitre 15](#). Box Vec

[La figure 14-1](#) montre comment les fermetures Rust sont disposées en mémoire. En haut de la figure, nous montrons quelques variables locales auxquelles nos fermetures se référeront : une chaîne et un enum simple, dont la valeur numérique se trouve être 27. food weather



Graphique 14-1. Disposition des fermetures en mémoire

La fermeture (a) utilise les deux variables. Apparemment, nous recherchons des villes qui ont à la fois des tacos et des tornades. En mémoire, cette fermeture ressemble à une petite structure contenant des références aux variables qu'elle utilise.

Notez qu'il ne contient pas de pointeur vers son code ! Ce n'est pas nécessaire : tant que Rust connaît le type de fermeture, il sait quel code exécuter lorsque vous l'appelez.

La fermeture (b) est exactement la même, sauf qu'il s'agit d'une fermeture, elle contient donc des valeurs au lieu de références. `move`

La fermeture (c) n'utilise aucune variable de son environnement. La structure est vide, donc cette fermeture ne prend aucune mémoire du

tout.

Comme le montre la figure, ces fermetures ne prennent pas beaucoup de place. Mais même ces quelques octets ne sont pas toujours nécessaires dans la pratique. Souvent, le compilateur peut insérer tous les appels à une fermeture, puis même les petites structures illustrées dans cette figure sont optimisées.

Dans [« Rappels »](#), nous allons montrer comment allouer des fermetures dans le tas et les appeler dynamiquement, en utilisant des objets de trait. C'est un peu plus lent, mais c'est toujours aussi rapide que n'importe quelle autre méthode d'objet de trait.

Fermetures et sécurité

Tout au long du chapitre jusqu'à présent, nous avons parlé de la façon dont Rust s'assure que les fermetures respectent les règles de sécurité du langage lorsqu'elles empruntent ou déplacent des variables du code environnant. Mais il y a d'autres conséquences qui ne sont pas exactement évidentes. Dans cette section, nous expliquerons un peu plus ce qui se passe lorsqu'une fermeture supprime ou modifie une valeur capturée.

Fermetures qui tuent

Nous avons vu des fermetures qui empruntent des valeurs et des fermetures qui les volent; ce n'était qu'une question de temps avant qu'ils aillent jusqu'au bout.

Bien sûr, *tuer* n'est pas vraiment la bonne terminologie. Dans Rust, nous *supprimons* des valeurs. La façon la plus simple de le faire est d'appeler : `drop()`

```
let my_str = "hello".to_string();
let f = || drop(my_str);
```

Quand est appelé, est abandonné. `f my_str`

Alors, que se passe-t-il si nous l'appelons deux fois?

```
f();
f();
```

Réfléchissons-y. La première fois que nous appelons , il tombe , ce qui signifie que la mémoire où la chaîne est stockée est libérée, renvoyée au sys-

tème. La deuxième fois que nous appelons, la même chose se produit. C'est un *double gratuit*, une erreur classique dans la programmation C ++ qui déclenche un comportement indéfini. `f my_str f`

Laisser tomber un deux fois serait une idée tout aussi mauvaise dans Rust. Heureusement, Rust ne peut pas être trompé si facilement: `String`

```
f(); // ok  
f(); // error: use of moved value
```

Rust sait que cette fermeture ne peut pas être appelée deux fois.

Une fermeture qui ne peut être appelée qu'une seule fois peut sembler une chose assez extraordinaire, mais nous avons parlé tout au long de ce livre de la propriété et des vies. L'idée que les valeurs sont épuisées (c'est-à-dire déplacées) est l'un des concepts fondamentaux de Rust. Cela fonctionne de la même manière avec les fermetures qu'avec tout le reste.

FnOnce

Essayons une fois de plus de tromper Rust pour qu'il en laisse tomber deux fois. Cette fois, nous allons utiliser cette fonction générique : `String`

```
fn call_twice<F>(closure: F) where F: Fn() {  
    closure();  
    closure();  
}
```

Cette fonction générique peut être passée toute fermeture qui implémente le trait : c'est-à-dire les fermetures qui ne prennent aucun argument et renvoient . (Comme pour les fonctions, le type de retour peut être omis s'il est ; est un raccourci pour .) `Fn() () () Fn() Fn() -> ()`

Maintenant, que se passe-t-il si nous passons notre fermeture dangereuse à cette fonction générique?

```
let my_str = "hello".to_string();  
let f = || drop(my_str);  
call_twice(f);
```

Encore une fois, la fermeture tombera quand elle sera appelée. L'appeler deux fois serait un double gratuit. Mais encore une fois, Rust n'est pas dupe : `my_str`

```

error: expected a closure that implements the `Fn` trait, but
      this closure only implements `FnOnce`  

|  

8 | let f = || drop(my_str);
|     ^^^^^^-----^  

|     |  

|     |     closure is `FnOnce` because it moves the variable  

|     |     out of its environment  

|     |     this closure implements `FnOnce`, not `Fn`  

9 | call_twice(f);
| ----- the requirement to implement `Fn` derives from here

```

Ce message d'erreur nous en dit plus sur la façon dont Rust gère les « fermetures qui tuent ». Ils auraient pu être complètement bannis de la langue, mais les fermetures de nettoyage sont parfois utiles. Donc, au lieu de cela, Rust restreint leur utilisation. Les fermetures qui suppriment des valeurs, comme `, ne` sont pas autorisées à avoir `.` Ils sont, littéralement, pas du tout. Ils mettent en œuvre un trait moins puissant, le trait des fermetures que l'on peut appeler une fois. `f Fn Fn FnOnce`

La première fois que vous appelez une fermeture, *la fermeture elle-même est épuisée*. C'est comme si les deux traits, `et ,` étaient définis comme ceci: `FnOnce Fn FnOnce`

```

// Pseudocode for `Fn` and `FnOnce` traits with no arguments.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}

```

Tout comme une expression arithmétique comme `est` est un raccourci pour un appel de méthode, `,` Rust traite comme un raccourci pour l'une des deux méthodes de trait montrées dans l'exemple précédent. Pour une fermeture, s'étend à `.` Cette méthode prend par référence, de sorte que la fermeture n'est pas déplacée. Mais si la fermeture n'est sûre à appeler qu'une seule fois, elle s'étend à `.` Cette méthode prend par valeur, de sorte que la fermeture est épuisée. `a + b Add::add(a,`
`b) closure() Fn closure() closure.call() self closure() clo`
`sure.call_once() self`

Bien sûr, nous avons délibérément semé le trouble ici en utilisant `.` En pratique, vous vous retrouverez principalement dans cette situation par

accident. Cela n'arrive pas souvent, mais de temps en temps, vous écrirez du code de fermeture qui utilise involontairement une valeur: drop()

```
let dict = produce_glossary();
let debug_dump_dict = || {
    for (key, value) in dict { // oops!
        println!("{}:{} - {}:", key, value);
    }
};
```

Ensuite, lorsque vous appelez plus d'une fois, vous obtiendrez un message d'erreur comme celui-ci : debug_dump_dict()

```
error: use of moved value: `debug_dump_dict`
|
19 |     debug_dump_dict();
|----- `debug_dump_dict` moved due to this call
20 |     debug_dump_dict();
|     ^^^^^^^^^^^^^^^^ value used here after move
|
note: closure cannot be invoked more than once because it moves the vari
`dict` out of its environment
|
13 |         for (key, value) in dict {
|             ^^^
```

Pour déboguer cela, nous devons comprendre pourquoi la fermeture est un fichier . Quelle valeur est utilisée ici? Le compilateur souligne utilement que c'est , ce qui dans ce cas est le seul auquel nous faisons référence. Ah, il y a le bug : on l'épuise en itérant dessus directement. Nous devrions tourner en boucle sur , plutôt que simplement – pour accéder aux valeurs par référence : FnOnce dict dict &dict dict

```
let debug_dump_dict = || {
    for (key, value) in &dict { // does not use up dict
        println!("{}:{} - {}:", key, value);
    }
};
```

Cela corrige l'erreur; la fonction est maintenant un et peut être appelée n'importe quel nombre de fois. Fn

FnMut

Il existe un autre type de fermeture, celui qui contient des données ou des références modifiables. `mut`

Rust considère que les valeurs non-sûres peuvent être partagées entre les threads. Mais il ne serait pas sûr de partager des non-fermetures qui contiennent des données : appeler une telle fermeture à partir de plusieurs threads pourrait conduire à toutes sortes de conditions de course car plusieurs threads essaient de lire et d'écrire les mêmes données en même temps. `mut` `mut` `mut`

Par conséquent, Rust a une autre catégorie de fermeture, , la catégorie de fermetures qui écrivent. les fermetures sont appelées par référence, comme si elles étaient définies comme suit : `FnMut` `FnMut` `mut`

```
// Pseudocode for `Fn`, `FnMut`, and `FnOnce` traits.
trait Fn() -> R {
    fn call(&self) -> R;
}

trait FnMut() -> R {
    fn call_mut(&mut self) -> R;
}

trait FnOnce() -> R {
    fn call_once(self) -> R;
}
```

Toute fermeture qui nécessite l'accès à une valeur, mais ne supprime aucune valeur, est une fermeture. Par exemple: `mut FnMut`

```
let mut i = 0;
let incr = || {
    i += 1; // incr borrows a mut reference to i
    println!("Ding! i is now: {}", i);
};
call_twice(incr);
```

La façon dont nous avons écrit, cela nécessite un . Puisque est un et non un , ce code ne parvient pas à compiler. Il y a une solution facile, cependant. Pour comprendre le correctif, prenons un peu de recul et résumons ce que vous avez appris sur les trois catégories de fermetures
Rust. `call_twice Fn incr FnMut Fn`

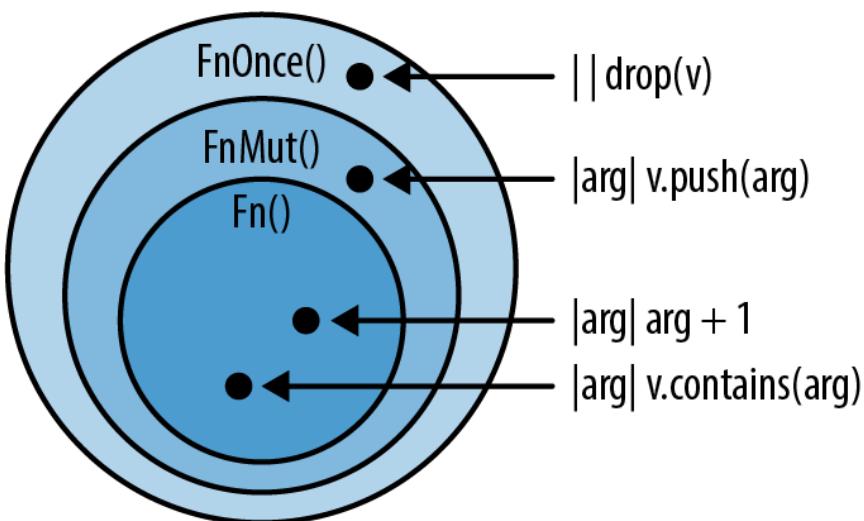
- `Fn` est la famille de fermetures et de fonctions que vous pouvez appeler plusieurs fois sans restriction. Cette catégorie la plus élevée com-

prend également toutes les fonctions. fn

- FnMut est la famille des fermetures qui peuvent être appelées plusieurs fois si la fermeture elle-même est déclarée .mut
- FnOnce est la famille de fermetures qui peuvent être appelées une fois, si l'appelant est propriétaire de la fermeture.

Chaque répond aux exigences de , et chaque répond aux exigences de . Comme le montre [la figure 14-2](#), il ne s'agit pas de trois catégories distinctes. Fn FnMut FnMut FnOnce

Au lieu de cela, est un sous-trait de , qui est un sous-trait de . Cela fait la catégorie la plus exclusive et la plus puissante. et sont des catégories plus larges qui incluent les fermetures avec des restrictions d'utilisation. Fn() FnMut() FnOnce() Fn FnMut FnOnce



Graphique 14-2. Diagramme de Venn des trois catégories de fermeture

Maintenant que nous avons organisé ce que nous savons, il est clair que pour accepter la plus grande famille de fermetures, notre fonction devrait vraiment accepter toutes les fermetures, comme ceci: call_twice FnMut

```
fn call_twice<F>(mut closure: F) where F: FnMut() {  
    closure();  
    closure();  
}
```

La limite sur la première ligne était , et maintenant c'est . Avec ce changement, nous acceptons toujours toutes les fermetures, et nous pouvons également utiliser sur les fermetures qui mutent les données: F :

```
Fn() F: FnMut() Fn call_twice
```

```
let mut i = 0;  
call_twice(|| i += 1); // ok!
```

```
assert_eq!(i, 2);
```

Copier et cloner pour les fermetures

Tout comme Rust détermine automatiquement quelles fermetures ne peuvent être appelées qu'une seule fois, il peut déterminer quelles fermetures peuvent être mises en œuvre et , et lesquelles ne le peuvent pas. `Copy` `Clone`

Comme nous l'avons expliqué précédemment, les fermetures sont représentées sous la forme de structures qui contiennent soit les valeurs (pour les fermetures), soit des références aux valeurs (pour les non-fermetures) des variables qu'elles capturent. Les règles pour et sur les fermetures sont tout comme les règles pour les structures régulières. Une non-fermeture qui ne mute pas les variables ne contient que des références partagées, qui sont à la fois et , de sorte que la fermeture est à la fois et aussi bien

```
:move move Copy Clone Copy Clone move Clone Copy Clone Copy
```

```
let y = 10;
let add_y = |x| x + y;
let copy_of_add_y = add_y; // This closure is `Copy`, so...
assert_eq!(add_y(copy_of_add_y(22)), 42); // ... we can call both.
```

D'autre part, une non-fermeture qui *mute* des valeurs a des références mutables dans sa représentation interne. Les références mutables ne sont ni ni , donc pas plus qu'une fermeture qui les utilise : `move` `Clone` `Copy`

```
let mut x = 0;
let mut add_to_x = |n| { x += n; x };

let copy_of_add_to_x = add_to_x; // this moves, rather than copies
assert_eq!(add_to_x(copy_of_add_to_x(1)), 2); // error: use of moved value
```

Pour une fermeture, les règles sont encore plus simples. Si tout ce qu'une fermeture capture est , c'est . Si tout ce qu'il capture est , c'est . Par exemple: `move` `move` `Copy` `Copy` `Clone` `Clone`

```
let mut greeting = String::from("Hello, ");
let greet = move |name| {
    greeting.push_str(name);
    println!("{} ", greeting);
};
```

```
greet.clone(("Alfred");
greet.clone(("Bruce");
```

Cette syntaxe est un peu bizarre, mais cela signifie simplement que nous clonons la fermeture et appelons ensuite le clone. Les résultats de ce programme sont les suivants : .clone()(...)

```
Hello, Alfred
Hello, Bruce
```

Lorsqu'il est utilisé dans , il est déplacé dans la structure qui représente en interne, car il s'agit d'une fermeture. Ainsi, lorsque nous clonons, tout ce qui s'y trouve est également cloné. Il existe deux copies de , qui sont chacune modifiées séparément lorsque les clones de sont appelés. Ce n'est pas si utile en soi, mais lorsque vous devez passer la même fermeture dans plus d'une fonction, cela peut être très

```
utile. greeting greet greet move greet greeting greet
```

Rappels

Beaucoup de bibliothèques utilisent des *rappels* dans le cadre de leur API : des fonctions fournies par l'utilisateur, pour que la bibliothèque appelle plus tard. En fait, vous avez déjà vu des API comme celle-ci dans ce livre. Au [chapitre 2](#), nous avons utilisé le framework pour écrire un serveur Web simple. Une partie importante de ce programme était le routeur, qui ressemblait à ceci: actix-web

```
App::new()
    .route("/", web::get().to(get_index))
    .route("/gcd", web::post().to(post_gcd))
```

Le but du routeur est d'acheminer les demandes entrantes d'Internet vers le bit de code Rust qui gère ce type particulier de demande. Dans cet exemple, et étaient les noms des fonctions que nous avons déclarées ailleurs dans le programme, en utilisant le mot-clé. Mais nous aurions pu adopter des fermetures à la place, comme ceci: get_index post_gcd fn

```
App::new()
    .route("/", web::get().to(|| {
        HttpResponse::Ok()
            .content_type("text/html")
            .body("<title>GCD Calculator</title>..."))
    }))
```

```

.route("/gcd", web::post()).to(|form: web::Form<GcdParameters>| {
    HttpResponse::Ok()
        .content_type("text/html")
        .body(format!("The GCD of {} and {} is {}.", form.n, form.m, gcd(form.n, form.m)))
})

```

C'est parce qu'il a été écrit pour accepter n'importe quel thread-safe comme argument. `actix-web Fn`

Comment pouvons-nous le faire dans nos propres programmes? Essayons d'écrire notre propre routeur très simple à partir de zéro, sans utiliser de code de . Nous pouvons commencer par déclarer quelques types pour représenter les requêtes et les réponses HTTP : `actix-web`

```

struct Request {
    method: String,
    url: String,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

struct Response {
    code: u32,
    headers: HashMap<String, String>,
    body: Vec<u8>
}

```

Maintenant, le travail d'un routeur consiste simplement à stocker une table qui mappe les URL aux rappels afin que le bon rappel puisse être appelé à la demande. (Par souci de simplicité, nous n'autoriserons les utilisateurs à créer que des itinéraires qui correspondent à une seule URL exacte.)

```

struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

impl<C> BasicRouter<C> where C: Fn(&Request) -> Response {
    /// Create an empty router.
    fn new() -> BasicRouter<C> {
        BasicRouter { routes: HashMap::new() }
    }

    /// Add a route to the router.
    fn add_route(&mut self, url: &str, callback: C) {

```

```

        self.routes.insert(url.to_string(), callback);
    }
}

```

Malheureusement, nous avons fait une erreur. L'avez-vous remarqué?

Ce routeur fonctionne bien tant que nous n'y ajoutons qu'une seule route:

```

let mut router = BasicRouter::new();
router.add_route("/", |_| get_form_response());

```

Cela compile et s'exécute. Malheureusement, si nous ajoutons un autre itinéraire:

```

router.add_route("/gcd", |req| get_gcd_response(req));

```

puis nous obtenons des erreurs:

```

error: mismatched types
|
41 |     router.add_route("/gcd", |req| get_gcd_response(req));
|                                         ^^^^^^^^^^^^^^^^^^^^^^^^^^
|
|                                         expected closure, found a different cl...
|
= note: expected type `#[closure@closures_bad_router.rs:40:27: 40:50]`  

         found type `#[closure@closures_bad_router.rs:41:30: 41:57]`  

note: no two closures, even if identical, have the same type  

help: consider boxing your closure and/or using it as a trait object

```

Notre erreur était dans la façon dont nous avons défini le type: BasicRouter

```

struct BasicRouter<C> where C: Fn(&Request) -> Response {
    routes: HashMap<String, C>
}

```

Nous avons involontairement déclaré que chacun a un seul type de rappel , et tous les rappels dans le sont de ce type. De retour dans [« Lequel utiliser »](#), nous avons montré un type qui avait le même problème: BasicRouter C HashMap Salad

```

struct Salad<V: Vegetable> {
    veggies: Vec<V>
}

```

La solution ici est la même que pour la salade: puisque nous voulons prendre en charge une variété de types, nous devons utiliser des boîtes et des objets de trait:

```
type BoxedCallback = Box<dyn Fn(&Request) -> Response>;  
  
struct BasicRouter {  
    routes: HashMap<String, BoxedCallback>  
}
```

Chaque boîte peut contenir un type de fermeture différent, de sorte qu'une seule peut contenir toutes sortes de rappels. Notez que le paramètre type a disparu. HashMap C

Cela nécessite quelques ajustements aux méthodes:

```
impl BasicRouter {  
    // Create an empty router.  
    fn new() -> BasicRouter {  
        BasicRouter { routes: HashMap::new() }  
    }  
  
    // Add a route to the router.  
    fn add_route<C>(&mut self, url: &str, callback: C)  
        where C: Fn(&Request) -> Response + 'static  
    {  
        self.routes.insert(url.to_string(), Box::new(callback));  
    }  
}
```

NOTE

Notez les deux limites dans la signature de type pour : un trait particulier et la durée de vie. La rouille nous fait ajouter cette liaison. Sans cela, l'appel à serait une erreur, car il n'est pas sûr de stocker une fermeture si elle contient des références empruntées à des variables qui sont sur le point de sortir de la portée. C add_route Fn 'static 'static Box::new(callback)

Enfin, notre routeur simple est prêt à gérer les demandes entrantes:

```
impl BasicRouter {  
    fn handle_request(&self, request: &Request) -> Response {  
        match self.routes.get(&request.url) {  
            None => not_found_response(),  
            Some(callback) => callback(request)  
        }  
    }  
}
```

```
    }  
}
```

Au prix d'une certaine flexibilité, nous pourrions également écrire une version plus économique en espace de ce routeur qui, plutôt que de stocker des objets traits, utilise des *pointeurs de fonction ou des types*. Ces types, tels que , agissent beaucoup comme des fermetures: fn fn(u32) -> u32

```
fn add_ten(x: u32) -> u32 {  
    x + 10  
}  
  
let fn_ptr: fn(u32) -> u32 = add_ten;  
let eleven = fn_ptr(1); //11
```

En fait, les fermetures qui ne capturent rien de leur environnement sont identiques aux pointeurs de fonction, car elles n'ont pas besoin de contenir d'informations supplémentaires sur les variables capturées. Si vous spécifiez le type approprié, soit dans une liaison, soit dans une signature de fonction, le compilateur vous permet de les utiliser de cette façon : fn

```
let closure_ptr: fn(u32) -> u32 = |x| x + 1;  
let two = closure_ptr(1); // 2
```

Contrairement à la capture des fermetures, ces pointeurs de fonction ne prennent qu'un seul . usize

Une table de routage contenant des pointeurs de fonction ressemblerait à ceci :

```
struct FnPointerRouter {  
    routes: HashMap<String, fn(&Request) -> Response>  
}
```

Ici, le ne stocke qu'un seul par , et critiquement, il n'y a pas de . Mis à part le lui-même, il n'y a pas d'allocation dynamique du tout. Bien sûr, les méthodes doivent également être ajustées: HashMap usize String Box HashMap

```
impl FnPointerRouter {  
    // Create an empty router.  
    fn new() -> FnPointerRouter {  
        FnPointerRouter { routes: HashMap::new() }  
    }
```

```
// Add a route to the router.  
fn add_route(&mut self, url: &str, callback: fn(&Request) -> Response)  
{  
    self.routes.insert(url.to_string(), callback);  
}  
}
```

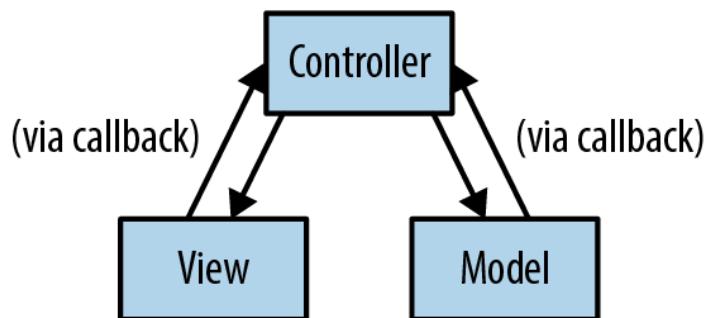
Comme le montre [la figure 14-1](#), les fermetures ont des types uniques car chacune capture des variables différentes, de sorte qu'entre autres choses, elles ont chacune une taille différente. S'ils ne capturent rien, cependant, il n'y a rien à stocker. En utilisant des pointeurs dans les fonctions qui prennent des rappels, vous pouvez limiter un appelant à utiliser uniquement ces fermetures non capturantes, ce qui permet de gagner en performance et en flexibilité dans le code à l'aide de rappels au détriment de la flexibilité pour les utilisateurs de votre API. fn

Utilisation efficace des fermetures

Comme nous l'avons vu, les fermetures de Rust sont différentes des fermetures dans la plupart des autres langues. La plus grande différence est que dans les langues avec GC, vous pouvez utiliser des variables locales dans une fermeture sans avoir à penser aux durées de vie ou à la propriété. Sans GC, les choses sont différentes. Certains modèles de conception courants en Java, C# et JavaScript ne fonctionneront pas dans Rust sans modifications.

Par exemple, prenez le modèle de conception Model-View-Controller (MVC en abrégé), illustré à [la figure 14-3](#). Pour chaque élément d'une interface utilisateur, une infrastructure MVC crée trois objets : un *modèle* représentant l'état de cet élément d'interface utilisateur, une *vue* responsable de son apparence et un *contrôleur* qui gère l'interaction utilisateur. D'innombrables variantes de MVC ont été implémentées au fil des ans, mais l'idée générale est que trois objets répartissent les responsabilités de l'interface utilisateur d'une manière ou d'une autre.

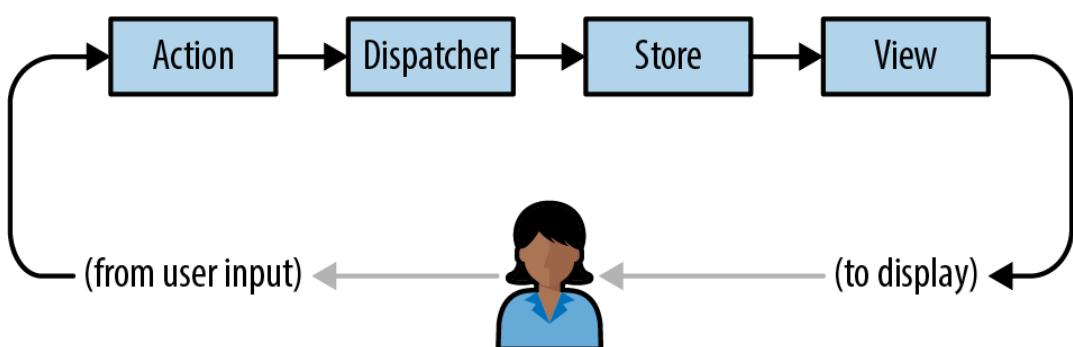
Voici le problème. En règle générale, chaque objet a une référence à l'un des autres ou aux deux, directement ou par le biais d'un rappel, comme illustré à [la figure 14-3](#). Chaque fois que quelque chose arrive à l'un des objets, il avertit les autres, de sorte que tout se met à jour rapidement. La question de savoir quel objet « possède » les autres ne se pose jamais.



Graphique 14-3. Modèle de conception Model-View-Controller

Vous ne pouvez pas implémenter ce modèle dans Rust sans apporter quelques modifications. La propriété doit être explicite et les cycles de référence doivent être éliminés. Le modèle et le contrôleur ne peuvent pas avoir de références directes l'un à l'autre.

Le pari radical de Rust est qu'il existe de bons modèles alternatifs. Parfois, vous pouvez résoudre un problème de propriété et de durée de vie de la fermeture en faisant en sorte que chaque fermeture reçoive les références dont elle a besoin comme arguments. Parfois, vous pouvez attribuer un numéro à chaque élément du système et faire circuler les numéros au lieu de références. Ou vous pouvez implémenter l'une des nombreuses variantes sur MVC où les objets n'ont pas tous de références les uns aux autres. Ou modélisez votre boîte à outils d'après un système non-MVC avec un flux de données unidirectionnel, comme l'architecture Flux de Facebook, illustrée à [la figure 14-4](#).



Graphique 14-4. L'architecture Flux, une alternative à MVC

En bref, si vous essayez d'utiliser des fermetures Rust pour créer une « mer d'objets », vous allez avoir du mal. Mais il existe des alternatives. Dans ce cas, il semble que le génie logiciel en tant que discipline gravite déjà autour des alternatives de toute façon, car elles sont plus simples.

Dans le chapitre suivant, nous passons à un sujet où les fermetures brillent vraiment. Nous allons écrire une sorte de code qui tire pleinement parti de la concision, de la vitesse et de l'efficacité des fermetures Rust et qui est amusant à écrire, facile à lire et éminemment pratique. Suivant : les itérateurs de rouille.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

Chapitre 15. Itérateurs

C'était la fin d'une très longue journée.

—Phil

Un *itérateur* est une valeur qui produit une séquence de valeurs, généralement sur laquelle une boucle peut fonctionner. La bibliothèque standard de Rust fournit des itérateurs qui traversent des vecteurs, des chaînes, des tables de hachage et d'autres collections, mais aussi des itérateurs pour produire des lignes de texte à partir d'un flux d'entrée, des connexions arrivant à un serveur réseau, des valeurs reçues d'autres threads sur un canal de communication, etc. Et bien sûr, vous pouvez implémenter des itérateurs à vos propres fins. La boucle de Rust fournit une syntaxe naturelle pour l'utilisation des itérateurs, mais les itérateurs eux-mêmes fournissent également un riche ensemble de méthodes pour cartographier, filtrer, joindre, collecter, etc. `for`

Les itérateurs de Rust sont flexibles, expressifs et efficaces. Considérons la fonction suivante, qui renvoie la somme des premiers entiers positifs (souvent appelé *1^e n ième nombre de triangle*) : n

```
fn triangle(n: i32) -> i32 {
    let mut sum = 0;
    for i in 1..=n {
        sum += i;
    }
    sum
}
```

L'expression est une valeur. A est un itérateur qui produit les entiers de sa valeur de début à sa valeur de fin (les deux inclusives), vous pouvez donc l'utiliser comme opérande de la boucle pour additionner les valeurs de à . 1..=n RangeInclusive<i32> RangeInclusive<i32> for 1 n

Mais les itérateurs ont également une méthode, que vous pouvez utiliser dans la définition équivalente: `fold`

```
fn triangle(n: i32) -> i32 {
    (1..=n).fold(0, |sum, item| sum + item)
}
```

En commençant par le total courant, prend chaque valeur qui produit et applique la fermeture au total courant et à la valeur. La valeur de retour de la fermeture est considérée comme le nouveau total courant. La dernière valeur qu'il renvoie est ce qu'il renvoie lui-même, dans ce cas, le total de la séquence entière. Cela peut sembler étrange si vous êtes habitué et boucles, mais une fois que vous vous y êtes habitué, c'est une alternative lisible et concise.

```
0 fold 1..=n |sum, item| sum + item
```

fold for while fold

C'est un tarif assez standard pour les langages de programmation fonctionnels, qui mettent l'accent sur l'expressivité. Mais les itérateurs de Rust ont été soigneusement conçus pour s'assurer que le compilateur peut également les traduire en un excellent code machine. Dans une version version de la deuxième définition montrée précédemment, Rust connaît la définition de et l'intègre dans . Ensuite, la fermeture est intégrée à cela. Enfin, Rust examine le code combiné et reconnaît qu'il existe un moyen plus simple de additionner les nombres de un à : la somme est toujours égale à . Rust traduit tout le corps de , boucle, fermeture, et tout, en une seule instruction de multiplication et quelques autres bits d'arithmétique.

```
fold triangle |sum, item| sum + item n n * (n+1) / 2 triangle
```

Cet exemple implique une arithmétique simple, mais les itérateurs fonctionnent également bien lorsqu'ils sont utilisés plus lourdement. Ils sont un autre exemple de Rust fournissant des abstractions flexibles qui imposent peu ou pas de frais généraux dans une utilisation typique.

Dans ce chapitre, nous allons vous expliquer :

- Les et traits, qui sont à la base des itérateurs de Rust `Iterator` `IntoIterator`
- Les trois étapes d'un pipeline d'itérateur typique : la création d'un itérateur à partir d'une sorte de source de valeur ; adapter un type d'itérateur à un autre en sélectionnant ou en traitant les valeurs au fur et à mesure; puis consommer les valeurs produites par l'itérateur
- Comment implémenter des itérateurs pour vos propres types

Il y a beaucoup de méthodes, donc c'est bien d'écrire une section une fois que vous avez l'idée générale. Mais les itérateurs sont très courants dans Rust idiomatique, et être familier avec les outils qui les accompagnent est essentiel pour maîtriser le langage.

Les traits Iterator et IntoIterator

Un itérateur est toute valeur qui implémente le trait

```
:std::iter::Iterator
```

```
trait Iterator {  
    type Item;  
    fn next(&mut self) -> Option<Self::Item>;  
    ... // many default methods  
}
```

`Item` est le type de valeur produite par l’itérateur. La méthode renvoie , où est la valeur suivante de l’itérateur ou renvoie pour indiquer la fin de la séquence. Ici, nous avons omis les nombreuses méthodes par défaut; nous les couvrirons individuellement tout au long du reste de ce chapitre. `next Some(v) v None` `Iterator`

S’il existe un moyen naturel d’itérer sur un type, ce type peut implémenter , dont la méthode prend une valeur et renvoie un itérateur sur celle-ci : `std::iter::IntoIterator` `into_iter`

```
trait IntoIterator where Self::IntoIter: Iterator<Item=Self::Item> {  
    type Item;  
    type IntoIter: Iterator;  
    fn into_iter(self) -> Self::IntoIter;  
}
```

`IntoIter` est le type de la valeur itératrice elle-même, et est le type de valeur qu’il produit. Nous appelons n’importe quel type qui implémente un *itérable*, car c’est quelque chose que vous pourriez itérer si vous le demandiez. `Item IntoIterator`

La boucle de Rust rassemble bien toutes ces pièces. Pour itérer sur les éléments d’un vecteur, vous pouvez écrire : `for`

```
println!("There's:");  
let v = vec![ "antimony", "arsenic", "aluminum", "selenium" ];  
  
for element in &v {  
    println!(" {}", element);  
}
```

Sous le capot, chaque boucle n'est qu'un raccourci pour les appels et les méthodes: `for IntoIterator Iterator`

```
let mut iterator = (&v).into_iter();
while let Some(element) = iterator.next() {
    println!("{}", element);
}
```

La boucle utilise pour convertir son opérande en itérateur, puis appelle à plusieurs reprises. Chaque fois que cela revient, la boucle exécute son corps ; et s'il revient, la boucle se

```
termine. for IntoIterator::into_iter &v Iterator::next Some(element) for None
```

Avec cet exemple à l'esprit, voici quelques termes pour les itérateurs :

- Comme nous l'avons dit, un *itérateur* est tout type qui implémente `.Iterator`
- Un *itérable* est tout type qui implémente : vous pouvez obtenir un itérateur dessus en appelant sa méthode. La référence vectorielle est l'itérable dans ce cas. `IntoIterator into_iter &v`
- Un itérateur *produit des valeurs*.
- Les valeurs produites par un itérateur sont *des éléments*. Ici, les éléments sont , , et ainsi de suite. "antimony" "arsenic"
- Le code qui reçoit les articles produits par un itérateur est le *consommateur*. Dans cet exemple, la boucle est le consommateur. `for`

Bien qu'une boucle appelle toujours son opérande, vous pouvez également passer directement des itérateurs aux boucles ; cela se produit lorsque vous effectuez une boucle sur un , par exemple. Tous les itérateurs implémentent automatiquement , avec une méthode qui renvoie simplement

```
l'itérateur. for into_iter for Range IntoIterator into_iter
```

Si vous appelez à nouveau la méthode d'un itérateur après son retour, le trait ne spécifie pas ce qu'il doit faire. La plupart des itérateurs reviendront à nouveau, mais pas tous. (Si cela provoque des problèmes, l'adaptateur couvert dans [« fuse »](#) peut aider.) `next None Iterator None fuse`

Création d'itérateurs

La documentation de la bibliothèque standard Rust explique en détail le type d'itérateurs fournis par chaque type, mais la bibliothèque suit certaines conventions générales pour vous aider à vous orienter et à trouver ce dont vous avez besoin.

Iter et méthodes iter_mut

La plupart des types de collection fournissent et des méthodes qui renvoient les itérateurs naturels sur le type, produisant une référence partagée ou modifiable à chaque élément. Les tranches de tableau aiment et ont et les méthodes aussi. Ces méthodes sont le moyen le plus courant d'obtenir un itérateur, si vous n'allez pas laisser une boucle s'en occuper pour vous: `iter iter_mut &[T] &mut [T]` `iter iter_mut for`

```
let v = vec![4, 20, 12, 8, 6];
let mut iterator = v.iter();
assert_eq!(iterator.next(), Some(&4));
assert_eq!(iterator.next(), Some(&20));
assert_eq!(iterator.next(), Some(&12));
assert_eq!(iterator.next(), Some(&8));
assert_eq!(iterator.next(), Some(&6));
assert_eq!(iterator.next(), None);
```

Le type d'élément de cet itérateur est : chaque appel à produit une référence à l'élément suivant, jusqu'à ce que nous atteignions la fin du vecteur. `&i32 next`

Chaque type est libre d'être mis en œuvre et de la manière la plus logique pour son objectif. La méthode `on` renvoie un itérateur qui produit un composant de chemin d'accès à la fois

```
: iter iter_mut iter std::path::Path
```

```
use std::ffi::OsStr;
use std::path::Path;

let path = Path::new("C:/Users/JimB/Downloads/Fedora.iso");
let mut iterator = path.iter();
assert_eq!(iterator.next(), Some(OsStr::new("C:")));
assert_eq!(iterator.next(), Some(OsStr::new("Users")));
assert_eq!(iterator.next(), Some(OsStr::new("JimB")));
...

```

Le type d'élément de cet itérateur est , une tranche empruntée d'une chaîne du type accepté par les appels du système

```
d'exploitation. &std::ffi::OsStr
```

S'il existe plus d'une façon courante d'itérer sur un type, le type fournit généralement des méthodes spécifiques pour chaque type de traversée, car une méthode simple serait ambiguë. Par exemple, il n'existe aucune méthode sur le type de tranche de chaîne. Au lieu de cela, si est un , renvoie alors un itérateur qui produit chaque octet de , alors qu'il interprète le contenu comme UTF-8 et produit chaque caractère

```
Unicode::iter iter &str s &str s.bytes() s s.chars()
```

Implémentations IntoIterator

Lorsqu'un type implémente , vous pouvez appeler sa méthode vous-même, tout comme une boucle : `IntoIterator into_iter for`

```
// You should usually use HashSet, but its iteration order is
// nondeterministic, so BTreeSet works better in examples.
use std::collections::BTreeSet;
let mut favorites = BTreeSet::new();
favorites.insert("Lucy in the Sky With Diamonds".to_string());
favorites.insert("Liebesträume No. 3".to_string());

let mut it = favorites.into_iter();
assert_eq!(it.next(), Some("Liebesträume No. 3".to_string()));
assert_eq!(it.next(), Some("Lucy in the Sky With Diamonds".to_string()))
assert_eq!(it.next(), None);
```

La plupart des collections fournissent en fait plusieurs implémentations de , pour les références partagées () , les références mutables () et les déplacements () : `IntoIterator &T &mut T T`

- Étant donné une *référence partagée* à la collection, renvoie un itérateur qui produit des références partagées à ses éléments. Par exemple, dans le code précédent, renverrait un itérateur dont le type est

```
.into_iter (&favorites).into_iter() Item &String
```

- Étant donné une *référence modifiable* à la collection, renvoie un itérateur qui produit des références modifiables aux éléments. Par exemple, si est certain , l'appel renvoie un itérateur dont le type est

```
.into_iter vector<String> (&mut
vector).into_iter() Item &mut String
```

- Lorsque la collection est passée par valeur, renvoie un itérateur qui prend possession de la collection et renvoie les éléments par valeur ; la propriété des articles passe de la collection au consommateur, et la col-

lection d'origine est consommée dans le processus. Par exemple, l'appel dans le code précédent renvoie un itérateur qui produit chaque chaîne par valeur ; le consommateur reçoit la propriété de chaque chaîne. Lorsque l'itérateur est abandonné, tous les éléments restant dans le sont également abandonnés et l'enveloppe maintenant vide de l'ensemble est éliminée.

```
into_iter favorites.into_iter() BTreeset
```

Étant donné qu'une boucle s'applique à son opérande, ces trois implémentations sont ce qui crée les idiomes suivants pour itérer sur des références partagées ou modifiables à une collection, ou consommer la collection et s'approprier ses éléments :

```
for IntoIterator::into_iter
```

```
for element in &collection { ... }
for element in &mut collection { ... }
for element in collection { ... }
```

Chacun d'entre eux entraîne simplement un appel à l'une des implementations répertoriées ici.

```
IntoIterator
```

Tous les types ne fournissent pas les trois implementations. Par exemple, , et ne pas implémenter sur des références modifiables, car modifier leurs éléments violerait probablement les invariants du type : la valeur modifiée pourrait avoir une valeur de hachage différente, ou être ordonnée différemment par rapport à ses voisins, de sorte que la modifier la laisserait mal placée. D'autres types prennent en charge la mutation, mais seulement partiellement. Par exemple, et produire des références modifiables aux valeurs de leurs entrées, mais uniquement des références partagées à leurs clés, pour des raisons similaires à celles données précédemment.

```
HashSet BTreeset BinaryHeap IntoIterator HashMap BTreesmap
```

Le principe général est que l'itération doit être efficace et prévisible, donc plutôt que de fournir des implementations coûteuses ou pouvant présenter un comportement surprenant (par exemple, ressasser des entrées modifiées et potentiellement les rencontrer à nouveau plus tard dans l'itération), Rust les omet complètement.

```
HashSet
```

Les tranches implémentent deux des trois variantes ; puisqu'ils ne possèdent pas leurs éléments, il n'y a pas de cas « par valeur ». Au lieu de cela, pour et renvoie un itérateur qui produit des références partagées et mutables aux éléments. Si vous imaginez le type de tranche sous-jacent comme une sorte de collection, cela s'intègre parfaitement dans le modèle global.

```
IntoIterator into_iter &[T] &mut [T] [T]
```

Vous avez peut-être remarqué que les deux premières variantes, pour les références partagées et mutables, sont équivalentes à l'appel ou au référent. Pourquoi Rust fournit-il les deux?

```
IntoIterator iter iter_mut
```

`IntoIterator` c'est ce qui fait fonctionner les boucles, donc c'est évidemment nécessaire. Mais lorsque vous n'utilisez pas de boucle, il est plus clair d'écrire que . L'itération par référence partagée est quelque chose dont vous aurez besoin fréquemment, donc et qui est toujours précieux pour leur ergonomie. `for for favorites.iter()
(&favorites).into_iter() iter iter_mut`

`IntoIterator` peut également être utile dans le code générique : vous pouvez utiliser une liaison comme pour restreindre la variable de type aux types qui peuvent être itérés. Ou, vous pouvez écrire pour exiger davantage l'itération pour produire un type particulier . Par exemple, cette fonction vide les valeurs de tout document itérable dont les éléments sont imprimables au format suivant : `T: IntoIterator<Item=U>`

```
fn dump<T, U>(t: T)  
    where T: IntoIterator<Item=U>,  
          U: Debug
```

```
{  
    for u in t {  
        println!("{}: {:?}", u);  
    }  
}
```

Vous ne pouvez pas écrire cette fonction générique en utilisant et , car ce ne sont pas des méthodes d'un trait quelconque: la plupart des types itérables ont juste des méthodes de ces noms. `iter iter_mut`

from_fn et successeurs

Un moyen simple et général de produire une séquence de valeurs consiste à fournir une fermeture qui les renvoie.

Étant donné qu'une fonction renvoie , renvoie un itérateur qui appelle simplement la fonction pour produire ses éléments. Par exemple: `Option<T> std::iter::from_fn`

```

use rand::random; // In Cargo.toml dependencies: rand = "0.7"
use std::iter::from_fn;

// Generate the lengths of 1000 random line segments whose endpoints
// are uniformly distributed across the interval [0, 1]. (This isn't a
// distribution you're going to find in the `rand_distr` crate, but
// it's easy to make yourself.)
let lengths: Vec<f64> =
    from_fn(|| Some((random::<f64>() - random::<f64>()).abs()))
        .take(1000)
        .collect();

```

Cela nécessite de faire un itérateur produisant des nombres aléatoires. Puisque l'itérateur revient toujours, la séquence ne se termine jamais, mais nous appelons à la limiter aux 1 000 premiers éléments. Construit ensuite le vecteur à partir de l'itération résultante. C'est un moyen efficace de construire des vecteurs initialisés; nous expliquons pourquoi dans [« Building Collections: collect and FromIterator »](#), plus loin dans ce chapitre. `from_fn Some take(1000) collect`

Si chaque élément dépend de celui d'avant, la fonction fonctionne bien. Vous fournissez un élément initial et une fonction qui prend un élément et renvoie un élément du suivant. S'il renvoie , l'itération se termine. Par exemple, voici une autre façon d'écrire la fonction à partir de notre traceur d'ensemble Mandelbrot dans [le chapitre 2](#)

```
:std::iter::successors Option None escape_time
```

```

use num::Complex;
use std::iter::successors;

fn escape_time(c: Complex<f64>, limit: usize) -> Option<usize> {
    let zero = Complex { re: 0.0, im: 0.0 };
    successors(Some(zero), |&z| { Some(z * z + c) })
        .take(limit)
        .enumerate()
        .find(|(_i, z)| z.norm_sqr() > 4.0)
        .map(|(i, _z)| i)
}

```

En commençant par zéro, l'appel produit une séquence de points sur le plan complexe en quadrature répétée du dernier point et en ajoutant le paramètre . Lors du tracé de l'ensemble de Mandelbrot, nous voulons voir si cette séquence orbite près de l'origine pour toujours ou s'envole vers l'infini. L'appel établit une limite sur la durée pendant laquelle nous

poursuivrons la séquence et numérote chaque point, transformant chaque point en un tuple. Nous avons l'habitude de chercher le premier point qui s'éloigne suffisamment de l'origine pour s'échapper. La méthode renvoie un : s'il en existe un ou non. L'appel à se transforme en , mais retourne inchangé : c'est exactement la valeur de retour que nous voulons.

```
successors c take(limit) enumerate z (i,
z) find find Option(Some((i, z)) None Option::map Some((i,
z)) Some(i) None
```

Les deux et acceptent les fermetures, afin que vos fermetures puissent capturer et modifier les variables des étendues environnantes. Par exemple, cette fonction utilise une fermeture pour capturer une variable et l'utiliser comme état d'exécution

```
:from_fn successors FnMut fibonacci move
```

```
fn fibonacci() -> impl Iterator<Item=usize> {
    let mut state = (0, 1);
    std::iter::from_fn(move || {
        state = (state.1, state.0 + state.1);
        Some(state.0)
    })
}

assert_eq!(fibonacci().take(8).collect::<Vec<_>>(),
           vec![1, 1, 2, 3, 5, 8, 13, 21]);
```

Une note de prudence: les méthodes et sont suffisamment flexibles pour que vous puissiez transformer à peu près n'importe quelle utilisation d'itérateurs en un seul appel à l'un ou l'autre, en passant des fermetures complexes pour obtenir le comportement dont vous avez besoin. Mais cela néglige la possibilité offerte par les itérateurs de clarifier la façon dont les données circulent dans le calcul et d'utiliser des noms standard pour les modèles courants. Assurez-vous de vous être familiarisé avec les autres méthodes d'itération de ce chapitre avant de vous appuyer sur ces deux méthodes; il y a souvent de meilleures façons de faire le travail.

```
from_fn successors
```

méthodes de drainage

De nombreux types de collection fournissent une méthode qui prend une référence modifiable à la collection et renvoie un itérateur qui transmet la propriété de chaque élément au consommateur. Cependant, contrairement à la méthode , qui prend la collection par valeur et la consomme,

emprunte simplement une référence modifiable à la collection, et lorsque l'itérateur est abandonné, il supprime tous les éléments restants de la collection et le laisse vide. `drain` `into_iter()` `drain`

Sur les types qui peuvent être indexés par une plage, comme `s`, vecteurs et `s`, la méthode prend une plage d'éléments à supprimer, plutôt que de drainer toute la séquence : `String` `VecDeque` `drain`

```
let mut outer = "Earth".to_string();
let inner = String::from_iter(outer.drain(1..4));

assert_eq!(outer, "Eh");
assert_eq!(inner, "art");
```

Si vous devez vider toute la séquence, utilisez la plage complète, , comme argument . . .

Autres sources d'itérateurs

Les sections précédentes concernent principalement les types de collection tels que les vecteurs et , mais il existe de nombreux autres types dans la bibliothèque standard qui prennent en charge l'itération. [Le tableau 15-1](#) résume les plus intéressants, mais il y en a beaucoup d'autres. Nous couvrons certaines de ces méthodes plus en détail dans les chapitres consacrés aux types spécifiques (à savoir, les chapitres [16](#), [17](#) et [18](#)). `HashMap`

Tableau 15-1. Autres itérateurs dans la bibliothèque standard

Type ou trait	Expression	Notes
<code>std::op::Rang</code>	<code>1 .. 10</code>	Les points de terminaison doivent être de type entier pour pouvoir être itérables. La plage inclut la valeur de début et exclut la valeur de fin.
	<code>(1 .. 10).s tep_by(2)</code>	Produit. 1 3 5 7 9
<code>std::op::From</code>	<code>1 ..</code>	Itération illimitée. Start doit être un entier. Peut paniquer ou déborder si la valeur atteint la limite du type.
<code>std::op::Inclus</code>	<code>1 .. =10</code>	Comme , mais inclut la valeur finale. Range
	<code>s ::Rang eInclus</code>	
	<code>ive</code>	
<code>Option<T></code>	<code>Some(10).iter()</code>	Se comporte comme un vecteur dont la longueur est soit 0 () ou 1 (). None Some (v)
<code>Result<T, E></code>	<code>Ok("bla h").iter()</code>	Semblable à , produisant des valeurs. Option Ok
<code>Vec<T>, &[T]</code>	<code>v.window s(16)</code>	Produit chaque tranche contiguë de la longueur donnée, de gauche à droite. Les fenêtres se chevauchent.
	<code>v.chunks_(16)</code>	Produit des tranches contiguës non superposées de la longueur donnée, de gauche à droite.
	<code>v.chunks_mut(1024)</code>	Comme , mais les tranches sont mutables. chunks

Type ou trait	Expression	Notes
	v.split(byte byt e & 1 != 0)	Produit des tranches séparées par des éléments qui correspondent au prédictat donné.
	v.split_ mut(...)	Comme ci-dessus, mais produit des tranches mutables.
	v.rsplit (...)	Comme , mais produit des tranches de droite à gauche. <code>split</code>
	v.splitn (n, ...)	Comme , mais produit au plus des tranches. <code>split n</code>
String, &str	s.bytes ()	Produit les octets du formulaire UTF-8.
	s.chars ()	Produit le s que représente l'UTF-8. <code>char</code>
	s.split_w hitespace ()	Divise la chaîne par espace et produit des tranches de caractères non spatiaux.
	s.lines ()	Produit des tranches des lignes de la chaîne.
	s.split ('/')	Divise la chaîne sur un motif donné, produisant les tranches entre les correspondances. Les motifs peuvent être beaucoup de choses: caractères, chaînes, fermetures.
	s.matches (char::is _numeric)	Produit des tranches correspondant au motif donné.

Type ou trait	Expression	Notes
std::co llectio ns::Has hMap , std::co llectio ns::BTr eeMap	map.keys () , map.valu es() map.value s_mut()	Produit des références partagées aux clés ou aux valeurs de la carte. Produit des références modifiables aux valeurs des entrées.
std::co llectio ns::Has hSet , std::co llectio ns::BTr eeSet	set1.unio n(set2) set1.inte rsection (set2)	Produit des références partagées à des éléments d'union de et .set1 set2 Produit des références partagées aux éléments d'intersection de et .set1 set2
std::sy nc::mps c::Rece iver	recv.ite r()	Produit des valeurs envoyées à partir d'un autre thread sur le fichier .Sender
std::i o::Read	stream.by tes()	Produit des octets à partir d'un flux d'E/S.
	stream.ch ars()	Analyse le flux en UTF-8 et produit s. cha r
std::i o::BufR ead	bufstrea m.lines()	Analyse le flux en UTF-8, produit des lignes en s. String
	bufstrea m.split (0)	Divise le flux sur un octet donné, produit des tampons inter-octets. Vec<u8>

Type ou trait	Expression	Notes
std::f s::Read Dir	std::fs:: read_dir (path)	Produit des entrées d'annuaire.
std::ne t::TcpL istener	listener. incoming ()	Produit des connexions réseau entrantes.
Fonctions libres	std::ite r::empty ()	Renvoie immédiatement. None
	std::ite r::once (5)	Produit la valeur donnée, puis se termine.
	std::ite r::repeat ("#9")	Produit la valeur donnée pour toujours.

Adaptateurs d'itérateur

Une fois que vous avez un itérateur en main, le trait fournit une large sélection de *méthodes* d'adaptateur, ou simplement des *adaptateurs*, qui consomment un itérateur et en construisent un nouveau avec des comportements utiles. Pour voir comment fonctionnent les adaptateurs, nous allons commencer par deux des adaptateurs les plus populaires, et . Ensuite, nous couvrirons le reste de la boîte à outils de l'adaptateur, couvrant presque toutes les façons que vous pouvez imaginer pour créer des séquences de valeurs à partir d'autres séquences: troncature, saut, combinaison, inversion, concaténation, répétition, etc. `Iterator map filter`

carte et filtre

L'adaptateur du trait vous permet de transformer un itérateur en appliquant une fermeture à ses éléments. L'adaptateur vous permet de filtrer

les éléments d'un itérateur, à l'aide d'une fermeture pour décider lesquels conserver et lesquels déposer. `Iterator map filter`

Par exemple, supposons que vous itérez sur des lignes de texte et que vous souhaitez omettre les espaces blancs de début et de fin de chaque ligne. La méthode de la bibliothèque standard supprime les espaces blancs de début et de fin d'un seul, renvoyant un nouveau rognage qui emprunte à l'original. Vous pouvez utiliser l'adaptateur pour appliquer à chaque ligne à partir de l'itérateur

```
:str::trim &str &str map str::trim
```

```
let text = "  ponies  \n  giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .collect();
assert_eq!(v, ["ponies", "giraffes", "iguanas", "squid"]);
```

L'appel renvoie un itérateur qui produit les lignes de la chaîne. L'appel de cet itérateur renvoie un deuxième itérateur qui s'applique à chaque ligne et produit les résultats en tant qu'éléments. Enfin, rassemble ces éléments dans un vecteur. `text.lines() map str::trim collect`

L'itérateur qui retourne est, bien sûr, lui-même un candidat à une adaptation ultérieure. Si vous souhaitez exclure les iguanes du résultat, vous pouvez écrire ce qui suit: `map`

```
let text = "  ponies  \n  giraffes\niguanas  \nsquid".to_string();
let v: Vec<&str> = text.lines()
    .map(str::trim)
    .filter(|s| *s != "iguanas")
    .collect();
assert_eq!(v, ["ponies", "giraffes", "squid"]);
```

Ici, renvoie un troisième itérateur qui produit uniquement les éléments de l'itérateur pour lesquels la fermeture renvoie . Une chaîne d'adaptateurs d'itérateur est comme un pipeline dans le shell Unix : chaque adaptateur a un seul but, et il est clair comment la séquence est transformée au fur et à mesure que l'on lit de gauche à droite. `filter map |s| *s != "iguanas" true`

Les signatures de ces adaptateurs sont les suivantes :

```
fn map<B, F>(self, f: F) -> impl Iterator<Item=B>
where Self: Sized, F: FnMut(Self::Item) -> B;
```

```
fn filter<P>(self, predicate: P) -> impl Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

Dans la bibliothèque standard, et renvoie en fait des types opaques spécifiques nommés et . Cependant, le simple fait de voir leurs noms n'est pas très instructif, donc dans ce livre, nous allons simplement écrire à la place, car cela nous dit ce que nous voulons vraiment savoir: la méthode renvoie un qui produit des éléments du type

```
donné. map filter struct std::iter::Map std::iter::Filter ->
impl Iterator<Item=...> Iterator
```

Étant donné que la plupart des adaptateurs prennent par valeur, ils doivent l'être (ce que sont tous les itérateurs courants). `self` `Self` `Sized`

Un itérateur transmet chaque article à sa fermeture en valeur et, à son tour, transmet la propriété du résultat de la fermeture à son consommateur. Un itérateur transmet chaque article à sa fermeture par référence partagée, en conservant la propriété au cas où l'article serait sélectionné pour être transmis à son consommateur. C'est pourquoi l'exemple doit être déréférencé pour le comparer avec : le type d'élément de l'itérateur est , donc le type de l'argument de fermeture est

```
.map filter s "iguanas" filter &str s &&str
```

Il y a deux points importants à noter à propos des adaptateurs d'itérateur.

Tout d'abord, le simple fait d'appeler un adaptateur sur un itérateur ne consomme aucun élément ; il renvoie simplement un nouvel itérateur, prêt à produire ses propres articles en puisant dans le premier itérateur au besoin. Dans une chaîne d'adaptateurs, la seule façon de faire fonctionner réellement est de faire appel à l'itérateur final. `next`

Ainsi, dans notre exemple précédent, l'appel de méthode lui-même n'analyse en fait aucune ligne de la chaîne; il renvoie simplement un itérateur qui *analyserait* les lignes si on le lui demandait. De même, et il suffit de renvoyer de nouveaux itérateurs qui *mapperaient ou filtreraient* si on le leur demandait. Aucun travail n'a lieu jusqu'à ce qu'il commence à faire appel à l'itérateur. `text.lines() map filter collect next filter`

Ce point est particulièrement important si vous utilisez des adaptateurs qui ont des effets secondaires. Par exemple, ce code n'imprime rien du tout :

```
[ "earth", "water", "air", "fire"]
    .iter().map(|elt| println!("{}", elt));
```

L'appel renvoie un itérateur sur les éléments du tableau et l'appel renvoie un deuxième itérateur qui applique la fermeture à chaque valeur produite par le premier. Mais il n'y a rien ici qui exige jamais une valeur de toute la chaîne, donc aucune méthode ne fonctionne jamais. En fait, Rust vous avertira à ce sujet: `iter map next`

```
warning: unused `std::iter::Map` that must be used
|
7 | /      ["earth", "water", "air", "fire"]
8 | |       .iter().map(|elt| println!("{}", elt));
| |
|
= note: iterators are lazy and do nothing unless consumed
```

Le terme « paresseux » dans le message d'erreur n'est pas un terme dé-sobligeant ; c'est juste du jargon pour tout mécanisme qui retarde un calcul jusqu'à ce que sa valeur soit nécessaire. Il est de convention de Rust que les itérateurs doivent faire le minimum de travail nécessaire pour satisfaire chaque appel à ; dans l'exemple, il n'y a pas du tout de tels appels, donc aucun travail n'a lieu. `next`

Le deuxième point important est que les adaptateurs d'itérateur sont une abstraction sans surcharge. Puisque , , et leurs compagnons sont génériques, les appliquer à un itérateur spécialise leur code pour le type d'itérateur spécifique impliqué. Cela signifie que Rust dispose de suffisamment d'informations pour intégrer la méthode de chaque itérateur dans son consommateur, puis traduire l'ensemble de l'arrangement en code machine en tant qu'unité. Ainsi, la chaîne // d'itérateurs que nous avons montrée précédemment est aussi efficace que le code que vous écririez probablement à la main: `map filter next lines map filter`

```
for line in text.lines() {
    let line = line.trim();
    if line != "iguanas" {
        v.push(line);
    }
}
```

Le reste de cette section couvre les différents adaptateurs disponibles sur le trait `Iterator`

filter_map et flat_map

L'adaptateur est correct dans les situations où chaque élément entrant produit un élément sortant. Mais que se passe-t-il si vous souhaitez supprimer certains éléments de l'itération au lieu de les traiter ou remplacer des éléments uniques par zéro ou plusieurs éléments ? Les adaptateurs et vous offrent cette flexibilité. `map filter_map flat_map`

L'adaptateur est similaire à sauf qu'il permet à sa fermeture de transformer l'élément en un nouvel élément (comme le fait) ou de supprimer l'élément de l'itération. Ainsi, c'est un peu comme une combinaison de `et`. Sa signature est la suivante : `filter_map map map filter_map`

```
fn filter_map<B, F>(self, f: F) -> impl Iterator<Item=B>
    where Self: Sized, F: FnMut(Self::Item) -> Option<B>;
```

C'est la même chose que la signature de `'`, sauf qu'ici la fermeture renvoie , pas simplement . Lorsque la fermeture revient, l'élément est supprimé de l'itération ; lorsqu'il retourne , alors est l'élément suivant produit par l'itérateur. `map Option B None Some(b) b filter_map`

Par exemple, supposons que vous souhaitez analyser une chaîne à la recherche de mots séparés par des espaces blancs qui peuvent être analysés en tant que nombres et traiter les nombres en supprimant les autres mots. Vous pouvez écrire :

```
use std::str::FromStr;

let text = "1\nfrond .25 289\n3.1415 estuary\n";
for number in text
    .split_whitespace()
    .filter_map(|w| f64::from_str(w).ok())
{
    println!("{:4.2}", number.sqrt());
}
```

Cela imprime les éléments suivants :

```
1.00
0.50
17.00
1.77
```

La fermeture donnée à tente d'analyser chaque tranche séparée par des espaces blancs à l'aide de . Cela renvoie un , qui se transforme en un : une erreur d'analyse devient , alors qu'un résultat d'analyse réussie devient .

L'itérateur supprime toutes les valeurs et produit la valeur pour chaque

```
.filter_map f64::from_str Result<f64,  
ParseFloatError> .ok() Option<f64> None Some(v) filter_map N  
one v Some(v)
```

Mais quel est l'intérêt de fusionner et de former une seule opération comme celle-ci, au lieu de simplement utiliser ces adaptateurs directement? L'adaptateur affiche sa valeur dans des situations comme celle qui vient d'être montrée, lorsque la meilleure façon de décider d'inclure ou non l'élément dans l'itération est d'essayer de le traiter. Vous pouvez faire la même chose avec seulement et , mais c'est un peu disgracieux: map filter filter_map filter map

```
text.split_whitespace()  
.map(|w| f64::from_str(w))  
.filter(|r| r.is_ok())  
.map(|r| r.unwrap())
```

Vous pouvez penser que l'adaptateur continue dans la même veine que et , sauf que maintenant la fermeture peut renvoyer non seulement un élément (comme avec) ou zéro ou un élément (comme avec), mais une séquence de n'importe quel nombre d'éléments. L'itérateur produit la concaténation des séquences renvoyées par la

```
fermeture.flat_map map filter_map map filter_map flat_map
```

La signature de est indiquée ici : flat_map

```
fn flat_map<U, F>(self, f: F) -> impl Iterator<Item=U::Item>  
where F: FnMut(Self::Item) -> U, U: IntoIterator;
```

La fermeture passée à doit retourner un itérable, mais toute sorte d'itérable fera l'affaire. flat_map ¹

Par exemple, supposons que nous ayons un tableau cartographiant les pays à leurs grandes villes. Compte tenu d'une liste de pays, comment pouvons-nous itérer sur leurs grandes villes?

```
use std::collections::HashMap;  
  
let mut major_cities = HashMap::new();
```

```

major_cities.insert("Japan", vec![ "Tokyo", "Kyoto"]);
major_cities.insert("The United States", vec![ "Portland", "Nashville"]);
major_cities.insert("Brazil", vec![ "São Paulo", "Brasília"]);
major_cities.insert("Kenya", vec![ "Nairobi", "Mombasa"]);
major_cities.insert("The Netherlands", vec![ "Amsterdam", "Utrecht"]);

let countries = [ "Japan", "Brazil", "Kenya"];

for &city in countries.iter().flat_map(|country| &major_cities[country])
    println!("{}", city);
}

```

Cela imprime les éléments suivants :

```

Tokyo
Kyoto
São Paulo
Brasília
Nairobi
Mombasa

```

Une façon de voir cela serait de dire que, pour chaque pays, nous récupérons le vecteur de ses villes, concaténons tous les vecteurs ensemble en une seule séquence et l'imprimons.

Mais rappelez-vous que les itérateurs sont paresseux : ce ne sont que les appels de la boucle à la méthode de l'itérateur qui font que le travail est effectué. La séquence concaténée complète n'est jamais construite en mémoire. Au lieu de cela, ce que nous avons ici est une petite machine d'état qui puise dans l'itérateur de ville, un élément à la fois, jusqu'à ce qu'il soit épousé, et seulement ensuite produit un nouvel itérateur de ville pour le pays suivant. L'effet est celui d'une boucle imbriquée, mais emballée pour être utilisée comme itérateur. `for flat_map next`

aplatisir

L'adaptateur concatène les éléments d'un itérateur, en supposant que chaque élément est lui-même itérable : `flatten`

```

use std::collections::BTreeMap;

// A table mapping cities to their parks: each value is a vector.
let mut parks = BTreeMap::new();
parks.insert("Portland", vec![ "Mt. Tabor Park", "Forest Park"]);
parks.insert("Kyoto",      vec![ "Tadasu-no-Mori Forest", "Maruyama Koen"])

```

```

    parks.insert("Nashville", vec![ "Percy Warner Park", "Dragon Park"]);

    // Build a vector of all parks. `values` gives us an iterator producing
    // vectors, and then `flatten` produces each vector's elements in turn.
    let all_parks: Vec<_> = parks.values().flatten().cloned().collect();

    assert_eq!(all_parks,
               vec![ "Tadasu-no-Mori Forest", "Maruyama Koen", "Percy Warner
                     "Dragon Park", "Mt. Tabor Park", "Forest Park"]);

```

Le nom « aplatisir » vient de l'image de l'aplatissement d'une structure à deux niveaux en une structure à un niveau: le et ses s de noms sont aplatis dans un itérateur produisant tous les noms. BTTreeMap Vec

La signature de est la suivante : flatten

```

fn flatten(self) -> impl Iterator<Item=Self::Item::Item>
    where Self::Item: IntoIterator;

```

En d'autres termes, les éléments de l'itérateur sous-jacent doivent eux-mêmes être implémentés afin qu'il s'agisse effectivement d'une séquence de séquences. La méthode renvoie ensuite un itérateur sur la concaténation de ces séquences. Bien sûr, cela se fait paresseusement, en dessinant un nouvel élément uniquement lorsque nous avons fini d'itérer sur le dernier. IntoIterator flatten self

La méthode est utilisée de plusieurs manières surprenantes. Si vous avez un et que vous voulez itérer uniquement sur les valeurs, fonctionne à merveille: flatten Vec<Option<...>> Some flatten

```

assert_eq!(vec![None, Some("day"), None, Some("one")])
    .into_iter()
    .flatten()
    .collect::<Vec<_>>(),
    vec![ "day", "one"]);

```

Cela fonctionne parce qu'il implémente lui-même , représentant une séquence de zéro ou un élément. Les éléments n'apportent rien à l'itération, alors que chaque élément apporte une seule valeur. De même, vous pouvez utiliser pour itérer sur les valeurs: se comporte de la même manière qu'un vecteur

vide. Option IntoIterator None Some flatten Option<Vec<...>> N
one

Result implémente également , avec la représentation d'une séquence vide, de sorte que l'application à un itérateur de valeurs extrait efficacement tous les s et les jette, ce qui entraîne un flux des valeurs de succès non emballées. Nous ne recommandons pas d'ignorer les erreurs dans votre code, mais c'est une astuce intéressante que les gens utilisent lorsqu'ils pensent savoir ce qui se passe. `IntoIterator Err flatten Result Err`

Vous pouvez vous retrouver à chercher quand ce dont vous avez réellement besoin est . Par exemple, la méthode de la bibliothèque standard, qui convertit une chaîne en majuscules, fonctionne comme ceci

`:flatten flat_map str::to_uppercase`

```
fn to_uppercase(&self) -> String {
    self.chars()
        .map(char::to_uppercase)
        .flatten() // there's a better way
        .collect()
}
```

La raison pour laquelle le est nécessaire est qu'il ne renvoie pas un seul caractère, mais un itérateur produisant un ou plusieurs caractères. Le mappage de chaque caractère à son équivalent majuscule donne un itérateur d'itérateurs de caractères, et le prend soin de les épisser tous ensemble en quelque chose que nous pouvons enfin dans un

`.flatten ch.to_uppercase() flatten collect String`

Mais cette combinaison de et est si courante que fournit l'adaptateur pour ce cas. (En fait, a été ajouté à la bibliothèque standard avant.) Ainsi, le code précédent pourrait plutôt être écrit

`:map flatten Iterator flat_map flat_map flatten`

```
fn to_uppercase(&self) -> String {
    self.chars()
        .flat_map(char::to_uppercase)
        .collect()
}
```

prendre et take_while

Les traits et les adaptateurs vous permettent de terminer une itération après un certain nombre d'éléments ou lorsqu'une fermeture décide de

couper les choses. Leurs signatures sont les suivantes

: Iterator take take_while

```
fn take(self, n: usize) -> impl Iterator<Item=Self::Item>
    where Self: Sized;

fn take_while<P>(self, predicate: P) -> impl Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

Les deux prennent possession d'un itérateur et renvoient un nouvel itérateur qui transmet les éléments du premier, mettant éventuellement fin à la séquence plus tôt. L'itérateur revient après avoir produit la plupart des articles. L'itérateur s'applique à chaque article et retourne à la place du premier article pour lequel il retourne et à chaque appel ultérieur à

.take None n take_while predicate None predicate false next

Par exemple, étant donné un message électronique avec une ligne vide séparant les en-têtes du corps du message, vous pouvez utiliser pour itérer uniquement sur les en-têtes : take_while

```
let message = "To: jimb\r\n"
    From: superego <editor@oreilly.com>\r\n\
    \r\n\
    Did you get any writing done today?\r\n\
    When will you stop wasting time plotting fractals?\r\n";
for header in message.lines().take_while(|l| !l.is_empty()) {
    println!("{}" , header);
}
```

Rappelez-vous de « [String Literals](#) » que lorsqu'une ligne d'une chaîne se termine par une barre oblique inverse, Rust n'inclut pas l'indentation de la ligne suivante dans la chaîne, de sorte qu'aucune des lignes de la chaîne n'a d'espace blanc de début. Cela signifie que la troisième ligne de est vide. L'adaptateur met fin à l'itération dès qu'il voit cette ligne vide, de sorte que ce code imprime uniquement les deux lignes

: message take_while

```
To: jimb
From: superego <editor@oreilly.com>
```

sauter et skip_while

Les traits et les méthodes sont le complément de et : ils laissent tomber un certain nombre d'éléments à partir du début d'une itération, ou laissent tomber des éléments jusqu'à ce qu'une fermeture en trouve un acceptable, puis passent les éléments restants inchangés. Leurs signatures sont les suivantes : `Iterator skip skip_while take take_while`

```
fn skip(self, n: usize) -> impl Iterator<Item=Self::Item>
    where Self: Sized;

fn skip_while<P>(self, predicate: P) -> impl Iterator<Item=Self::Item>
    where Self: Sized, P: FnMut(&Self::Item) -> bool;
```

Une utilisation courante de l'adaptateur consiste à ignorer le nom de la commande lors de l'itération sur les arguments de ligne de commande d'un programme. Dans [le chapitre 2](#), notre plus grande calculatrice de dénominateur commun a utilisé le code suivant pour effectuer une boucle sur ses arguments de ligne de commande : `skip`

```
for arg in std::env::args().skip(1) {
    ...
}
```

La fonction renvoie un itérateur qui produit les arguments du programme sous la forme `s`, le premier élément étant le nom du programme lui-même. Ce n'est pas une chaîne que nous voulons traiter dans cette boucle. L'appel de cet itérateur renvoie un nouvel itérateur qui supprime le nom du programme la première fois qu'il est appelé, puis produit tous les arguments suivants. `std::env::args String skip(1)`

L'adaptateur utilise une fermeture pour décider du nombre d'éléments à supprimer depuis le début de la séquence. Vous pouvez itérer sur les lignes du corps du message de la section précédente comme ceci: `skip_while`

```
for body in message.lines()
    .skip_while(|l| !l.is_empty())
    .skip(1) {
        println!("{}" , body);
    }
```

Cela permet de sauter les lignes non vides, mais cet itérateur produit la ligne vide elle-même (après tout, la fermeture renvoyée pour cette ligne. Nous utilisons donc également la méthode pour laisser tomber cela, nous

donnant un itérateur dont le premier élément sera la première ligne du corps du message. Pris avec la déclaration de la section précédente, ce code imprime: `skip_while false skip message`

```
Did you get any writing done today?  
When will you stop wasting time plotting fractals?
```

peekable

Un itérateur jetable vous permet de jeter un coup d'œil à l'article suivant qui sera produit sans le consommer réellement. Vous pouvez transformer n'importe quel itérateur en un itérateur peekable en appelant la méthode du trait : `Iterator peekable`

```
fn peekable(self) -> std::iter::Peekable<Self>  
    where Self: Sized;
```

Ici, `self` est un type qui implémente `Iterator`, et `Self` est le type de l'itérateur sous-jacent. `Peekable<Self>` struct `Iterator<Item=Self::Item>` `Self`

Un itérateur a une méthode supplémentaire qui renvoie un `Option<&Item>` : si l'itérateur sous-jacent est fait et sinon, où est une référence partagée à l'élément suivant. (Notez que si le type d'élément de l'itérateur est déjà une référence à quelque chose, cela finit par être une référence à une référence.) `Peekable peek Option<&Item> None Some(r) r`

L'appel tente de dessiner l'élément suivant à partir de l'itérateur sous-jacent et, s'il y en a un, le met en cache jusqu'à l'appel suivant à `next`. Toutes les autres méthodes sur connaissent ce cache: par exemple, sur un itérateur `peekable` sait vérifier le cache après avoir épuisé l'itérateur sous-jacent. `peek next Iterator Peekable iter.last() iter`

Les itérateurs `peekables` sont essentiels lorsque vous ne pouvez pas décider du nombre d'articles à consommer à partir d'un itérateur avant d'être allé trop loin. Par exemple, si vous analysez des nombres à partir d'un flux de caractères, vous ne pouvez pas décider où le nombre se termine tant que vous n'avez pas vu le premier caractère non numérique qui le suit :

```
use std::iter::Peekable;  
  
fn parse_number<I>(tokens: &mut Peekable<I>) -> u32  
    where I: Iterator<Item=char>
```

```

{
    let mut n = 0;
    loop {
        match tokens.peek() {
            Some(r) if r.is_digit(10) => {
                n = n * 10 + r.to_digit(10).unwrap();
            }
            _ => return n
        }
        tokens.next();
    }
}

let mut chars = "226153980,1766319049".chars().peekable();
assert_eq!(parse_number(&mut chars), 226153980);
// Look, `parse_number` didn't consume the comma! So we will.
assert_eq!(chars.next(), Some(','));
assert_eq!(parse_number(&mut chars), 1766319049);
assert_eq!(chars.next(), None);

```

La fonction permet de vérifier le caractère suivant et ne le consomme que s'il s'agit d'un chiffre. S'il ne s'agit pas d'un chiffre ou si l'itérateur est épuisé (c'est-à-dire si revient), nous retournons le nombre que nous avons analysé et laissons le caractère suivant dans l'itérateur, prêt à être consommé.

`parse_number peek peek None`

fusible

Une fois que `an` est revenu, le trait ne spécifie pas comment il doit se comporter si vous appelez à nouveau sa méthode. La plupart des itérateurs reviennent à nouveau, mais pas tous. Si votre code compte sur ce comportement, vous risquez d'être surpris.

`Iterator None next None`

L'adaptateur prend n'importe quel itérateur et en produit un qui continuera certainement à revenir une fois qu'il l'aura fait la première fois:

`fuse None`

```

struct Flaky(bool);

impl Iterator for Flaky {
    type Item = &'static str;
    fn next(&mut self) -> Option<Self::Item> {
        if self.0 {
            self.0 = false;
            Some("totally the last item")
        } else {
            None
        }
    }
}

```

```

    } else {
        self.0 = true; // D'oh!
        None
    }
}

let mut flaky = Flaky(true);
assert_eq!(flaky.next(), Some("totally the last item"));
assert_eq!(flaky.next(), None);
assert_eq!(flaky.next(), Some("totally the last item"));

let mut not_flaky = Flaky(true).fuse();
assert_eq!(not_flaky.next(), Some("totally the last item"));
assert_eq!(not_flaky.next(), None);
assert_eq!(not_flaky.next(), None);

```

L'adaptateur est probablement le plus utile dans le code générique qui doit fonctionner avec des itérateurs d'origine incertaine. Plutôt que d'espérer que chaque itérateur auquel vous aurez à faire face se comportera bien, vous pouvez l'utiliser pour vous en assurer. `fuse`

Itérateurs réversibles et régime

Certains itérateurs sont capables de dessiner des éléments des deux extrémités de la séquence. Vous pouvez inverser ces itérateurs à l'aide de l'adaptateur. Par exemple, un itérateur sur un vecteur pourrait tout aussi bien dessiner des éléments à partir de la fin du vecteur que dès le début. De tels itérateurs peuvent implémenter le trait, qui étend

```
:rev std::iter::DoubleEndedIterator Iterator
```

```

trait DoubleEndedIterator: Iterator {
    fn next_back(&mut self) -> Option<Self::Item>;
}

```

Vous pouvez penser à un itérateur à double extrémité comme ayant deux doigts marquant l'avant et l'arrière actuels de la séquence. Dessiner des éléments de chaque extrémité fait avancer ce doigt vers l'autre; lorsque les deux se rencontrent, l'itération est terminée :

```

let bee_parts = ["head", "thorax", "abdomen"];

let mut iter = bee_parts.iter();
assert_eq!(iter.next(), Some(&"head"));

```

```

assert_eq!(iter.next_back(), Some(&"abdomen"));
assert_eq!(iter.next(),      Some(&"thorax"));

assert_eq!(iter.next_back(), None);
assert_eq!(iter.next(),      None);

```

La structure d'un itérateur sur une tranche rend ce comportement facile à mettre en œuvre : il s'agit littéralement d'une paire de pointeurs vers le début et la fin de la gamme d'éléments que nous n'avons pas encore produits ; et il suffit de dessiner un élément de l'un ou de l'autre. Les itérateurs pour les collections ordonnées aiment et sont également à double extrémité: leur méthode dessine d'abord les plus grands éléments ou entrées. En général, la bibliothèque standard fournit une itération à double extrémité chaque fois que cela est

pratique. `next` `next_back` `BTreeSet` `BTreeMap` `next_back`

Mais tous les itérateurs ne peuvent pas le faire aussi facilement : un itérateur produisant des valeurs à partir d'autres threads arrivant à un canal n'a aucun moyen d'anticiper ce que pourrait être la dernière valeur reçue. En général, vous devrez consulter la documentation de la bibliothèque standard pour voir quels itérateurs implémentent et lesquels ne le font pas. `Receiver DoubleEndedIterator`

Si un itérateur est à double extrémité, vous pouvez l'inverser avec l'adaptateur : `rev`

```

fn rev(self) -> impl Iterator<Item=Self>
    where Self: Sized + DoubleEndedIterator;

```

L'itérateur retourné est également à double extrémité: ses méthodes et celles-ci sont simplement échangées: `next` `next_back`

```

let meals = [ "breakfast", "lunch", "dinner" ];

let mut iter = meals.iter().rev();
assert_eq!(iter.next(), Some(&"dinner"));
assert_eq!(iter.next(), Some(&"lunch"));
assert_eq!(iter.next(), Some(&"breakfast"));
assert_eq!(iter.next(), None);

```

La plupart des adaptateurs d'itérateur, s'ils sont appliqués à un itérateur réversible, renvoient un autre itérateur réversible. Par exemple, et préserver la réversibilité. `map` `filter`

inspector

L'adaptateur est pratique pour déboguer les pipelines des adaptateurs d'itérateur, mais il n'est pas beaucoup utilisé dans le code de production. Il applique simplement une fermeture à une référence partagée à chaque élément, puis transmet l'élément. La fermeture ne peut pas affecter les éléments, mais elle peut faire des choses comme les imprimer ou faire des affirmations à leur sujet. `inspect`

Cet exemple montre un cas dans lequel la conversion d'une chaîne en majuscules modifie sa longueur :

```
let upper_case: String = "große".chars()
    .inspect(|c| println!("before: {:?}", c))
    .flat_map(|c| c.to_uppercase())
    .inspect(|c| println!(" after:      {:?}", c))
    .collect();
assert_eq!(upper_case, "GROSSE");
```

L'équivalent majuscule de la lettre allemande minuscule « ß » est « SS », c'est pourquoi renvoie un itérateur sur les caractères, pas un seul caractère de remplacement. Le code précédent permet de concaténer toutes les séquences qui retournent en une seule , en imprimant les éléments suivants au fur et à mesure

```
:char::to_uppercase flat_map to_uppercase String
```

```
before: 'g'
  after:      'G'
before: 'r'
  after:      'R'
before: 'o'
  after:      'O'
before: 'ß'
  after:      'S'
  after:      'S'
before: 'e'
  after:      'E'
```

chaîne

L'adaptateur ajoute un itérateur à un autre. Plus précisément, renvoie un itérateur qui tire des éléments jusqu'à ce qu'il soit épuisé, puis tire des éléments de `.chain i1.chain(i2) i1 i2`

La signature de l'adaptateur est la suivante : `chain`

```
fn chain<U>(self, other: U) -> impl Iterator<Item=Self::Item>
    where Self: Sized, U: IntoIterator<Item=Self::Item>;
```

En d'autres termes, vous pouvez enchaîner un itérateur avec n'importe quel itérable qui produit le même type d'article.

Par exemple:

```
let v: Vec<i32> = (1..4).chain([20, 30, 40]).collect();
assert_eq!(v, [1, 2, 3, 20, 30, 40]);
```

Un itérateur est réversible, si ses deux itérateurs sous-jacents sont : `chain`

```
let v: Vec<i32> = (1..4).chain([20, 30, 40]).rev().collect();
assert_eq!(v, [40, 30, 20, 3, 2, 1]);
```

Un itérateur vérifie si chacun des deux itérateurs sous-jacents est revenu et dirige et appelle l'un ou l'autre, selon le cas. `chain None next next_back`

énumérer

L'adaptateur du trait attache un index en cours d'exécution à la séquence, en prenant un itérateur qui produit des éléments et en renvoyant un itérateur qui produit des paires. Cela semble trivial à première vue, mais il est utilisé étonnamment souvent. `Iterator enumerate A, B, C, ... (0, A), (1, B), (2, C), ...`

Les consommateurs peuvent utiliser cet indice pour distinguer un article d'un autre et établir le contexte dans lequel traiter chacun d'eux. Par exemple, le traceur de jeu de Mandelbrot du [chapitre 2](#) divise l'image en huit bandes horizontales et attribue chacune d'elles à un fil différent. Ce code permet d'indiquer à chaque thread à quelle partie de l'image correspond sa bande. `enumerate`

Il commence par un tampon rectangulaire de pixels:

```
let mut pixels = vec![0; columns * rows];
```

Ensuite, il permet de diviser l'image en bandes horizontales, une par thread: `chunks_mut`

```

let threads = 8;
let band_rows = rows / threads + 1;
...
let bands: Vec<&mut [u8]> = pixels.chunks_mut(band_rows * columns).coll

```

Et puis il itére sur les bandes, en commençant un fil pour chacune d'elles:

```

for (i, band) in bands.into_iter().enumerate() {
    let top = band_rows * i;
    // start a thread to render rows `top..top + band_rows`
    ...
}

```

Chaque itération obtient une paire , où est la tranche du tampon de pixels dans lequel le thread doit dessiner, et est l'index de cette bande dans l'image globale, gracieuseté de l'adaptateur. Compte tenu des limites du tracé et de la taille des bandes, il suffit d'informations pour que le fil détermine quelle partie de l'image lui a été attribuée et donc dans quoi dessiner. (i, band) band &mut [u8] i enumerate band

Vous pouvez considérer les paires qui produisent comme analogues aux paires que vous obtenez lorsque vous itérez sur une collection associative ou une autre collection associative. Si vous itérez sur une tranche ou un vecteur, le est la « clé » sous laquelle le apparaît. (index, item) enumerate (key, value) HashMap index item

fermeture éclair

L'adaptateur combine deux itérateurs en un seul itérateur qui produit des paires contenant une valeur de chaque itérateur, comme une fermeture à glissière reliant ses deux côtés en une seule couture. L'itérateur zippé se termine lorsque l'un des deux itérateurs sous-jacents se termine. zip

Par exemple, vous pouvez obtenir le même effet que l'adaptateur en compressant la plage d'extrémité illimitée avec l'autre itérateur :enumerate 0..

```

let v: Vec<_> = (0..).zip("ABCD".chars()).collect();
assert_eq!(v, vec![(0, 'A'), (1, 'B'), (2, 'C'), (3, 'D')]);

```

En ce sens, vous pouvez penser à une généralisation de : alors qu'attache des indices à la séquence, attache les éléments de tout itérateur arbitraire. Nous avons déjà suggéré que cela puisse aider à fournir un con-

texte pour le traitement des éléments; est une façon plus souple de faire de même. `zip` `enumerate` `enumerate` `zip` `enumerate` `zip`

L'argument de `n'a pas besoin d'être un itérateur lui-même; il peut être n'importe quel qu'il est possible de le faire : zip`

```
use std::iter::repeat;

let endings = ["once", "twice", "chicken soup with rice"];
let rhyme: Vec<_> = repeat("going")
    .zip(endings)
    .collect();
assert_eq!(rhyme, vec![("going", "once"),
                      ("going", "twice"),
                      ("going", "chicken soup with rice")]);
```

by_ref

Tout au long de cette section, nous avons attaché des adaptateurs à des itérateurs. Une fois que vous l'avez fait, pouvez-vous jamais retirer l'adaptateur à nouveau? Habituellement, non: les adaptateurs prennent possession de l'itérateur sous-jacent et ne fournissent aucune méthode pour le rendre.

La méthode d'un itérateur emprunte une référence modifiable à l'itérateur afin que vous puissiez appliquer des adaptateurs à la référence. Lorsque vous avez fini de consommer des articles à partir de ces adaptateurs, vous les déposez, l'emprunt se termine et vous retrouvez l'accès à votre itérateur d'origine. `by_ref`

Par exemple, plus tôt dans le chapitre, nous avons montré comment utiliser et traiter les lignes d'en-tête et le corps d'un message électronique. Mais que se passe-t-il si vous voulez faire les deux, en utilisant le même itérateur sous-jacent? En utilisant `, nous pouvons utiliser pour gérer les en-têtes, et lorsque cela est fait, récupérez l'itérateur sous-jacent, qui est parti exactement en position pour gérer le corps du message: take_while skip_while by_ref take_while take_while`

```
let message = "To: jimb\r\n"
              From: id\r\n"
              \r\n"
              Ooooooh, donuts!!\r\n";
let mut lines = message.lines();
```

```

    println!("Headers:");
    for header in lines.by_ref().take_while(|l| !l.is_empty()) {
        println!("{}" , header);
    }

    println!("\nBody:");
    for body in lines {
        println!("{}" , body);
    }
}

```

L'appel emprunte une référence mutable à l'itérateur, et c'est de cette référence que l'itérateur s'approprie. Cet itérateur sort de la portée à la fin de la première boucle, ce qui signifie que l'emprunt est terminé, de sorte que vous pouvez l'utiliser à nouveau dans la deuxième boucle. Cela imprime les éléments suivants

```
: lines.by_ref() take_while for lines for
```

```
Headers:
```

```
To: jimb
```

```
From: id
```

```
Body:
```

```
Oooooh, donuts!!
```

La définition de l'adaptateur est triviale : elle renvoie une référence mutable à l'itérateur. Ensuite, la bibliothèque standard inclut cette étrange petite implémentation : `by_ref`

```

impl<'a, I: Iterator + ?Sized> Iterator for &'a mut I {
    type Item = I::Item;
    fn next(&mut self) -> Option<I::Item> {
        (**self).next()
    }
    fn size_hint(&self) -> (usize, Option<usize>) {
        (**self).size_hint()
    }
}

```

En d'autres termes, si est un type d'itérateur, alors est un itérateur aussi, dont et les méthodes s'en remettent à son référent. Lorsque vous appelez un adaptateur sur une référence modifiable à un itérateur, l'adaptateur prend possession de la *référence*, et non de l'itérateur lui-même. C'est

juste un emprunt qui se termine lorsque l'adaptateur sort de sa portée. `I &mut I next size_hint`

cloné, copié

L'adaptateur prend un itérateur qui produit des références et renvoie un itérateur qui produit des valeurs clonées à partir de ces références, un peu comme `.clone()`. Naturellement, le type de référent doit implémenter `.Clone`. Par exemple:

```
let a = ['1', '2', '3', '∞'];

assert_eq!(a.iter().next(), Some(&'1'));
assert_eq!(a.iter().cloned().next(), Some('1'));
```

L'adaptateur est la même idée, mais plus restrictive : le type de référent doit implémenter `.Clone`. Un appel `clone` est à peu près le même que `.copy`. Étant donné que chaque type qui implémente également `.Copy`, est strictement plus général, mais en fonction du type d'élément, un appel peut effectuer des quantités arbitraires d'allocation et de copie. Si vous supposez que cela ne se produirait jamais parce que votre type d'article est quelque chose de simple, il est préférable d'utiliser pour que le vérificateur de type vérifie vos hypothèses.

`copied` `Copy iter.copied()` `iter.map(|r|`

`*r) Copy Clone cloned clone copied`

cycle

L'adaptateur renvoie un itérateur qui répète à l'infini la séquence produite par l'itérateur sous-jacent. L'itérateur sous-jacent doit être implémenté afin de pouvoir enregistrer son état initial et le réutiliser chaque fois que le cycle redémarre.

`cycle std::clone::Clone cycle`

Par exemple:

```
let dirs = ["North", "East", "South", "West"];
let mut spin = dirs.iter().cycle();
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
assert_eq!(spin.next(), Some(&"South"));
assert_eq!(spin.next(), Some(&"West"));
assert_eq!(spin.next(), Some(&"North"));
assert_eq!(spin.next(), Some(&"East"));
```

Ou, pour une utilisation vraiment gratuite des itérateurs :

```

use std::iter::{once, repeat};

let fizzes = repeat("").take(2).chain(once("fizz")).cycle();
let buzzes = repeat("").take(4).chain(once("buzz")).cycle();
let fizzes_buzzes = fizzes.zip(buzzes);

let fizz_buzz = (1..100).zip(fizzes_buzzes)
    .map(|tuple|
        match tuple {
            (i, ("", "")) => i.to_string(),
            (_, (fizz, buzz)) => format!("{}{}", fizz, buzz)
        });
}

for line in fizz_buzz {
    println!("{}", line);
}

```

Cela joue un jeu de mots pour enfants, maintenant parfois utilisé comme une question d'entretien d'embauche pour les codeurs, dans lequel les joueurs comptent à tour de rôle, remplaçant tout nombre divisible par trois par le mot , et tout nombre divisible par cinq avec . Les nombres divisibles par les deux deviennent . fizz buzz fizzbuzz

Itérateurs consommateurs

Jusqu'à présent, nous avons couvert la création d'itérateurs et leur adaptation en nouveaux itérateurs; ici, nous terminons le processus en montrant des moyens de les consommer.

Bien sûr, vous pouvez consommer un itérateur avec une boucle ou appeler explicitement, mais il existe de nombreuses tâches courantes que vous ne devriez pas avoir à écrire encore et encore. Le trait fournit un large choix de méthodes pour couvrir beaucoup d'entre eux. `for next Iterator`

Accumulation simple: nombre, somme, produit

La méthode tire des éléments d'un itérateur jusqu'à ce qu'il revienne et vous indique combien il en a obtenu. Voici un programme court qui compte le nombre de lignes sur son entrée standard : `count None`

```

use std::io::prelude::*;

fn main() {

```

```

let stdin = std::io::stdin();
println!("{}", stdin.lock().lines().count());
}

```

Les méthodes et calculent la somme ou le produit des éléments de l'itérateur, qui doivent être des entiers ou des nombres à virgule flottante
: sum product

```

fn triangle(n: u64) -> u64 {
    (1..=n).sum()
}

assert_eq!(triangle(20), 210);

fn factorial(n: u64) -> u64 {
    (1..=n).product()
}

assert_eq!(factorial(20), 2432902008176640000);

```

(Vous pouvez étendre et travailler avec d'autres types en implémentant les et traits, que nous ne décrirons pas dans ce livre.)
sum product std::iter::Sum std::iter::Product

max, min

Les méthodes et sur le retour le moins ou le plus grand article que l'itérateur produit. Le type d'élément de l'itérateur doit être implémenté afin que les éléments puissent être comparés les uns aux autres. Par exemple:
min max Iterator std::cmp::Ord

```

assert_eq!([-2, 0, 1, 0, -2, -5].iter().max(), Some(&1));
assert_eq!([-2, 0, 1, 0, -2, -5].iter().min(), Some(&-5));

```

Ces méthodes renvoient un afin qu'elles puissent revenir si l'itérateur ne produit aucun élément. Option<Self::Item> None

Comme expliqué dans [« Comparaisons d'équivalence »](#), les types à virgule flottante de Rust et implémentent uniquement , pas , de sorte que vous ne pouvez pas utiliser les méthodes et pour calculer le plus petit ou le plus grand d'une séquence de nombres à virgule flottante. Ce n'est pas un aspect populaire de la conception de Rust, mais c'est délibéré: il n'est pas clair ce que de telles fonctions devraient faire avec les valeurs IEEE NaN. Le simple fait de les ignorer risquerait de masquer des problèmes plus

graves dans le

```
code. f32 f64 std::cmp::PartialOrd std::cmp::Ord min max
```

Si vous savez comment vous souhaitez gérer les valeurs NaN, vous pouvez utiliser les méthodes et itérateur à la place, qui vous permettent de fournir votre propre fonction de comparaison. `max_by` `min_by`

max_by, min_by

Les méthodes et renvoient l'élément maximal ou minimum produit par l'itérateur, tel que déterminé par une fonction de comparaison que vous fournissez : `max_by` `min_by`

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

// Compare two f64 values. Panic if given a NaN.
fn cmp(lhs: &f64, rhs: &f64) -> Ordering {
    lhs.partial_cmp(rhs).unwrap()
}

let numbers = [1.0, 4.0, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0));
assert_eq!(numbers.iter().copied().min_by(cmp), Some(1.0));

let numbers = [1.0, 4.0, std::f64::NAN, 2.0];
assert_eq!(numbers.iter().copied().max_by(cmp), Some(4.0)); // panics
```

Les méthodes et transmettent les éléments à la fonction de comparaison par référence afin qu'ils puissent fonctionner efficacement avec n'importe quel type d'itérateur, donc s'attend à prendre ses arguments par référence, même si nous avons l'habitude d'obtenir un itérateur qui produit des éléments. `max_by` `min_by` `cmp` `copied` `f64`

max_by_key, min_by_key

Les méthodes et sur vous permettent de sélectionner l'élément maximum ou minimum déterminé par une fermeture appliquée à chaque élément. La fermeture peut sélectionner un champ de l'élément ou effectuer un calcul sur les éléments. Étant donné que vous êtes souvent intéressé par les données associées à un minimum ou à un maximum, et pas seulement à l'extremum lui-même, ces fonctions sont souvent plus utiles que et .

Leurs signatures sont les suivantes

```
:max_by_key min_by_key Iterator min max
```

```

fn min_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;

fn max_by_key<B: Ord, F>(self, f: F) -> Option<Self::Item>
    where Self: Sized, F: FnMut(&Self::Item) -> B;

```

Autrement dit, étant donné une fermeture qui prend un article et retourne tout type commandé, retournez l'article pour lequel la fermeture a retourné le maximum ou le minimum , ou si aucun article n'a été produit. B B None

Par exemple, si vous devez scanner une table de hachage des villes pour trouver les villes avec les plus grandes et les plus petites populations, vous pouvez écrire:

```

use std::collections::HashMap;

let mut populations = HashMap::new();
populations.insert("Portland", 583_776);
populations.insert("Fossil", 449);
populations.insert("Greenhorn", 2);
populations.insert("Boring", 7_762);
populations.insert("The Dalles", 15_340);

assert_eq!(populations.iter().max_by_key(|&(_name, pop)| pop),
           Some((&"Portland", &583_776)));
assert_eq!(populations.iter().min_by_key(|&(_name, pop)| pop),
           Some((&"Greenhorn", &2)));

```

La fermeture est appliquée à chaque élément produit par l'itérateur et renvoie la valeur à utiliser à des fins de comparaison, dans ce cas, la population de la ville. La valeur renvoyée est l'article entier, pas seulement la valeur renvoyée par la fermeture. (Naturellement, si vous faisiez souvent des requêtes comme celle-ci, vous voudriez probablement trouver un moyen plus efficace de trouver les entrées que de faire une recherche linéaire dans le tableau.) |&(_name, pop)| pop

Comparaison des séquences d'éléments

Vous pouvez utiliser les opérateurs et pour comparer des chaînes, des vecteurs et des tranches, en supposant que leurs éléments individuels peuvent être comparés. Bien que les itérateurs ne prennent pas en charge les opérateurs de comparaison de Rust, ils fournissent des méthodes telles

que et qui font le même travail, en tirant des paires d'éléments des itérateurs et en les comparant jusqu'à ce qu'une décision puisse être prise. Par exemple: < == eq lt

```
let packed = "Helen of Troy";
let spaced = "Helen    of      Troy";
let obscure = "Helen of Sandusky"; // nice person, just not famous

assert!(packed != spaced);
assert!(packed.split_whitespace().eq(spaced.split_whitespace()));

// This is true because ' ' < 'o'.
assert!(spaced < obscure);

// This is true because 'Troy' > 'Sandusky'.
assert!(spaced.split_whitespace().gt(obscure.split_whitespace()));
```

Appels à renvoyer des itérateurs sur les mots séparés par des espaces blancs de la chaîne. L'utilisation des méthodes et sur ces itérateurs effectue une comparaison mot par mot, au lieu d'une comparaison caractère par caractère. Tout cela est possible car les implémentations et .split_whitespace eq gt &str PartialOrd PartialEq

Les itérateurs fournissent les méthodes et les méthodes pour les comparaisons d'égalité, et , , , et les méthodes pour les comparaisons ordonnées. Les méthodes et se comportent comme les méthodes correspondantes des et traits. eq ne lt le gt ge cmp partial_cmp Ord PartialOrd

tout et n'importe quoi

Les méthodes et appliquent une fermeture à chaque article produit par l'itérateur et retournent si la fermeture revient pour un article ou pour tous les articles : any all true true

```
let id = "Iterator";

assert!( id.chars().any(char::is_uppercase));
assert!(!id.chars().all(char::is_uppercase));
```

Ces méthodes ne consomment que le nombre d'éléments nécessaires pour déterminer la réponse. Par exemple, si la fermeture revient pour un article donné, elle revient immédiatement, sans tirer d'autres éléments de l'itérateur. true any true

position, rposition et ExactSizeIterator

La méthode applique une fermeture à chaque élément de l’itérateur et renvoie l’index du premier élément pour lequel la fermeture renvoie . Plus précisément, il renvoie un de l’index : si la fermeture ne renvoie aucun élément, renvoie . Il arrête de dessiner des éléments dès que la fermeture revient. Par

exemple: position true Option true position None true

```
let text = "Xerxes";
assert_eq!(text.chars().position(|c| c == 'e'), Some(1));
assert_eq!(text.chars().position(|c| c == 'z'), None);
```

La méthode est la même, sauf qu’elle recherche à partir de la droite. Par exemple : rposition

```
let bytes = b"Xerxes";
assert_eq!(bytes.iter().rposition(|&c| c == b'e'), Some(4));
assert_eq!(bytes.iter().rposition(|&c| c == b'x'), Some(0));
```

La méthode nécessite un itérateur réversible afin de pouvoir dessiner des éléments à partir de l’extrême droite de la séquence. Il nécessite également un itérateur de taille exacte afin qu’il puisse attribuer des indices de la même manière, en commençant par à gauche. Un itérateur de taille exacte est celui qui implémente le

trait: rposition position 0 std::iter::ExactSizeIterator

```
trait ExactSizeIterator: Iterator {
    fn len(&self) -> usize { ... }
    fn is_empty(&self) -> bool { ... }
}
```

La méthode renvoie le nombre d’éléments restants et la méthode renvoie si l’itération est terminée. len is_empty true

Naturellement, tous les itérateurs ne savent pas combien d’articles ils produiront à l’avance. Par exemple, l’itérateur utilisé précédemment ne le fait pas (puisque UTF-8 est un codage à largeur variable), vous ne pouvez donc pas l’utiliser sur les chaînes. Mais un itérateur sur un tableau d’octets connaît certainement la longueur du tableau, il peut donc implémenter .str::chars rposition ExactSizeIterator

plier et rfold

La méthode est un outil très général pour accumuler une sorte de résultat sur toute la séquence d'éléments produits par un itérateur. Compte tenu d'une valeur initiale, que nous appellerons *l'accumulateur*, et d'une fermeture, applique à plusieurs reprises la fermeture à l'accumulateur actuel et à l'élément suivant de l'itérateur. La valeur renvoyée par la fermeture est considérée comme le nouvel accumulateur, à transmettre à la fermeture avec l'élément suivant. La valeur finale de l'accumulateur est ce qui lui-même retourne. Si la séquence est vide, renvoie simplement l'accumulateur initial. `fold` `fold` `fold`

Beaucoup d'autres méthodes pour consommer les valeurs d'un itérateur peuvent être écrites comme des utilisations de : `fold`

```
let a = [5, 6, 7, 8, 9, 10];

assert_eq!(a.iter().fold(0, |n, _| n+1), 6);           // count
assert_eq!(a.iter().fold(0, |n, i| n+i), 45);         // sum
assert_eq!(a.iter().fold(1, |n, i| n*i), 151200);     // product

// max
assert_eq!(a.iter().cloned().fold(i32::min_value(), std::cmp::max),
          10);
```

La signature de la méthode est la suivante : `fold`

```
fn fold<A, F>(self, init: A, f: F) -> A
    where Self: Sized, F: FnMut(A, Self::Item) -> A;
```

Voici le type d'accumulateur. L'argument est un , tout comme le premier argument et la valeur de retour de la fermeture, et la valeur de retour de lui-même. `A` `init` `A` `fold`

Notez que les valeurs de l'accumulateur sont déplacées dans et hors de la fermeture, de sorte que vous pouvez utiliser avec des types non-accumulateur : `fold` `Copy`

```
let a = ["Pack", "my", "box", "with",
         "five", "dozen", "liquor", "jugs"];

// See also: the `join` method on slices, which won't
// give you that extra space at the end.
let pangram = a.iter()
    .fold(String::new(), |s, w| s + w + " ");
assert_eq!(pangram, "Pack my box with five dozen liquor jugs");
```

La méthode est identique à , sauf qu'elle nécessite un itérateur à double extrémité et traite ses éléments du dernier au premier : `rfold fold`

```
let weird_pangram = a.iter()
    .rfold(String::new(), |s, w| s + w + " ");
assert_eq!(weird_pangram, "jugs liquor dozen five with box my Pack ");
```

try_fold et try_rfold

La méthode est identique à , sauf que l'itération peut se fermer plus tôt, sans consommer toutes les valeurs de l'itérateur. La valeur renvoyée par la fermeture que vous passez indique si elle doit revenir immédiatement ou continuer à plier les éléments de l'itérateur. `try_fold fold try_fold`

Votre fermeture peut renvoyer l'un des types suivants, indiquant comment le pliage doit se dérouler:

- Si votre fermeture revient, peut-être parce qu'elle effectue des E/S ou effectue une autre opération faillible, alors le retour indique de continuer le pliage, avec comme nouvelle valeur d'accumulateur. Le retour provoque l'arrêt immédiat du pliage. La valeur finale du pli est une valeur portante sur l'accumulateur final, ou l'erreur renvoyée par la fermeture. `Result<T, E> Ok(v) try_fold v Err(e) Result`
 - Si votre fermeture revient , indique alors que le pliage doit continuer avec comme nouvelle valeur d'accumulateur et indique que l'itération doit s'arrêter immédiatement. La valeur finale du pli est également un fichier. `Option<T> Some(v) v None Option`
 - Enfin, la fermeture peut renvoyer une valeur. Ce type est un enum avec deux variantes, et , ce qui signifie continuer avec une nouvelle valeur d'accumulateur , ou s'arrêter tôt. Le résultat du pli est une valeur : si le pli a consommé l'itérateur entier, donnant la valeur finale de l'accumulateur ; ou , si la fermeture a renvoyé cette valeur. `std::ops::ControlFlow Continue(c) Break(b) c ControlFlow Continue(v) v Break(b)`
- Continuer(c) et se comporter exactement comme et . L'avantage d'utiliser au lieu de est que cela rend votre code un peu plus lisible lorsqu'une sortie anticipée n'indique pas une erreur, mais simplement que la réponse est prête tôt. Nous en montrons un exemple ci-dessous.
- ```
Break(b) Ok(c) Err(b) ControlFlow Result
```

Voici un programme qui additionne les nombres lus à partir de son entrée standard :

```
use std::error::Error;
use std::io::prelude::*;
use std::str::FromStr;

fn main() -> Result<(), Box<dyn Error>> {
 let stdin = std::io::stdin();
 let sum = stdin.lock()
 .lines()
 .try_fold(0, |sum, line| -> Result<u64, Box<dyn Error>> {
 Ok(sum + u64::from_str(&line?.trim())?)
 })?;
 println!("{}", sum);
 Ok(())
}
```

L’itérateur sur les flux d’entrée mis en mémoire tampon produit des éléments de type , et l’analyse de l’entier peut également échouer. L’utilisation ici permet à la fermeture de retourner , de sorte que nous pouvons utiliser l’opérateur pour propager les défaillances de la fermeture à la fonction. lines Result<String,

```
std::io::Error> String try_fold Result<u64, ...> ? main
```

Parce qu’il est si flexible, il est utilisé pour mettre en œuvre de nombreuses autres méthodes de consommation. Par exemple, voici une implémentation de :try\_fold Iterator all

```
fn all<P>(&mut self, mut predicate: P) -> bool
 where P: FnMut(Self::Item) -> bool,
 Self: Sized
{
 use std::ops::ControlFlow::*;
 self.try_fold(() , |_, item| {
 if predicate(item) { Continue(()) } else { Break(()) }
 }) == Continue(())
}
```

Notez que cela ne peut pas être écrit avec ordinaire : promet d’arrêter de consommer des éléments de l’itérateur sous-jacent dès qu’il renvoie false, mais consomme toujours l’itérateur entier. fold all predicate fold

Si vous implémentez votre propre type d’itérateur, il vaut la peine d’examiner si votre itérateur pourrait implémenter plus efficacement que la

définition par défaut du trait. Si vous pouvez accélérer, toutes les autres méthodes construites dessus en bénéficieront également. `try_fold` `Iterator` `try_fold`

La méthode, comme son nom l'indique, est la même que , sauf qu'elle tire des valeurs de l'arrière, au lieu de l'avant, et nécessite un itérateur à double extrémité. `try_rfold` `try_fold`

## nième, nth\_back

La méthode prend un index , ignore autant d'éléments de l'itérateur et renvoie l'élément suivant, ou si la séquence se termine avant ce point. L'appel équivaut à `.nth n None .nth(0) .next()`

Il ne prend pas possession de l'itérateur comme le ferait un adaptateur, vous pouvez donc l'appeler plusieurs fois:

```
let mut squares = (0..10).map(|i| i*i);

assert_eq!(squares.nth(4), Some(16));
assert_eq!(squares.nth(0), Some(25));
assert_eq!(squares.nth(6), None);
```

Sa signature est présentée ici :

```
fn nth(&mut self, n: usize) -> Option<Self::Item>
 where Self: Sized;
```

La méthode est à peu près la même, sauf qu'elle puise à l'arrière d'un itérateur à double extrémité. L'appel équivaut à : il renvoie le dernier élément, ou si l'itérateur est

vide. `nth_back .nth_back(0) .next_back() None`

## dernier

La méthode renvoie le dernier élément produit par l'itérateur ou s'il est vide. Sa signature est la suivante : `last None`

```
fn last(self) -> Option<Self::Item>;
```

Par exemple:

```
let squares = (0..10).map(|i| i*i);
assert_eq!(squares.last(), Some(81));
```

Cela consomme tous les éléments de l'itérateur en commençant par l'avant, même si l'itérateur est réversible. Si vous avez un itérateur réversible et que vous n'avez pas besoin de consommer tous ses éléments, vous devez simplement écrire `.iter.next_back()`

## find, rfind et find\_map

La méthode tire des éléments d'un itérateur, en renvoyant le premier élément pour lequel la fermeture donnée est renvoyée , ou si la séquence se termine avant qu'un élément approprié ne soit trouvé. Sa signature est :  
`:find true None`

```
fn find<P>(&mut self, predicate: P) -> Option<Self::Item>
 where Self: Sized,
 P: FnMut(&Self::Item) -> bool;
```

La méthode est similaire, mais elle nécessite un itérateur à double extrémité et recherche les valeurs de l'arrière vers l'avant, en renvoyant le *dernier* élément pour lequel la fermeture renvoie `.rfind true`

Par exemple, en utilisant le tableau des villes et des populations de « [max by key, min by key](#) », vous pourriez écrire:

```
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 1_000_000),
 None);
assert_eq!(populations.iter().find(|&(_name, &pop)| pop > 500_000),
 Some((&"Portland", &583_776)));
```

Aucune des villes du tableau n'a une population supérieure à un million, mais il y a une ville avec un demi-million d'habitants.

Parfois, votre clôture n'est pas seulement un simple prédictat jetant un jugement booléen sur chaque élément et passant à autre chose: il peut s'agir de quelque chose de plus complexe qui produit une valeur intéressante en soi. Dans ce cas, c'est exactement ce que vous voulez. Sa signature est :`find_map`

```
fn find_map<B, F>(&mut self, f: F) -> Option where
 F: FnMut(Self::Item) -> Option;
```

C'est comme , sauf qu'au lieu de retourner , la fermeture devrait renvoyer une certaine valeur. renvoie le premier qui est

```
.find bool Option find_map Option Some
```

Par exemple, si nous avons une base de données des parcs de chaque ville, nous pourrions vouloir voir si l'un d'entre eux sont des volcans et fournir le nom du parc si oui:

```
let big_city_with_volcano_park = populations.iter()
 .find_map(|(&city, _)| {
 if let Some(park) = find_volcano_park(city, &parks) {
 // find_map returns this value, so our caller knows
 // *which* park we found.
 return Some((city, park.name));
 }

 // Reject this item, and continue the search.
 None
 });

assert_eq!(big_city_with_volcano_park,
 Some(("Portland", "Mt. Tabor Park")));
```

## Building Collections: collect et FromIterator

Tout au long du livre, nous avons utilisé la méthode pour construire des vecteurs contenant les éléments d'un itérateur. Par exemple, dans [le chapitre 2](#), nous avons appelé pour obtenir un itérateur sur les arguments de ligne de commande du programme, puis appelé la méthode de cet itérateur pour les rassembler dans un vecteur

```
:collect std::env::args() collect
```

```
let args: Vec<String> = std::env::args().collect();
```

Mais n'est pas spécifique aux vecteurs: en fait, il peut construire n'importe quel type de collection à partir de la bibliothèque standard de Rust, tant que l'itérateur produit un type d'élément approprié: collect

```
use std::collections::{HashSet, BTreeSet, LinkedList, HashMap, BTreeMap}

let args: HashSet<String> = std::env::args().collect();
let args: BTreeSet<String> = std::env::args().collect();
let args: LinkedList<String> = std::env::args().collect();

// Collecting a map requires (key, value) pairs, so for this example,
// zip the sequence of strings with a sequence of integers.
```

```

let args: HashMap<String, usize> = std::env::args().zip(0...).collect();
let args: BTreeMap<String, usize> = std::env::args().zip(0...).collect();

// and so on

```

Naturellement, elle-même ne sait pas comment construire tous ces types. Au contraire, lorsqu'un type de collection aime ou sait comment se construire à partir d'un itérateur, il implémente le trait, pour lequel il ne s'agit que d'un placage

pratique: collect Vec HashMap std::iter::FromIterator collect

```

trait FromIterator<A>: Sized {
 fn from_iter<T: IntoIterator<Item=A>>(iter: T) -> Self;
}

```

Si un type de collection implémente , sa fonction associée au type génère une valeur de ce type à partir d'un élément de production itérable de type .FromIterator<A> from\_iter A

Dans le cas le plus simple, l'implémentation pourrait simplement construire une collection vide, puis ajouter les éléments de l'itérateur un par un. Par exemple, la mise en œuvre de fonctionne de cette façon. std::collections::LinkedList FromIterator

Cependant, certains types peuvent faire mieux que cela. Par exemple, la construction d'un vecteur à partir d'un itérateur pourrait être aussi simple que : iter

```

let mut vec = Vec::new();
for item in iter {
 vec.push(item)
}
vec

```

Mais ce n'est pas idéal : à mesure que le vecteur grandit, il peut avoir besoin d'étendre sa mémoire tampon, nécessitant un appel à l'allocateur de tas et une copie des éléments existants. Les vecteurs prennent des mesures algorithmiques pour maintenir cette surcharge basse, mais s'il y avait un moyen d'allouer simplement un tampon de la bonne taille pour commencer, il n'y aurait pas besoin de redimensionner du tout.

C'est là qu'intervient la méthode du trait : Iterator size\_hint

```

trait Iterator {
 ...
 fn size_hint(&self) -> (usize, Option<usize>) {
 (0, None)
 }
}

```

Cette méthode renvoie une limite inférieure et une limite supérieure facultative sur le nombre d'éléments que l'itérateur produira. La définition par défaut renvoie zéro comme limite inférieure et refuse de nommer une limite supérieure, en disant, en fait, « Je n'en ai aucune idée », mais de nombreux itérateurs peuvent faire mieux que cela. Un itérateur sur un , par exemple, sait exactement combien d'éléments il produira, tout comme un itérateur sur un ou . Ces itérateurs fournissent leurs propres définitions spécialisées pour . Range Vec HashMap size\_hint

Ces limites sont exactement les informations dont l'implémentation a besoin pour dimensionner correctement le tampon du nouveau vecteur dès le départ. Les insertions vérifient toujours que la mémoire tampon est suffisamment grande, de sorte que même si l'indice est incorrect, seules les performances sont affectées, pas la sécurité. D'autres types peuvent prendre des mesures similaires: par exemple, et également utiliser pour choisir une taille initiale appropriée pour leur table de hachage. Vec FromIterator HashSet HashMap Iterator::size\_hint

Une remarque sur l'inférence de type : en haut de cette section, il est un peu étrange de voir le même appel, , produire quatre types de collections différents en fonction de son contexte. Le type de retour de est son paramètre de type, de sorte que les deux premiers appels sont équivalents à ce qui suit : std::env::args().collect() collect

```

let args = std::env::args().collect::<Vec<String>>();
let args = std::env::args().collect::<HashSet<String>>();

```

Mais tant qu'il n'y a qu'un seul type qui pourrait éventuellement fonctionner comme argument de , l'inférence de type de Rust vous le fournira. Lorsque vous épelez le type de , vous vous assurez que c'est le cas. collect args

## Le trait d'extension

Si un type implémente le trait, sa méthode ajoute les éléments d'un produit itérable à la collection : `std::iter::Extend` `extend`

```
let mut v: Vec<i32> = (0..5).map(|i| 1 << i).collect();
v.extend([31, 57, 99, 163]);
assert_eq!(v, [1, 2, 4, 8, 16, 31, 57, 99, 163]);
```

Toutes les collections standard implémentent , elles ont donc toutes cette méthode; il en va de même pour . Les tableaux et les tranches, qui ont une longueur fixe, ne le font pas. `Extend String`

La définition du trait est la suivante :

```
trait Extend<A> {
 fn extend<T>(&mut self, iter: T)
 where T: IntoIterator<Item=A>;
}
```

Évidemment, c'est très similaire à : cela crée une nouvelle collection, alors qu'elle étend une collection existante. En fait, plusieurs implementations de dans la bibliothèque standard créent simplement une nouvelle collection vide, puis appellent pour la remplir. Par exemple, l'implémentation de `for` fonctionne de cette

façon: `std::iter::FromIterator` `Extend FromIterator` `extend FromIterator` `std::collections::LinkedList`

```
impl<T> FromIterator<T> for LinkedList<T> {
 fn from_iter<I: IntoIterator<Item = T>>(&mut self, iter: I) -> Self {
 let mut list = Self::new();
 list.extend(iter);
 list
 }
}
```

## partition

La méthode divise les éléments d'un itérateur entre deux collections, en utilisant une fermeture pour décider de l'appartenance de chaque élément : `partition`

```
let things = ["doorknob", "mushroom", "noodle", "giraffe", "grapefruit"]

// Amazing fact: the name of a living thing always starts with an
```

```
// odd-numbered letter.

let (living, nonliving): (Vec<&str>, Vec<&str>)
 = things.iter().partition(|name| name.as_bytes()[0] & 1 != 0);

assert_eq!(living, vec!["mushroom", "giraffe", "grapefruit"]);
assert_eq!(nonliving, vec!["doorknob", "noodle"]);
```

Comme , peut faire n'importe quel type de collections que vous aimez, bien que les deux doivent être du même type. Et comme , vous devrez spécifier le type de retour : l'exemple précédent écrit le type de et permet à l'inférence de type de choisir les bons paramètres de type pour l'appel à .collect partition collect living nonliving partition

La signature de est la suivante : partition

```
fn partition<B, F>(self, f: F) -> (B, B)
where Self: Sized,
 B: Default + Extend<Self::Item>,
 F: FnMut(&Self::Item) -> bool;
```

Alors que nécessite son type de résultat pour implémenter , à la place nécessite , que toutes les collections Rust implémentent en renvoyant une collection vide, et

```
.collect FromIterator partition std::default::Default std::
default::Extend
```

D'autres langages proposent des opérations qui divisent simplement l'itérateur en deux itérateurs, au lieu de créer deux collections. Mais ce n'est pas un bon choix pour Rust : les éléments tirés de l'itérateur sous-jacent mais pas encore tirés de l'itérateur partitionné approprié devraient être mis en mémoire tampon quelque part ; vous finiriez par construire une collection d'une sorte ou d'une autre à l'interne, de toute façon. partition

## for\_each et try\_for\_each

La méthode applique simplement une fermeture à chaque élément: for\_each

```
["doves", "hens", "birds"].iter()
.zip(["turtle", "french", "calling"])
.zip(2..5)
.rev()
.map(|((item, kind), quantity)| {
```

```

 format!("{} {} {}", quantity, kind, item)
 })
 .for_each(|gift| {
 println!("You have received: {}", gift);
 });
}

```

Cette impression est :

```

You have received: 4 calling birds
You have received: 3 french hens
You have received: 2 turtle doves

```

Ceci est très similaire à une boucle simple, dans laquelle vous pouvez également utiliser des structures de contrôle comme et . Mais de longues chaînes d'appels d'adaptateurs comme celle-ci sont un peu gênantes dans les boucles: for break continue for

```

for gift in ["doves", "hens", "birds"].iter()
 .zip(["turtle", "french", "calling"])
 .zip(2..5)
 .rev()
 .map(|((item, kind), quantity)| {
 format!("{} {} {}", quantity, kind, item)
 })
{
 println!("You have received: {}", gift);
}

```

Le motif étant lié, , peut se retrouver assez loin du corps de boucle dans lequel il est utilisé. gift

Si votre fermeture doit être faillible ou sortir plus tôt, vous pouvez utiliser : try\_for\_each

```

...
.try_for_each(|gift| {
 writeln!(&mut output_file, "You have received: {}", gift)
})?;

```

## Mise en œuvre de vos propres itérateurs

Vous pouvez implémenter les caractéristiques et pour vos propres types, ce qui rend tous les adaptateurs et consommateurs présentés dans ce chapitre disponibles pour utilisation, ainsi que de nombreux autres codes de bibliothèque et de caisse écrits pour fonctionner avec l'interface d'itérateur standard. Dans cette section, nous allons montrer un itérateur simple sur un type de plage, puis un itérateur plus complexe sur un type d'arbre binaire. `IntoIterator` `Iterator`

Supposons que nous ayons le type de plage suivant (simplifié à partir du type de la bibliothèque standard) : `std::ops::Range<T>`

```
struct I32Range {
 start: i32,
 end: i32
}
```

L'itération sur un nécessite deux éléments d'état : la valeur actuelle et la limite à laquelle l'itération doit se terminer. Cela se trouve être un bon ajustement pour le type lui-même, en utilisant comme valeur suivante et comme limite. Vous pouvez donc mettre en œuvre de la sorte

```
: I32Range I32Range start end Iterator
```

```
impl Iterator for I32Range {
 type Item = i32;
 fn next(&mut self) -> Option<i32> {
 if self.start >= self.end {
 return None;
 }
 let result = Some(self.start);
 self.start += 1;
 result
 }
}
```

Cet itérateur produit des éléments, c'est donc le type. Si l'itération est terminée, renvoie ; sinon, il produit la valeur suivante et met à jour son état pour se préparer au prochain appel. `i32 Item next None`

Bien sûr, une boucle permet de convertir son opérande en itérateur. Mais la bibliothèque standard fournit une implémentation générale de pour chaque type qui implémente , donc est prêt à l'emploi: `for IntoIterator::into_iter IntoIterator Iterator I32Range`

```

let mut pi = 0.0;
let mut numerator = 1.0;

for k in (I32Range { start: 0, end: 14 }) {
 pi += numerator / (2*k + 1) as f64;
 numerator /= -3.0;
}
pi *= f64::sqrt(12.0);

// IEEE 754 specifies this result exactly.
assert_eq!(pi as f32, std::f32::consts::PI);

```

Mais est un cas particulier, en ce sens que l'itérable et l'itérateur sont du même type. De nombreux cas ne sont pas si simples. Par exemple, voici le type d'arbre binaire du [chapitre 10](#) : I32Range

```

enum BinaryTree<T> {
 Empty,
 NonEmpty(Box<TreeNode<T>>)
}

struct TreeNode<T> {
 element: T,
 left: BinaryTree<T>,
 right: BinaryTree<T>
}

```

La façon classique de marcher dans un arbre binaire est de se récurser, en utilisant la pile d'appels de fonction pour garder une trace de votre place dans l'arbre et des nœuds à visiter. Mais lors de la mise en œuvre pour , chaque appel à doit produire exactement une valeur et un retour. Pour garder une trace des nœuds d'arborescence qu'il n'a pas encore produits, l'itérateur doit maintenir sa propre pile. Voici un type d'itérateur possible pour : Iterator BinaryTree<T> next BinaryTree

```

use self::BinaryTree::*;

// The state of an in-order traversal of a `BinaryTree`.
struct TreeIter<'a, T> {
 // A stack of references to tree nodes. Since we use `Vec`'s
 // `push` and `pop` methods, the top of the stack is the end of the
 // vector.
 //
 // The node the iterator will visit next is at the top of the stack
}

```

```

 // with those ancestors still unvisited below it. If the stack is empty
 // the iteration is over.
 unvisited: Vec<&'a TreeNode<T>>
}

```

Lorsque nous créons un nouveau `TreeIter`, son état initial devrait être sur le point de produire le nœud foliaire le plus à gauche de l'arbre. Selon les règles de la pile, elle devrait donc avoir cette feuille sur le dessus, suivie de ses ancêtres non visités: les nœuds le long du bord gauche de l'arbre. Nous pouvons initialiser en parcourant le bord gauche de l'arbre de la racine à la feuille et en poussant chaque nœud que nous rencontrons, nous allons donc définir une méthode pour le

faire: `TreeIter unvisited unvisited TreeIter`

```

impl<'a, T: 'a> TreeIter<'a, T> {
 fn push_left_edge(&mut self, mut tree: &'a BinaryTree<T>) {
 while let NonEmpty(ref node) = *tree {
 self.unvisited.push(node);
 tree = &node.left;
 }
 }
}

```

L'écriture permet à la boucle de changer le nœud pointant vers le long du bord gauche, mais comme il s'agit d'une référence partagée, elle ne peut pas muter les nœuds eux-mêmes. `mut tree` `tree tree`

Avec cette méthode d'assistance en place, nous pouvons donner une méthode qui renvoie un itérateur sur l'arbre: `BinaryTree iter`

```

impl<T> BinaryTree<T> {
 fn iter(&self) -> TreeIter<T> {
 let mut iter = TreeIter { unvisited: Vec::new() };
 iter.push_left_edge(self);
 iter
 }
}

```

La méthode construit un `TreeIter` avec une pile vide, puis appelle pour l'initialiser. Le nœud le plus à gauche se retrouve en haut, comme l'exigent les règles de la pile. `iter TreeIter unvisited push_left_edge unvisited`

Suivant les pratiques de la bibliothèque standard, on peut ensuite implémenter sur une référence partagée à un arbre avec un appel à

```
:IntoIterator BinaryTree::iter

impl<'a, T: 'a> IntoIterator for &'a BinaryTree<T> {
 type Item = &'a T;
 type IntoIter = TreeIter<'a, T>;
 fn into_iter(self) -> Self::IntoIter {
 self.iter()
 }
}
```

La définition établit comme type d'itérateur pour un fichier

```
.IntoIter TreeIter &BinaryTree
```

Enfin, dans la mise en œuvre, nous pouvons réellement marcher dans l'arbre. Comme la méthode de , la méthode de l'itérateur est guidée par les règles de la pile : Iterator BinaryTree iter next

```
impl<'a, T> Iterator for TreeIter<'a, T> {
 type Item = &'a T;
 fn next(&mut self) -> Option<&'a T> {
 // Find the node this iteration must produce,
 // or finish the iteration. (Use the `?` operator
 // to return immediately if it's `None`.)
 let node = self.unvisited.pop()?;
 // After `node`, the next thing we produce must be the leftmost
 // child in `node`'s right subtree, so push the path from here
 // down. Our helper method turns out to be just what we need.
 self.push_left_edge(&node.right);

 // Produce a reference to this node's value.
 Some(&node.element)
 }
}
```

Si la pile est vide, l'itération est terminée. Sinon, est une référence au nœud à visiter maintenant; cet appel renverra une référence à son champ. Mais d'abord, nous devons faire progresser l'état de l'itérateur vers le nœud suivant. Si ce nœud a un sous-arbre droit, le nœud suivant à visiter est le nœud le plus à gauche du sous-arbre, et nous pouvons l'utiliser pour le pousser, ainsi que ses ancêtres non visités, sur la pile. Mais si ce nœud n'a pas de sous-arbre droit, n'a aucun effet, ce qui est exactement ce que nous voulons: nous pouvons compter sur le nouveau

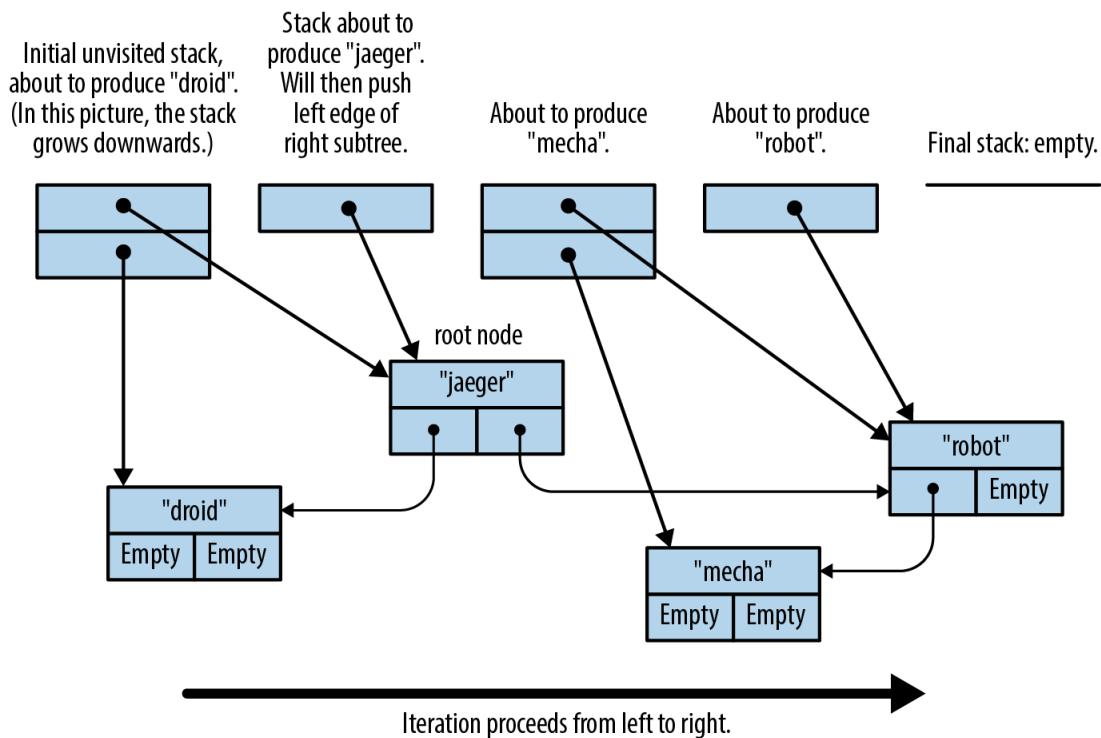
sommet de la pile pour être le premier ancêtre non visité, le cas échéant. node element push\_left\_edge push\_left\_edge node

Avec et les implémentations en place, nous pouvons enfin utiliser une boucle pour itérer sur un par référence. En utilisant la méthode on de [« Remplissage d'un arbre binaire »](#): IntoIterator Iterator for BinaryTree add BinaryTree

```
// Build a small tree.
let mut tree = BinaryTree::Empty;
tree.add("jaeger");
tree.add("robot");
tree.add("droid");
tree.add("mecha");

// Iterate over it.
let mut v = Vec::new();
for kind in &tree {
 v.push(*kind);
}
assert_eq!(v, ["droid", "jaeger", "mecha", "robot"]);
```

[La figure 15-1](#) montre comment la pile se comporte lorsque nous itérons dans un arbre d'échantillonnage. À chaque étape, le prochain nœud à visiter se trouve en haut de la pile, avec tous ses ancêtres non visités en dessous. unvisited



Graphique 15-1. Itération sur un arbre binaire

Tous les adaptateurs d'itérateurs habituels et les consommateurs sont prêts à être utilisés sur nos arbres:

```
assert_eq!(tree.iter()
 .map(|name| format!("mega-{}", name))
 .collect::<Vec<_>>(),
 vec!["mega-droid", "mega-jaeger",
 "mega-mecha", "mega-robot"]);
```

Les itérateurs sont l'incarnation de la philosophie de Rust de fournir des abstractions puissantes et sans coût qui améliorent l'expressivité et la lisibilité du code. Les itérateurs ne remplacent pas entièrement les boucles, mais ils fournissent une primitive capable avec une évaluation paisseuse intégrée et d'excellentes performances.

- 1 En fait, puisqu'est un itérable se comportant comme une suite de zéro ou un élément, est équivalent à , en supposant renvoie un  
. Option iterator.filter\_map  
(closure) iterator.flat\_map(closure) closure Option<T>

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

# Chapitre 16. Collections

*Nous nous comportons tous comme le démon de Maxwell. Les organismes s'organisent. Dans l'expérience quotidienne réside la raison pour laquelle les physiciens sobres à travers deux siècles ont maintenu ce fantasme de dessin animé vivant. Nous trions le courrier, construisons des châteaux de sable, résolvons des puzzles, séparons le bon grain de l'ivraie, réarrangeons les pièces d'échecs, collectionnons des timbres, alphabétisons les livres, créons une symétrie, composons des sonnets et des sonates, et mettons de l'ordre dans nos pièces, et tout cela nous ne nécessite pas une grande énergie, tant que nous pouvons appliquer l'intelligence.*

—James Gleick, *L'information : une histoire, une théorie, un déluge*

La bibliothèque standard Rust contient plusieurs *collections*, des types génériques pour stocker des données en mémoire. Nous avons déjà utilisé des collections, telles que `Vec` et `HashMap`, tout au long de ce livre. Dans ce chapitre, nous couvrirons en détail les méthodes de ces deux types, ainsi que les autres demi-douzaines de collections standard. Avant de commencer, abordons quelques différences systématiques entre les collections de Rust et celles d'autres langues. `Vec` `HashMap`

Tout d'abord, les déménagements et les emprunts sont partout. Rust utilise des mouvements pour éviter la copie en profondeur des valeurs. C'est pourquoi la méthode prend son argument par valeur, pas par référence. La valeur est déplacée dans le vecteur. Les diagrammes [du chapitre 4](#) montrent comment cela fonctionne dans la pratique : pousser un Rust vers `a` est rapide, car Rust n'a pas besoin de copier les données de caractère de la chaîne, et la propriété de la chaîne est toujours claire. `Vec<T>::push(item)` `String` `Vec<String>`

Deuxièmement, Rust n'a pas d'erreurs d'invalidation, le genre de bogue de pointeur pendant où une collection est redimensionnée ou modifiée d'une autre manière, alors que le programme contient un pointeur vers les données qu'elle contient. Les erreurs d'invalidation sont une autre source de comportement indéfini en C++, et elles provoquent occasionnellement même dans les langages sécurisés pour la mémoire. Le vérificateur d'emprunt de Rust les exclut au moment de la compilation. `ConcurrentModificationException`

Enfin, Rust n'a pas , nous verrons donc s dans des endroits où d'autres langues utiliseraient . null Option null

En dehors de ces différences, les collections de Rust sont à peu près ce à quoi vous vous attendez. Si vous êtes un programmeur expérimenté pressé, vous pouvez parcourir ici, mais ne manquez pas [« Entrées »](#).

## Aperçu

[Le tableau 16-1](#) montre les huit collections standard de Rust. Tous sont des types génériques.

Tableau 16-1. Résumé des collections standard

| <b>Collection</b>                                                                   | <b>Description</b>                                              | <b>Type de collection similaire dans...</b> |                                              |                                |
|-------------------------------------------------------------------------------------|-----------------------------------------------------------------|---------------------------------------------|----------------------------------------------|--------------------------------|
|                                                                                     |                                                                 | <b>C++</b>                                  | <b>Java</b>                                  | <b>Python</b>                  |
| <code>Vec&lt;T&gt;</code>                                                           | Baie extensible                                                 | <code>vector</code><br><code>r</code>       | <code>ArrayList</code><br><code>List</code>  | <code>list</code>              |
| <code>VecDeque&lt;T&gt;</code>                                                      | File d'attente à double extrémité (tampon annulaire extensible) |                                             | <code>deque</code><br><code>Deque</code>     | <code>collections.deque</code> |
| <code>LinkedList&lt;T&gt;</code>                                                    | Liste doublement liée                                           |                                             | <code>list</code><br><code>LinkedList</code> | <code>—</code>                 |
| <code>BinaryHeap&lt;T&gt;</code><br>where <code>T:</code><br><code>Ord</code>       | Nombre maximal de tas                                           | <code>priorPriorityQueue</code>             | <code>PriorityQueue</code>                   | <code>heapq</code>             |
| <code>HashMap&lt;K, V&gt;</code><br>where <code>K:</code><br><code>Eq + Hash</code> | Table de hachage clé-valeur                                     | <code>unordered_map</code>                  | <code>HashMap</code>                         | <code>dict</code>              |
| <code>BTreesMap&lt;K, V&gt;</code><br>where <code>K:</code><br><code>Ord</code>     | Table clé-valeur triée                                          | <code>map</code>                            | <code>TreeMap</code><br><code>Map</code>     | <code>—</code>                 |
| <code>HashSet&lt;T&gt;</code><br>where <code>T:</code><br><code>Eq + Hash</code>    | Ensemble non ordonné basé sur le hachage                        | <code>unordered_set</code>                  | <code>HashSet</code><br><code>Set</code>     | <code>set</code>               |

| Collection                     | Description   | Type de collection similaire dans... |         |        |
|--------------------------------|---------------|--------------------------------------|---------|--------|
|                                |               | C++                                  | Java    | Python |
| BTreeSet<T><br>where T:<br>Ord | Ensemble trié | set                                  | TreeSet | —      |

`Vec<T>`, et sont les types de collection les plus généralement utiles. Les autres ont des utilisations de niche. Ce chapitre traite de chaque type de collection à tour de rôle : `HashMap<K, V>` `HashSet<T>`

#### *Vec<T>*

Tableau de valeurs de type . Environ la moitié de ce chapitre est consacrée à ses nombreuses méthodes utiles. `T Vec`

#### *VecDeque<T>*

Comme , mais mieux pour une utilisation en tant que file d'attente premier entré, premier sorti. Il prend en charge l'ajout et la suppression efficaces de valeurs au début de la liste ainsi qu'à l'arrière. Cela se fait au prix d'un léger ralentissement de toutes les autres opérations. `Vec<T>`

#### *BinaryHeap<T>*

Une file d'attente prioritaire. Les valeurs de a sont organisées de manière à ce qu'il soit toujours efficace de rechercher et de supprimer la valeur maximale. `BinaryHeap`

#### *HashMap<K, V>*

Table de paires clé-valeur. La recherche d'une valeur par sa clé est rapide. Les entrées sont stockées dans un ordre arbitraire.

#### *BTreeMap<K, V>*

Comme , mais il conserve les entrées triées par clé. A stocke ses entrées dans l'ordre de comparaison. À moins que vous n'ayez besoin que les entrées restent triées, a est plus rapide. `HashMap<K, V>` `BTreeMap<String, int>` `String HashMap`

#### *HashSet<T>*

Ensemble de valeurs de type . L'ajout et la suppression de valeurs sont rapides, et il est rapide de demander si une valeur donnée est dans l'ensemble ou non. `T HashSet`

#### *BTreeSet<T>*

Comme , mais il conserve les éléments triés par valeur. Encore une fois, à moins que vous n'ayez besoin que les données soient triées, a est plus rapide. HashSet<T> HashSet

Parce qu'il est rarement utilisé (et qu'il existe de meilleures alternatives, à la fois en termes de performances et d'interface, pour la plupart des cas d'utilisation), nous ne le décrivons pas ici. LinkedList

## Vec<T>

Nous supposerons une certaine familiarité avec , puisque nous l'avons utilisé tout au long du livre. Pour une introduction, voir « [Vecteurs](#) ». Ici, nous allons enfin décrire en profondeur ses méthodes et son fonctionnement interne. Vec

Le moyen le plus simple de créer un vecteur est d'utiliser la macro : vec !

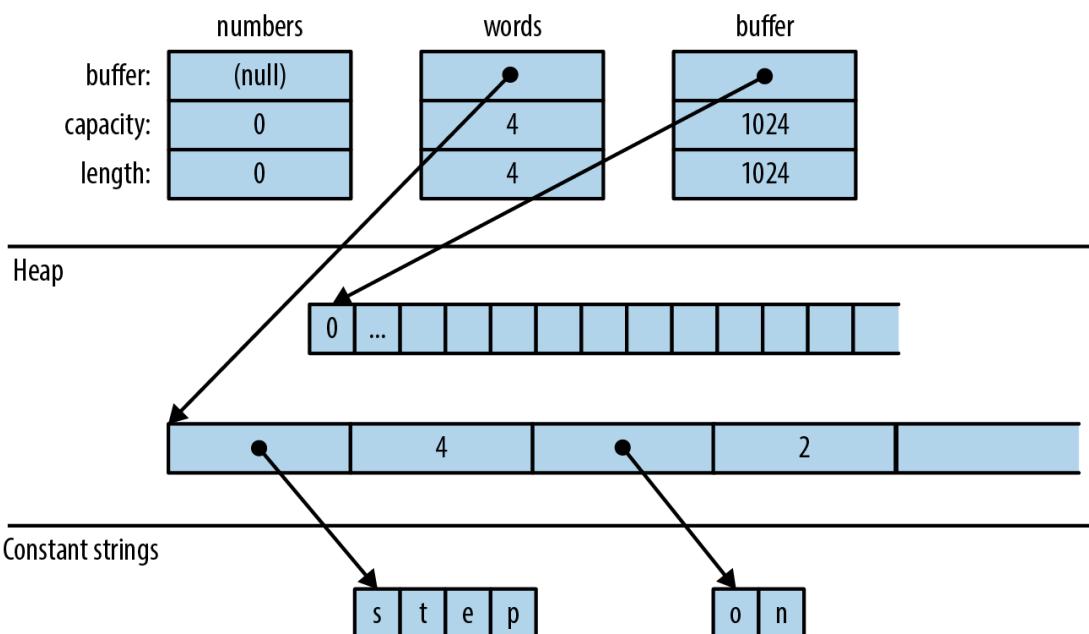
```
// Create an empty vector
let mut numbers: Vec<i32> = vec![];

// Create a vector with given contents
let words = vec!["step", "on", "no", "pets"];
let mut buffer = vec![0u8; 1024]; // 1024 zeroed-out bytes
```

Comme décrit au [chapitre 4](#), un vecteur comporte trois champs : la longueur, la capacité et un pointeur vers une allocation de tas où les éléments sont stockés. [La figure 16-1](#) montre comment les vecteurs précédents apparaîtraient en mémoire. Le vecteur vide, , a initialement une capacité de 0. Aucune mémoire de tas n'est allouée tant que le premier élément n'est pas ajouté. numbers

Comme toutes les collections, implémente , de sorte que vous pouvez créer un vecteur à partir de n'importe quel itérateur en utilisant la méthode de l'itérateur, comme décrit dans [« Building Collections: collect and FromIterator »](#): Vec std::iter::FromIterator .collect()

```
// Convert another collection to a vector.
let my_vec = my_set.into_iter().collect::<Vec<String>>();
```



Graphique 16-1. Disposition vectorielle en mémoire : chaque élément de mots est une valeur composée d'un pointeur et d'une longueur &str

## Accès aux éléments

L'obtention d'éléments d'un tableau, d'une tranche ou d'un vecteur par index est simple :

```
// Get a reference to an element
let first_line = &lines[0];

// Get a copy of an element
let fifth_number = numbers[4]; // requires Copy
let second_line = lines[1].clone(); // requires Clone

// Get a reference to a slice
let my_ref = &buffer[4..12];

// Get a copy of a slice
let my_copy = buffer[4..12].to_vec(); // requires Clone
```

Toutes ces formes paniquent si un index est hors limites.

Rust est pointilleux sur les types numériques, et il ne fait aucune exception pour les vecteurs. Les longueurs de vecteurs et les indices sont de type . Essayer d'utiliser un , , ou comme index vectoriel est une erreur. Vous pouvez utiliser un cast pour convertir si nécessaire; voir [«Type Casts »](#).  
`usize u32 u64 isize n as usize`

Plusieurs méthodes permettent d'accéder facilement à des éléments particuliers d'un vecteur ou d'une tranche (notez que toutes les méthodes de tranche sont également disponibles sur les tableaux et les vecteurs) :

```
slice.first()
```

Renvoie une référence au premier élément de , le cas échéant. slice

Le type de retour est , donc la valeur de retour est si est vide et si elle n'est pas vide : Option<&T> None slice Some(&slice[0])

```
if let Some(item) = v.first() {
 println!("We got one! {}", item);
}
```

```
slice.last()
```

Similaire mais renvoie une référence au dernier élément.

```
slice.get(index)
```

Renvoie la référence à , s'il existe. Si a moins d'éléments, cela renvoie : Some slice[index] slice index+1 None

```
let slice = [0, 1, 2, 3];
assert_eq!(slice.get(2), Some(&2));
assert_eq!(slice.get(4), None);
```

```
slice.first_mut(),, slice.last_mut() slice.get_mut(index)
```

Variations de ce qui précède qui empruntent des références: mut

```
let mut slice = [0, 1, 2, 3];
{
 let last = slice.last_mut().unwrap(); // type of last: &mut i
 assert_eq!(*last, 3);
 *last = 100;
}
assert_eq!(slice, [0, 1, 2, 100]);
```

Étant donné que le renvoi d'une valeur by signifierait son déplacement, les méthodes qui accèdent aux éléments en place renvoient généralement ces éléments par référence. T

Une exception est la méthode, qui fait des copies: .to\_vec()

```
slice.to_vec()
```

Clone une tranche entière, renvoyant un nouveau vecteur :

```
let v = [1, 2, 3, 4, 5, 6, 7, 8, 9];
assert_eq!(v.to_vec(),
```

```

 vec![1, 2, 3, 4, 5, 6, 7, 8, 9]);
assert_eq!(v[0..6].to_vec(),
 vec![1, 2, 3, 4, 5, 6]);

```

Cette méthode n'est disponible que si les éléments sont clonables, c'est-à-dire `.where T: Clone`

## Itération

Les vecteurs, les tableaux et les tranches sont itérables, soit par valeur, soit par référence, selon le modèle décrit dans [« Implémentations IntoIterator »](#):

- L'itération sur un ou un tableau produit des éléments de type `.T`. Les éléments sont déplacés hors du vecteur ou du tableau un par un, le consommant. `Vec<T> [T; N] T`
- L'itération sur une valeur de type `, ,` ou, c'est-à-dire une référence à un tableau, une tranche ou un vecteur, produit des éléments de type `, des références aux éléments individuels, qui ne sont pas déplacés.` `&[T; N] &[T] &Vec<T> &T`
- Itération sur une valeur de type `, ,` ou produit des éléments de type `.&mut [T; N] &mut [T] &mut Vec<T> &mut T`

Les tableaux, les tranches et les vecteurs ont également des méthodes (décrivées dans [« iter and iter\\_mut Methods »](#)) pour créer des itérateurs qui produisent des références à leurs éléments. `.iter()` `.iter_mut()`

Nous couvrirons quelques façons plus sophistiquées d'itérer sur une tranche dans [« Splitting »](#).

## Vecteurs de croissance et de rétrécissement

La *longueur* d'un tableau, d'une tranche ou d'un vecteur est le nombre d'éléments qu'il contient :

`slice.len()`

Renvoie la longueur d'un `.slice.usize`

`slice.is_empty()`

Est vrai si ne contient aucun élément (c'est-à-dire) `.slice.slice.len() == 0`

Les autres méthodes de cette section concernent la croissance et le rétrécissement des vecteurs. Ils ne sont pas présents sur les tableaux et les

tranches, qui ne peuvent pas être redimensionnées une fois créés.

Tous les éléments d'un vecteur sont stockés dans un morceau de mémoire contigu et alloué au tas. La *capacité* d'un vecteur est le nombre maximal d'éléments qui pourraient tenir dans ce morceau. gère normalement la capacité pour vous, allouant automatiquement un tampon plus grand et y déplaçant les éléments lorsque plus d'espace est nécessaire. Il existe également quelques méthodes pour gérer explicitement la capacité : `vec`

#### `Vec::with_capacity(n)`

Crée un nouveau vecteur vide avec capacité . n

#### `vec.capacity()`

Renvoie la capacité de , sous la forme d'un fichier . Il est toujours vrai que

. vec.usize() >= vec.len()

#### `vec.reserve(n)`

S'assure que le vecteur a au moins une capacité de réserve suffisante pour plus d'éléments: c'est-à-dire est au moins . S'il y a déjà assez de place, cela ne fait rien. Si ce n'est pas le cas, cela alloue un tampon plus grand et y déplace le contenu du vecteur. n vec.capacity() vec.len() + n

#### `vec.reserve_exact(n)`

Comme , mais dit de ne pas allouer de capacité supplémentaire pour la croissance future, au-delà de . Après, c'est exactement

. vec.reserve(n) vec.n vec.capacity() vec.len() + n

#### `vec.shrink_to_fit()`

Essaie de libérer la mémoire supplémentaire si elle est supérieure à

. vec.capacity() vec.len()

`Vec<T>` a de nombreuses méthodes qui ajoutent ou suppriment des éléments, modifiant la longueur du vecteur. Chacun d'entre eux prend son argument par référence. `self` mut

Ces deux méthodes ajoutent ou suppriment une seule valeur à la fin d'un vecteur :

#### `vec.push(value)`

Ajoute le donné à la fin de . value vec

#### `vec.pop()`

Supprime et renvoie le dernier élément. Le type de retour est . Cela renvoie si l'élément éclaté est et si le vecteur était déjà vide. Option<T> Some(x) x None

Notez que prend son argument par valeur, pas par référence. De même, renvoie la valeur poppée, pas une référence. Il en va de même pour la plupart des méthodes restantes de cette section. Ils déplacent les valeurs à l'intérieur et à l'extérieur des vecteurs. `.push()` `.pop()`

Ces deux méthodes ajoutent ou suppriment une valeur n'importe où dans un vecteur :

`vec.insert(index, value)`

Insère le donné à , en faisant glisser toutes les valeurs existantes à un endroit vers la droite pour faire de la place. `value` `vec[index]` `vec[index..]`

Panique si `index > vec.len()`

`vec.remove(index)`

Supprime et renvoie , en faisant glisser toutes les valeurs existantes à un endroit vers la gauche pour combler l'écart. `vec[index]` `vec[index+1..]`

Panique si , puisque dans ce cas il n'y a pas d'élément à supprimer. `index >= vec.len()` `vec[index]`

Plus le vecteur est long, plus cette opération est lente. Si vous vous retrouvez à faire beaucoup, envisagez d'utiliser un (expliqué dans « [VecDeque<T>](#) ») au lieu d'un `.vec.remove(0)` `VecDeque` `Vec`

Les deux et sont plus lents plus les éléments doivent être déplacés. `.insert()` `.remove()`

Quatre méthodes modifient la longueur d'un vecteur en une valeur spécifique :

`vec.resize(new_len, value)`

Définit la longueur de . Si cela augmente la longueur, des copies de sont ajoutées pour remplir le nouvel espace. Le type d'élément doit implémenter le trait. `vec` `new_len` `vec` `value` `Clone`

`vec.resize_with(new_len, closure)`

Tout comme , mais appelle la fermeture pour construire chaque nouvel élément. Il peut être utilisé avec des vecteurs d'éléments qui ne sont pas `.vec.resize` `Clone`

`vec.truncate(new_len)`

Réduit la longueur de à , en laissant tomber tous les éléments qui se trouvaient dans la plage . vec new\_len vec[new\_len..]

Si est déjà inférieur ou égal à , rien ne se passe. vec.len() new\_len  
vec.clear()

Supprime tous les éléments de . C'est la même chose que  
. vec vec.truncate(0)

Quatre méthodes ajoutent ou suppriment plusieurs valeurs à la fois :

`vec.extend(iterable)`

Ajoute tous les éléments de la valeur donnée à la fin de , dans l'ordre. C'est comme une version à valeurs multiples de . L'argument peut être n'importe quoi qui implémente

. iterable vec .push() iterable IntoIterator<Item=T>

Cette méthode est si utile qu'il existe un trait standard, le trait, que toutes les collections standard implémentent. Malheureusement, cela provoque un regroupement avec d'autres méthodes de trait dans une grande pile au bas du code HTML généré, il est donc difficile de trouver quand vous en avez besoin. Vous n'avez qu'à vous rappeler qu'il est là! Voir [« The Extend Trait »](#) pour plus d'informations. Extend rustdoc .extend()

`vec.split_off(index)`

Comme , sauf qu'il renvoie un contenant les valeurs supprimées de la fin de . C'est comme une version à valeurs multiples de  
. vec.truncate(index) Vec<T> vec .pop()

`vec.append(&mut vec2)`

Cela déplace tous les éléments de dans , où est un autre vecteur de type . Après, est vide. vec2 vec vec2 Vec<T> vec2

C'est comme si cela existait encore par la suite, avec sa capacité non affectée. vec.extend(vec2) vec2

`vec.drain(range)`

Cela supprime le de et renvoie un itérateur sur les éléments supprimés, où est une valeur de plage, comme ou . range vec[range] vec range .. 0..4

Il existe également quelques méthodes bizarres pour supprimer sélectivement certains éléments d'un vecteur :

`vec.retain(test)`

Supprime tous les éléments qui ne réussissent pas le test donné.  
L'argument est une fonction ou une fermeture qui implémente .  
Pour chaque élément de , cet appel , et s'il retourne , l'élément est  
supprimé du vecteur et supprimé. `test FnMut(&T) ->`  
`bool vec test(&element) false`

En dehors de la performance, c'est comme écrire :

```
vec = vec.into_iter().filter(test).collect();

vec.dedup()
```

Laisse tomber les éléments répétés. C'est comme l'utilitaire shell Unix. Il recherche les endroits où les éléments adjacents sont égaux et supprime les valeurs égales supplémentaires afin qu'il n'en reste qu'une seule : `uniq vec`

```
let mut byte_vec = b"Mississippi".to_vec();
byte_vec.dedup();
assert_eq!(&byte_vec, b"Missipi");
```

Notez qu'il y a encore deux caractères dans la sortie. Cette méthode supprime uniquement les doublons *adjacents*. Pour éliminer tous les doublons, vous avez trois options : trier le vecteur avant d'appeler, déplacer les données dans un ensemble, ou (pour conserver les éléments dans leur ordre d'origine) utiliser cette astuce

```
: 's' .dedup() .retain()
```

```
let mut byte_vec = b"Mississippi".to_vec();

let mut seen = HashSet::new();
byte_vec.retain(|r| seen.insert(*r));

assert_eq!(&byte_vec, b"Misp");
```

Cela fonctionne car il retourne lorsque l'ensemble contient déjà l'élément que nous insérons. `.insert() false`

```
vec.dedup_by(same)
```

La même chose que , mais il utilise la fonction ou la fermeture , au lieu de l'opérateur, pour vérifier si deux éléments doivent être considérés comme égaux. `vec.dedup() same(&mut elem1, &mut elem2) ==`

```
vec.dedup_by_key(key)
```

La même chose que , mais il traite deux éléments comme égaux si

```
.vec.dedup() key(&mut elem1) == key(&mut elem2)
```

Par exemple, si est un , vous pouvez écrire : errors Vec<Box<dyn Error>>

```
// Remove errors with redundant messages.
errors.dedup_by_key(|err| err.to_string());
```

De toutes les méthodes abordées dans cette section, seules les valeurs clo-  
nent. Les autres travaillent en déplaçant les valeurs d'un endroit à un  
autre. .resize()

## Joignant

Deux méthodes fonctionnent sur *des tableaux de tableaux, par lesquels*  
nous entendons tout tableau, tranche ou vecteur dont les éléments sont  
eux-mêmes des tableaux, des tranches ou des vecteurs :

```
slices.concat()
```

Renvoie un nouveau vecteur créé en concaténant toutes les tranches  
:

```
assert_eq!([[1, 2], [3, 4], [5, 6]].concat(),
 vec![1, 2, 3, 4, 5, 6]);
```

```
slices.join(&separator)
```

La même chose, sauf qu'une copie de la valeur est insérée entre les  
tranches : separator

```
assert_eq!([[1, 2], [3, 4], [5, 6]].join(&0),
 vec![1, 2, 0, 3, 4, 0, 5, 6]);
```

## Fractionnement

Il est facile d'obtenir de nombreuses non-références dans un tableau, une  
tranche ou un vecteur à la fois : mut

```
let v = vec![0, 1, 2, 3];
let a = &v[i];
let b = &v[j];

let mid = v.len() / 2;
```

```
let front_half = &v[..mid];
let back_half = &v[mid..];
```

Obtenir plusieurs références n'est pas si facile: mut

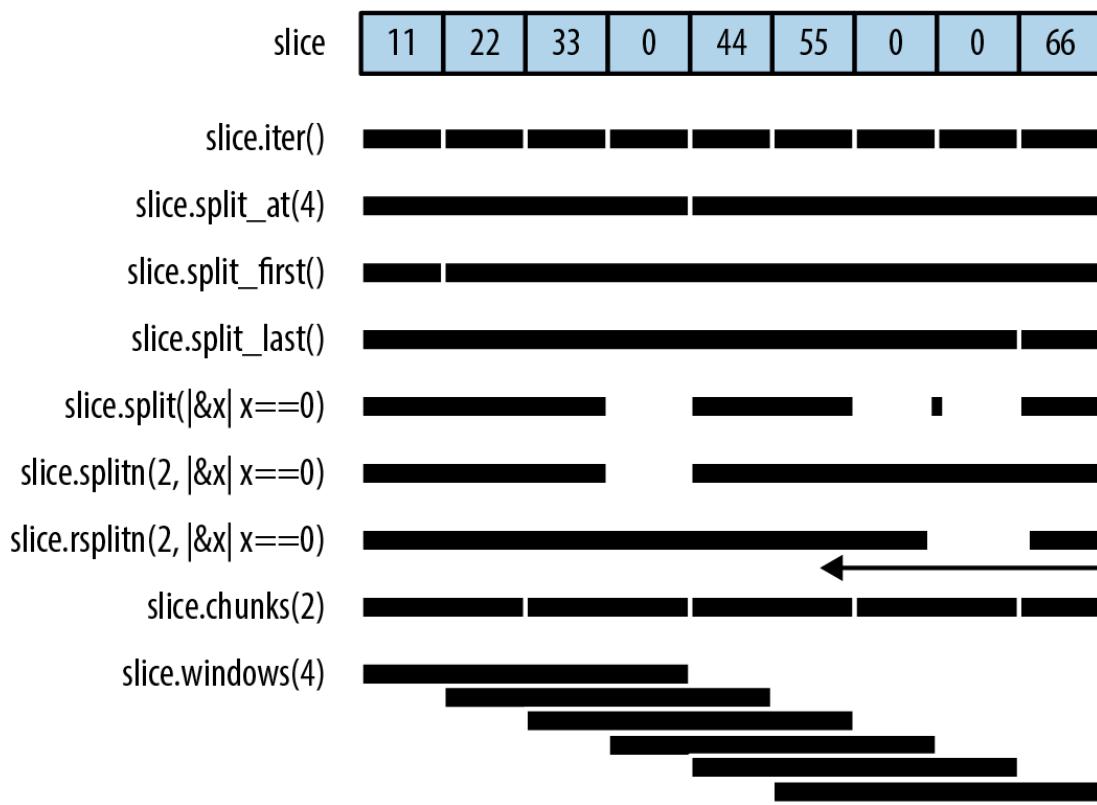
```
let mut v = vec![0, 1, 2, 3];
let a = &mut v[i];
let b = &mut v[j]; // error: cannot borrow `v` as mutable
 // more than once at a time

*a = 6; // references `a` and `b` get used here,
*b = 7; // so their lifetimes must overlap
```

Rust l'interdit parce que si , alors et serait deux références au même en-tier, en violation des règles de sécurité de Rust. (Voir [« Partage versus mutation ».](#)) i == j a b mut

Rust a plusieurs méthodes qui peuvent emprunter des références à deux ou plusieurs parties d'un tableau, d'une tranche ou d'un vecteur à la fois. Contrairement au code précédent, ces méthodes sont sûres, car de par leur conception, elles divisent toujours les données en régions *sans chevauchement*. Beaucoup de ces méthodes sont également pratiques pour travailler avec des non-tranches, il existe donc des versions non-. mut mut mut mut

[La figure 16-2](#) illustre ces méthodes.



Graphique 16-2. Méthodes de fractionnement illustrées (note: le petit rectangle dans la sortie de `rsplitn` est une tranche vide causée par les deux séparateurs adjacents, et produit sa sortie dans l'ordre de bout en début, contrairement aux autres) `slice.split()` `rsplitn`

Aucune de ces méthodes ne modifie directement un tableau, une tranche ou un vecteur ; ils renvoient simplement de nouvelles références à des parties des données à l'intérieur:

`slice.iter()`, `slice.iter_mut()`

Produire une référence à chaque élément de `.slice`. Nous les avons couverts dans « [Itération](#) ».

`slice.split_at(index)`, `slice.split_at_mut(index)`

Casser une tranche en deux, en renvoyant une paire. `slice.split_at(index)` est équivalent à `(&slice[..index], &slice[index..])`. Ces méthodes paniquent si `index` est hors limites.

`slice.split_first()`, `slice.split_first_mut()`

Renvoie également une paire : une référence au premier élément (`slice[0]`) et une référence de tranche à tout le reste (`slice[1..]`).

Le type de retour de `slice.split_first()` est ; le résultat est si `slice` est

`vide`.

`slice.split_last()`, `slice.split_last_mut()`

Ceux-ci sont analogues mais séparent le dernier élément plutôt que le premier.

Le type de retour de `slice.split_last()` est `Option<(&T, &[T])>`

```
slice.split(is_sep), slice.split_mut(is_sep)
```

Divisez en une ou plusieurs sous-couches, en utilisant la fonction ou la fermeture pour déterminer où diviser. Ils renvoient un itérateur sur les sous-couches. `slice is_sep`

Lorsque vous consommez l’itérateur, il appelle chaque élément de la tranche. Si est , l’élément est un séparateur. Les séparateurs ne sont inclus dans aucune sous-sous-section de

```
sortie. is_sep(&element) is_sep(&element) true
```

La sortie contient toujours au moins une sous-section, plus une par séparateur. Les sous-couches vides sont incluses chaque fois que des séparateurs apparaissent adjacents les uns aux autres ou aux extrémités de .`slice`

```
slice.split_inclusive(is_sep),
slice.split_inclusive_mut(is_sep)
```

Ceux-ci fonctionnent comme et , mais incluent le séparateur à la fin de la sous-section précédente plutôt que de l’exclure. `split split_mut`

```
slice.rsplit(is_sep), slice.rsplit_mut(is_sep)
```

Tout comme et , mais commencez à la fin de la tranche. `slice slice_mut`  
`slice.splitn(n, is_sep), slice.splitn_mut(n, is_sep)`

Les mêmes mais ils produisent au plus des sous-couches. Une fois les premières tranches trouvées, n’est pas appelé à nouveau. La dernière sous-sous-section contient tous les éléments restants. `n n-1 is_sep`

```
slice.rsplitn(n, is_sep), slice.rsplitn_mut(n, is_sep)
```

Tout comme et sauf que la tranche est scannée dans l’ordre inverse. C’est-à-dire que ces méthodes se divisent sur les *derniers* séparateurs de la tranche, plutôt que sur le premier, et les sous-couches sont produites à partir de la fin. `.splitn() .splitn_mut() n-1`

```
slice.chunks(n), slice.chunks_mut(n)
```

Renvoyer un itérateur sur des sous-sous-couches de longueur sans chevauchement . S’il ne se divise pas exactement, le dernier morceau contiendra moins d’éléments. `n n slice.len() n`

```
slice.rchunks(n), slice.rchunks_mut(n)
```

Tout comme et , mais commencez à la fin de la tranche. `slice.chunks slice.chunks_mut`

```
slice.chunks_exact(n), slice.chunks_exact_mut(n)
```

Renvoyer un itérateur sur des sous-sous-couches de longueur sans chevauchement . Si ne divise pas , le dernier morceau (avec moins d’éléments)

```
est disponible dans la méthode du
résultat. n n slice.len() n remainder()
slice.rchunks_exact(n), slice.rchunks_exact_mut(n)
```

Tout comme et , mais commencez à la fin de la  
tranche. slice.chunks\_exact slice.chunks\_exact\_mut

Il existe une autre méthode pour itérer sur les sous-couches :

```
slice.windows(n)
```

Renvoie un itérateur qui se comporte comme une « fenêtre coulissante » sur les données dans . Il produit des sous-couches qui couvrent des éléments consécutifs de . La première valeur produite est , la seconde est , et ainsi de

```
suite.slice n slice &slice[0..n] &slice[1..n+1]
```

Si est supérieure à la longueur de , aucune tranche n'est produite. Si est 0, la méthode panique. n slice n

Par exemple, si , alors nous pouvons produire toutes les périodes de sept jours en appelant . days.len() ==  
31 days days.windows(7)

Une fenêtre coulissante de taille 2 est pratique pour explorer comment une série de données change d'un point de données à l'autre :

```
let changes = daily_high_temperatures
 .windows(2) // get adjacent days' tem
 .map(|w| w[1] - w[0]) // how much did it change
 .collect::<Vec<_>>();
```

Étant donné que les sous-sous-couches se chevauchent, il n'existe aucune variante de cette méthode qui renvoie des références. mut

## Échange

Il existe des méthodes pratiques pour échanger le contenu des tranches :

```
slice.swap(i, j)
```

Échange les deux éléments et . slice[i] slice[j]

```
slice_a.swap(&mut slice_b)
```

Échange l'intégralité du contenu de et . et doit avoir la même longueur. slice\_a slice\_b slice\_a slice\_b

Les vecteurs ont une méthode connexe pour supprimer efficacement n'importe quel élément:

```
vec.swap_remove(i)
```

Supprime et renvoie . C'est comme si, sauf qu'au lieu de faire glisser le reste des éléments du vecteur pour combler l'écart, il déplace simplement le dernier élément de l'espace dans l'espace. C'est utile lorsque vous ne vous souciez pas de l'ordre des éléments laissés dans le vecteur.

```
vec[i] vec.remove(i) vec
```

## Remplissage

Il existe deux méthodes pratiques pour remplacer le contenu des tranches modifiables :

```
slice.fill(value)
```

Remplit la tranche avec des clones de . value

```
slice.fill_with(function)
```

Remplit la tranche avec des valeurs créées en appelant la fonction donnée. Ceci est particulièrement utile pour les types qui implémentent , mais ne sont pas , comme ou quand n'est pas

```
.Default Clone Option<T> Vec<T> T Clone
```

## Tri et recherche

Les tranches offrent trois méthodes de tri :

```
slice.sort()
```

Trie les éléments dans un ordre croissant. Cette méthode est présente uniquement lorsque le type d'élément implémente . Ord

```
slice.sort_by(cmp)
```

Trie les éléments de l'utilisation d'une fonction ou d'une fermeture pour spécifier l'ordre de tri. doit mettre en œuvre

```
.slice cmp cmp Fn(&T, &T) -> std::cmp::Ordering
```

La mise en œuvre manuelle est pénible, sauf si vous déléguez à une méthode : cmp .cmp( )

```
students.sort_by(|a, b| a.last_name.cmp(&b.last_name));
```

Pour trier par un champ, en utilisant un deuxième champ comme bris d'égalité, comparez les tuples :

```
students.sort_by(|a, b| {
 let a_key = (&a.last_name, &a.first_name);
 let b_key = (&b.last_name, &b.first_name);
 a_key.cmp(&b_key)
});
```

```
slice.sort_by_key(key)
```

Trie les éléments de dans l'ordre croissant par une clé de tri, donnée par la fonction ou la fermeture . Le type de doit implémenter où

```
.slice key key Fn(&T) -> K K: Ord
```

Ceci est utile lorsqu'il contient un ou plusieurs champs ordonnés, de sorte qu'il peut être trié de plusieurs façons: T

```
// Sort by grade point average, lowest first.
students.sort_by_key(|s| s.grade_point_average());
```

Notez que ces valeurs de clé de tri ne sont pas mises en cache pendant le tri, de sorte que la fonction peut être appelée plus de  $n$  fois. key

Pour des raisons techniques, ne peut pas renvoyer de références empruntées à l'élément. Cela ne fonctionnera pas : key(element)

```
students.sort_by_key(|s| &s.last_name); // error: can't infer life
```

Rust ne peut pas comprendre les durées de vie. Mais dans ces cas, il est assez facile de se rabattre sur ..sort\_by()

Les trois méthodes effectuent un tri stable.

Pour trier dans l'ordre inverse, vous pouvez utiliser avec une fermeture qui permute les deux arguments. Prendre des arguments plutôt que de produire efficacement l'ordre inverse. Ou, vous pouvez simplement appeler la méthode après le tri: sort\_by cmp |b, a| |a, b| .reverse()

```
slice.reverse()
```

Inverse une tranche en place.

Une fois qu'une tranche est triée, elle peut être recherchée efficacement :

```
slice.binary_search(&value),,
slice.binary_search_by(&value,
cmp) slice.binary_search_by_key(&value, key)
```

Toutes les recherches dans le fichier trié donné . Notez que c'est passé par référence. `value slice value`

Le type de retour de ces méthodes est . Ils renvoient si égaux dans l'ordre de tri spécifié. S'il n'y a pas un tel index, ils renvoient de telle sorte que l'insertion de à préserverait l'ordre. `Result<usize, usize> Ok(index) slice[index] value Err(insertion_point) value insertion_point`

Bien sûr, une recherche binaire ne fonctionne que si la tranche est en fait triée dans l'ordre spécifié. Sinon, les résultats sont arbitraires : garbage in, garbage out.

Depuis et ont des valeurs NaN, ils ne s'implémentent pas et ne peuvent pas être utilisés directement comme clés avec les méthodes de tri et de recherche binaire. Pour obtenir des méthodes similaires qui fonctionnent sur des données à virgule flottante, utilisez la caisse. `f32 f64 Ord ord_subset`

Il existe une méthode pour rechercher un vecteur qui n'est pas trié :

`slice.contains(&value)`

Renvoie si un élément de est égal à . Cela vérifie simplement chaque élément de la tranche jusqu'à ce qu'une correspondance soit trouvée. Encore une fois, est passé par référence. `true slice value value`

Pour trouver l'emplacement d'une valeur dans une tranche, comme en JavaScript, utilisez un itérateur : `array.indexOf(value)`

`slice.iter().position(|x| *x == value)`

Cela renvoie un fichier . `Option<usize>`

## Comparaison des tranches

Si un type prend en charge les opérateurs et (le trait décrit dans [« Comparaisons d'équivalence »](#)), les tableaux, les tranches et les vecteurs les prennent également en charge. Deux tranches sont égales si elles ont la même longueur et que leurs éléments correspondants sont égaux. Il en va de même pour les tableaux et les vecteurs. `T == != PartialEq [T; N] [T] Vec<T>`

Si prend en charge les opérateurs , , et (le trait, décrit dans [« Comparaisons ordonnées »](#)), alors les tableaux, les tranches et les vecteurs de faire

aussi. Les comparaisons de tranches sont lexicographiques. `T < <= > >= PartialOrd T`

Deux méthodes pratiques effectuent des comparaisons de tranches courantes :

`slice.starts_with(other)`

Renvoie si commence par une séquence de valeurs égales aux éléments de la tranche : `true slice other`

```
assert_eq!([1, 2, 3, 4].starts_with(&[1, 2]), true);
assert_eq!([1, 2, 3, 4].starts_with(&[2, 3]), false);
```

`slice.ends_with(other)`

Similaire mais vérifie la fin de : `slice`

```
assert_eq!([1, 2, 3, 4].ends_with(&[3, 4]), true);
```

## Éléments aléatoires

Les nombres aléatoires ne sont pas intégrés à la bibliothèque standard Rust. La caisse, qui les fournit, offre ces deux méthodes pour obtenir une sortie aléatoire à partir d'un tableau, d'une tranche ou d'un vecteur : `:rand`

`slice.choose(&mut rng)`

Renvoie une référence à un élément aléatoire d'une tranche. Comme et , cela renvoie un qui n'est que si la tranche est vide. `slice.first() slice.last() Option<&T> None`

`slice.shuffle(&mut rng)`

Réorganise de manière aléatoire les éléments d'une tranche en place. La tranche doit être passée par référence. `mut`

Ce sont des méthodes du trait, vous avez donc besoin d'un , un générateur de nombres aléatoires, afin de les appeler. Heureusement, il est facile d'en obtenir un en appelant . Pour mélanger le vecteur , nous pouvons écrire: `rand::Rng Rng rand::thread_rng() my_vec`

```
use rand::seq::SliceRandom;
use rand::thread_rng;

my_vec.shuffle(&mut thread_rng());
```

# Rust exclut les erreurs d'invalidation

La plupart des langages de programmation traditionnels ont des collections et des itérateurs, et ils ont tous une variante de cette règle : ne modifiez pas une collection pendant que vous itérez dessus. Par exemple, l'équivalent Python d'un vecteur est une liste :

```
my_list = [1, 3, 5, 7, 9]
```

Supposons que nous essayions de supprimer toutes les valeurs supérieures à 4 de : my\_list

```
for index, val in enumerate(my_list):
 if val > 4:
 del my_list[index] # bug: modifying list while iterating

print(my_list)
```

(La fonction est l'équivalent de Python de la méthode de Rust, décrite dans [« énumérer »](#).). `enumerate` .`enumerate()`

Ce programme, étonnamment, imprime . Mais sept est plus grand que quatre. Comment cela s'est-il glissé? Il s'agit d'une erreur d'invalidation : le programme modifie les données tout en itérant dessus, *invalidant* l'itérateur. En Java, le résultat serait une exception ; en C++, il s'agit d'un comportement non défini. En Python, bien que le comportement soit bien défini, il n'est pas intuitif : l'itérateur saute un élément. n'est jamais . [1, 3, 7] val 7

Essayons de reproduire ce bug dans Rust:

```
fn main() {
 let mut my_vec = vec![1, 3, 5, 7, 9];

 for (index, &val) in my_vec.iter().enumerate() {
 if val > 4 {
 my_vec.remove(index); // error: can't borrow `my_vec` as m
 }
 }
 println!("{:?}", my_vec);
}
```

Naturellement, Rust rejette ce programme au moment de la compilation. Lorsque nous appelons , il emprunte une référence partagée (non-) au

vecteur. La référence vit aussi longtemps que l'itérateur, jusqu'à la fin de la boucle. Nous ne pouvons pas modifier le vecteur en appelant tant qu'une non-référence

```
exists. my_vec.iter() mut for my_vec.remove(index) mut
```

Avoir une erreur qui vous est signalée est bien, mais bien sûr, vous devez toujours trouver un moyen d'obtenir le comportement souhaité! La solution la plus simple ici est d'écrire:

```
my_vec.retain(|&val| val <= 4);
```

Ou, vous pouvez faire ce que vous feriez en Python ou dans n'importe quel autre langage : créer un nouveau vecteur à l'aide d'un fichier  
.filter

## VecDeque<T>

`vec` prend en charge l'ajout et la suppression efficaces d'éléments uniquement à la fin. Lorsqu'un programme a besoin d'un endroit pour stocker des valeurs qui « font la queue », cela peut être lent. `vec`

Rust's est un *deque* (prononcé « deck »), une file d'attente à double extrémité. Il prend en charge les opérations d'ajout et de suppression efficaces à l'avant et à l'arrière: `std::collections::VecDeque<T>`

```
deque.push_front(value)
```

Ajoute une valeur à l'avant de la file d'attente.

```
deque.push_back(value)
```

Ajoute une valeur à la fin. (Cette méthode est utilisée beaucoup plus que , car la convention habituelle pour les files d'attente est que les valeurs sont ajoutées à l'arrière et supprimées à l'avant, comme les personnes qui attendent dans une file.) `.push_front()`

```
deque.pop_front()
```

Supprime et renvoie la valeur frontale de la file d'attente, en renvoyant un fichier si la file d'attente est vide, par exemple `.Option<T> None vec.pop()`

```
deque.pop_back()
```

Supprime et renvoie la valeur à l'arrière, en renvoyant à nouveau un fichier `.Option<T>`

```
deque.front(), deque.back()
```

Travailler comme et . Ils renvoient une référence à l'élément avant ou arrière de la file d'attente. La valeur renvoyée est un si la file d'attente est vide. `vec.first()` `vec.last()` Option<&T> None

`deque.front_mut()`, `deque.back_mut()`

Travailler comme et , revenir

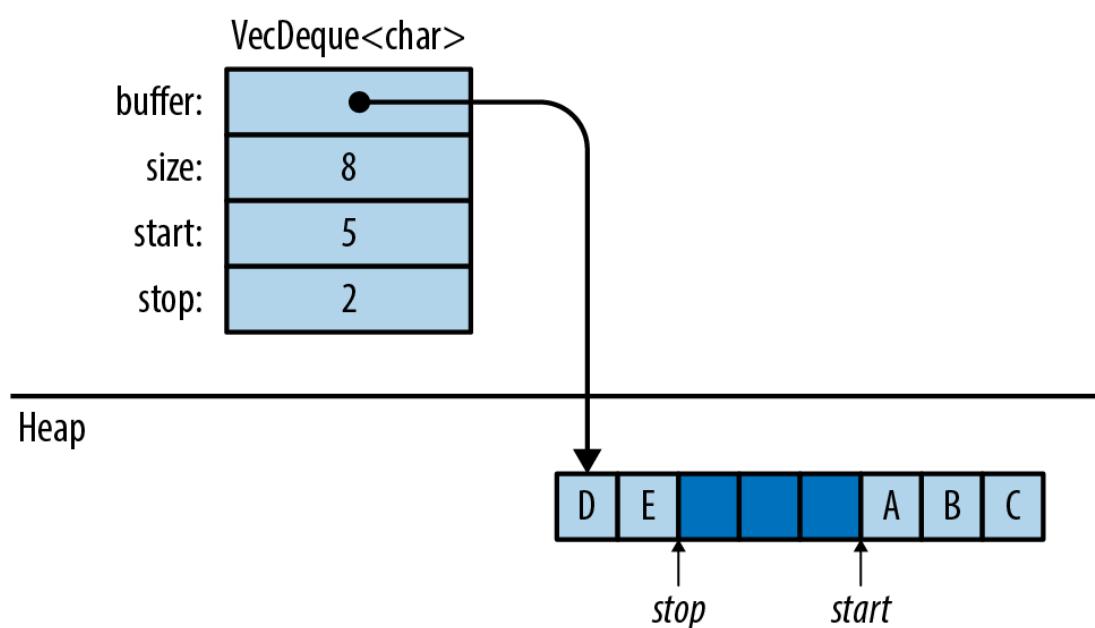
. `vec.first_mut()` `vec.last_mut()` Option<&mut T>

L'implémentation de est un tampon en anneau, comme le montre [la figure 16-3](#). VecDeque

Comme un , il a une allocation de tas unique où les éléments sont stockés. Contrairement à , les données ne commencent pas toujours au début de cette région, et elles peuvent « envelopper » la fin, comme indiqué. Les éléments de ce deque, dans l'ordre, sont . a des champs privés, étiquetés et dans la figure, qu'il utilise pour se souvenir de l'endroit où dans la mémoire tampon les données commencent et se terminent. `Vec` `Vec` [ 'A' , 'B' , 'C' , 'D' , 'E' ] `VecDeque` `start` `stop`

Ajouter une valeur à la file d'attente, à chaque extrémité, signifie réclamer l'un des emplacements inutilisés, illustré comme les blocs les plus sombres, envelopper ou allouer une plus grande partie de la mémoire si nécessaire.

`VecDeque` gère l'emballage, de sorte que vous n'avez pas à y penser. [La figure 16-3](#) est une vue des coulisses de la façon dont Rust fait vite. `.pop_front()`



Graphique 16-3. Comment a est stocké en mémoire `VecDeque`

Souvent, lorsque vous avez besoin d'un deque, et sont les deux seules méthodes dont vous aurez besoin. Les fonctions associées au type et

```
VecDeque::with_capacity(n), pour créer des files d'attente, sont comme leurs homologues dans Vec. De nombreuses méthodes sont également implémentées pour VecDeque : et, .insert(index, value), .remove(index), .extend(iterable), etc. .push_back() .pop_front() VecDeque::new() Vec.len() .is_empty()
```

Les deque, comme les vecteurs, peuvent être itérés par valeur, par référence partagée ou par référence. Ils ont les trois méthodes d'itération , et . Ils peuvent être indexés de la manière habituelle :

```
.mut .into_iter() .iter() .iter_mut() deque[index]
```

Parce que les deque ne stockent pas leurs éléments de manière contiguë dans la mémoire, ils ne peuvent pas hériter de toutes les méthodes de tranches. Mais si vous êtes prêt à payer le coût du déplacement du contenu, fournit une méthode qui résoudra ce problème: VecDeque

```
deque.make_contiguous()
```

Prend et réorganise le dans la mémoire contiguë, en renvoyant . &mut  
self VecDeque &mut [T]

Vec s et s sont étroitement liés, et la bibliothèque standard fournit deux implémentations de traits pour une conversion facile entre les deux : VecDeque

```
Vec::from(deque)
```

Vec<T> implémente , donc cela transforme un deque en vecteur. Cela coûte du temps O(n), car cela peut nécessiter de réorganiser les éléments. From<VecDeque<T>>

```
VecDeque::from(vec)
```

VecDeque<T> implémente , donc cela transforme un vecteur en deque. C'est aussi O(n), mais c'est généralement rapide, même si le vecteur est grand, car l'allocation de tas du vecteur peut simplement être déplacée vers la nouvelle deque. From<Vec<T>>

Cette méthode facilite la création d'une deque avec des éléments spécifiés, même s'il n'existe pas de macro standard : vec\_deque! []

```
use std::collections::VecDeque;
```

```
let v = VecDeque::from(vec![1, 2, 3, 4]);
```

# BinaryHeap<T>

A est une collection dont les éléments sont maintenus vaguement organisés de sorte que la plus grande valeur bouillonne toujours jusqu'à l'avant de la file d'attente. Voici les trois méthodes les plus couramment utilisées

: BinaryHeap BinaryHeap

`heap.push(value)`

Ajoute une valeur au tas.

`heap.pop()`

Supprime et renvoie la plus grande valeur du tas. Il renvoie un c'est-à-dire si le tas était vide. Option<T> None

`heap.peek()`

Renvoie une référence à la valeur la plus élevée du tas. Le type de retour est . Option<&T>

`heap.peek_mut()`

Renvoie un , qui agit comme une référence modifiable à la plus grande valeur du tas et fournit la fonction associée au type pour extraire cette valeur du tas. En utilisant cette méthode, nous pouvons choisir de pop ou non pop du tas en fonction de la valeur maximale: PeekMut<T> pop()

```
use std::collections::binary_heap::PeekMut;

if let Some(top) = heap.peek_mut() {
 if *top > 10 {
 PeekMut::pop(top);
 }
}
```

BinaryHeap prend également en charge un sous-ensemble des méthodes sur , y compris , , , .clear() et .Vec BinaryHeap::new() .len() .is\_empty() .capacity() .append(&mut heap2)

Par exemple, supposons que nous remplissions a avec un tas de nombres

: BinaryHeap

```
use std::collections::BinaryHeap;
```

```
let mut heap = BinaryHeap::from(vec![2, 3, 8, 6, 9, 5, 4]);
```

La valeur se trouve en haut du tas : 9

```
assert_eq!(heap.peek(), Some(&9));
assert_eq!(heap.pop(), Some(9));
```

La suppression de la valeur réorganise également légèrement les autres éléments de sorte qu'ils soient maintenant à l'avant, et ainsi de suite : 9 8

```
assert_eq!(heap.pop(), Some(8));
assert_eq!(heap.pop(), Some(6));
assert_eq!(heap.pop(), Some(5));
...
```

Bien sûr, ne se limite pas aux chiffres. Il peut contenir n'importe quel type de valeur qui implémente le trait intégré. `BinaryHeap Ord`

Cela rend utile en tant que file d'attente de travail. Vous pouvez définir une structure de tâche qui s'implémente sur la base de la priorité afin que les tâches de priorité plus élevée soient plus importantes que les tâches de priorité inférieure. Ensuite, créez un pour contenir toutes les tâches en attente. Sa méthode renverra toujours l'élément le plus important, la tâche sur laquelle votre programme devrait travailler ensuite. `BinaryHeap Ord Greater BinaryHeap .pop()`

Remarque: est itérable, et il a une méthode, mais les itérateurs produisent les éléments du tas dans un ordre arbitraire, pas du plus grand au plus petit. Pour consommer les valeurs d'un ordre de priorité, utilisez une boucle : `BinaryHeap .iter() BinaryHeap while`

```
while let Some(task) = heap.pop() {
 handle(task);
}
```

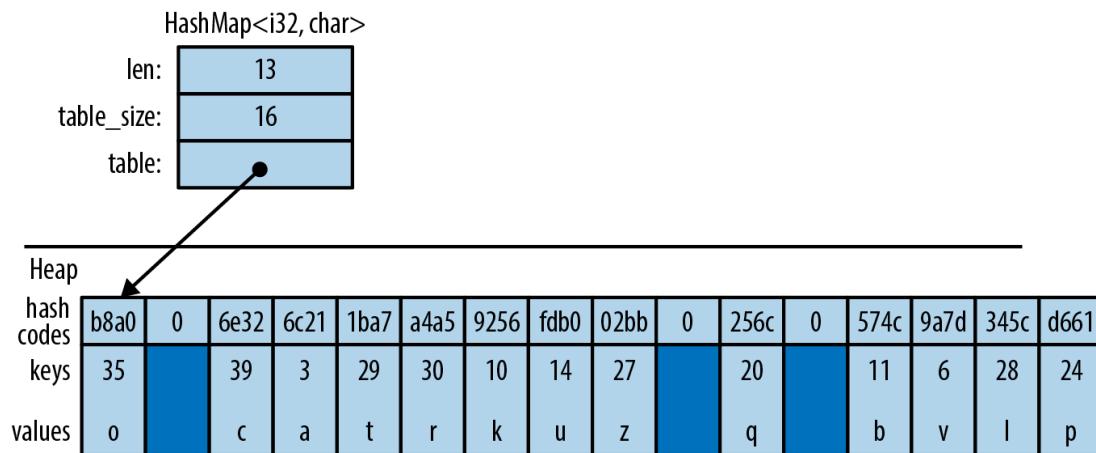
## HashMap<K, V> et BTreeMap<K, V>

Une *carte* est une collection de paires clé-valeur (*appelées entrées*). Il n'y a pas deux entrées qui ont la même clé, et les entrées sont maintenues organisées de sorte que si vous avez une clé, vous pouvez rechercher efficacement la valeur correspondante dans une carte. En bref, une carte est une table de choix.

Rust propose deux types de cartes : et . Les deux partagent bon nombre des mêmes méthodes; la différence réside dans la façon dont les deux entrées de conservation sont organisées pour une recherche rapide. `HashMap<K, V>` `BTreeMap<K, V>`

A stocke les clés et les valeurs dans une table de hachage, il nécessite donc un type de clé qui implémente et , les traits standard pour le hachage et l'égalité. **HashMap<K, Hash<Eq>**

La figure 16-4 montre comment a est organisé en mémoire. Les régions plus sombres ne sont pas utilisées. Toutes les clés, valeurs et codes de hachage mis en cache sont stockés dans une seule table allouée au tas. L'ajout d'entrées finit par forcer l'allocation d'une table plus grande et à y déplacer toutes les données. `HashMap`



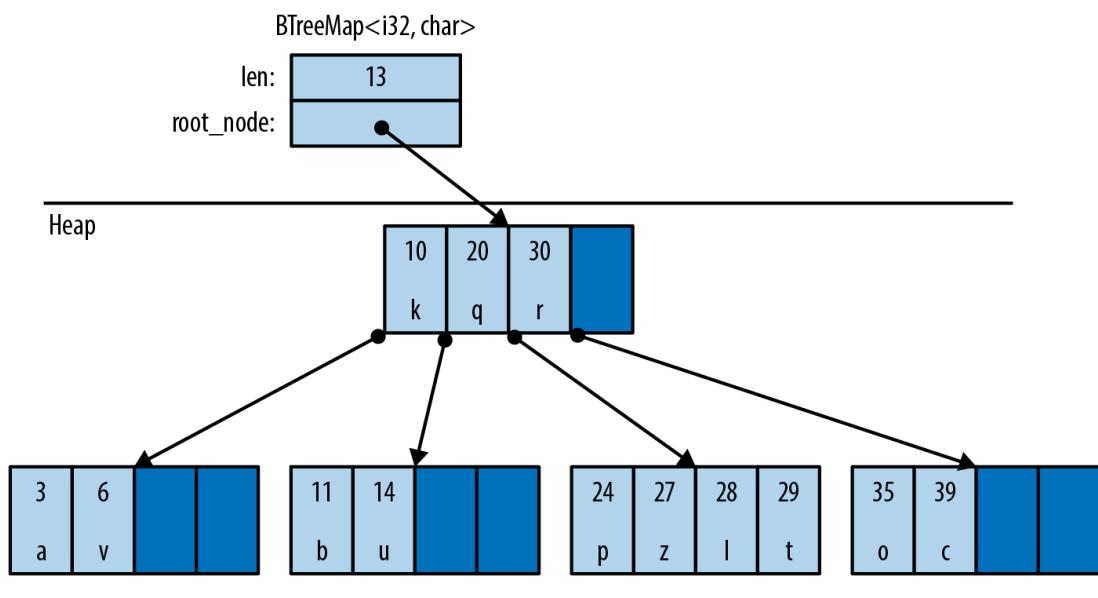
Graphique 16-4. A en mémoire HashMap

A stocke les entrées dans l'ordre par clé, dans une arborescence, de sorte qu'il nécessite un type de clé qui implémente . [La figure 16-5](#) montre un fichier . Encore une fois, les régions les plus sombres sont des capacités inutilisées. BTreesMap K Ord BTreesMap

A stocke ses entrées dans des *nœuds*. La plupart des nœuds d'un ne contiennent que des paires clé-valeur. Les nœuds non feuilles, comme le nœud racine illustré dans cette figure, ont également de la place pour les pointeurs vers les nœuds enfants. Pointeur entre et pointe vers un nœud enfant contenant des clés entre et . L'ajout d'entrées nécessite souvent de faire glisser certaines des entrées existantes d'un nœud vers la droite, pour les garder triées, et implique parfois l'allocation de nouveaux nœuds.

```
BTreeMap BTreeMap (20, 'q') (30, 'r') 20 30
```

Cette image est un peu simplifiée pour tenir sur la page. Par exemple, les nœuds réels ont de la place pour 11 entrées, pas 4. BTreeMap



Graphique 16-5. A en mémoire BTreeMap

La bibliothèque standard Rust utilise des arbres B plutôt que des arbres binaires équilibrés, car les arbres B sont plus rapides sur le matériel moderne. Un arbre binaire peut utiliser moins de comparaisons par recherche qu'un arbre B, mais la recherche d'un arbre B a une meilleure *localité*, c'est-à-dire que les accès à la mémoire sont regroupés plutôt que dispersés sur l'ensemble du tas. Cela rend les erreurs de cache cpu plus rares. C'est une augmentation significative de la vitesse.

Il existe plusieurs façons de créer une carte :

`HashMap::new()`, `BTreeMap::new()`

Créez de nouvelles cartes vides.

`iter.collect()`

Peut être utilisé pour créer et remplir une nouvelle paire clé-valeur ou à partir de paires clé-valeur. doit être un fichier

`.HashMap BTreeMap iter Iterator<Item=(K, V)>`

`HashMap::with_capacity(n)`

Crée une nouvelle carte de hachage vide avec de la place pour au moins *n* entrées. s, comme les vecteurs, stockent leurs données dans une seule allocation de tas, de sorte qu'ils ont une capacité et les méthodes associées , et . ne le font pas. `HashMap hash_map.capacity()` `hash_map.reserve(additional)` `hash_map.shrink_to_fit()` `BTreeMap`

HashMap s et s ont les mêmes méthodes de base pour travailler avec des clés et des valeurs : BTreeMap

`map.len()`

Renvoie le nombre d'entrées.

`map.is_empty()`

Renvoie s'il n'y a pas d'entrées. `true` `map`

`map.contains_key(&key)`

Renvoie si la carte comporte une entrée pour le fichier . `true` `key`

`map.get(&key)`

Recherche une entrée avec le fichier . Si une entrée correspondante est trouvée, celle-ci renvoie , où est une référence à la valeur correspondante. Sinon, cela renvoie `.map key Some(r) r None`

`map.get_mut(&key)`

Similaire, mais il renvoie une référence à la valeur. `mut`

En général, les cartes vous permettent d'avoir accès aux valeurs qui y sont stockées, mais pas aux clés. Les valeurs sont à vous de modifier comme vous le souhaitez. Les clés appartiennent à la carte elle-même; il doit s'assurer qu'ils ne changent pas, car les entrées sont organisées par leurs clés. Modifier une clé en place serait un bug. `mut`

`map.insert(key, value)`

Insère l'entrée et renvoie l'ancienne valeur, le cas échéant. Le type de retour est .

S'il y a déjà une entrée pour dans la carte, la nouvelle entrée écrase l'ancienne.

`(key, value) map Option<V> key value`

`map.extend(iterable)`

Itère sur les éléments de et insère chacune de ces paires clé-valeur dans . `(K, V) iterable map`

`map.append(&mut map2)`

Déplace toutes les entrées de dans . Après, est vide. `map2 map map2`

`map.remove(&key)`

Recherche et supprime toute entrée avec la valeur donnée de , renvoyant la valeur supprimée, le cas échéant. Le type de retour est . `key map Option<V>`

`map.remove_entry(&key)`

Recherche et supprime toute entrée avec la valeur donnée de , renvoyant la clé et la valeur supprimées, le cas échéant. Le type de retour est . `key map Option<(K, V)>`

`map.retain(test)`

Supprime tous les éléments qui ne réussissent pas le test donné.

L'argument est une fonction ou une fermeture qui implémente .

Pour chaque élément de , cet appel , et s'il retourne , l'élément est supprimé de la carte et supprimé. `test FnMut(&K, &mut V) -> bool map test(&key, &mut value) false`

En dehors de la performance, c'est comme écrire :

```
map = map.into_iter().filter(test).collect();

map.clear()
```

Supprime toutes les entrées.

Une carte peut également être interrogée entre crochets : . C'est-à-dire que les cartes implémentent le trait intégré. Cependant, cela panique s'il n'y a pas déjà une entrée pour le donné , comme un accès au tableau hors limites, alors utilisez cette syntaxe uniquement si l'entrée que vous recherchez est sûre d'être remplie. `map[&key] Index key`

L'argument de , , et n'a pas besoin d'avoir le type exact . Ces méthodes sont génériques sur les types qui peuvent être empruntés à . Il est correct d'appeler un , même si ce n'est pas exactement un , car implémente . Pour plus de détails, voir [« Emprunter et empruntermut »](#)

```
».key .contains_key() .get() .get_mut() .remove() &K K fish_
map.contains_key("conger") HashMap<String,
Fish> "conger" String String Borrow<&str>
```

Étant donné que a conserve ses entrées triées par clé, il prend en charge une opération supplémentaire : `BTreeMap<K, V>`

```
btree_map.split_off(&key)
```

Se divise en deux. Entrées dont les touches sont inférieures à celles qui restent dans . Renvoie un nouveau contenant les autres entrées. `btree_map key btree_map BTreeMap<K, V>`

## Entrées

Les deux et ont un type correspondant. Le but des entrées est d'éliminer les recherches de carte redondantes. Par exemple, voici du code pour obtenir ou créer un enregistrement étudiant : `HashMap BTreeMap Entry`

```
// Do we already have a record for this student?
if !student_map.contains_key(name) {
 // No: create one.
 student_map.insert(name.to_string(), Student::new());
}
// Now a record definitely exists.
let record = student_map.get_mut(name).unwrap();
...
```

Cela fonctionne bien, mais il accède deux ou trois fois, en faisant la même recherche à chaque fois. `student_map`

L'idée avec les entrées est que nous ne faisons la recherche qu'une seule fois, produisant une valeur qui est ensuite utilisée pour toutes les opérations ultérieures. Ce one-liner est équivalent à tout le code précédent, sauf qu'il ne fait la recherche qu'une seule fois : `Entry`

```
let record = student_map.entry(name.to_string()).or_insert_with(Student
```

La valeur renvoyée par agit comme une référence modifiable à un endroit de la carte qui est soit *occupé* par une paire clé-valeur, soit *vacant*, ce qui signifie qu'il n'y a pas encore d'entrée. Si elle est vacante, la méthode de l'entrée insère un nouveau fichier . La plupart des utilisations des entrées sont comme ceci: court et

```
doux. Entry student_map.entry(name.to_string()) .or_insert_w
ith() Student
```

Toutes les valeurs sont créées par la même méthode : `Entry`

```
map.entry(key)
```

Renvoie un pour le fichier . S'il n'y a pas de telle clé dans la carte, cela renvoie un fichier vacant . `Entry key Entry`

Cette méthode prend son argument par référence et renvoie un avec une durée de vie correspondante : `self mut Entry`

```
pub fn entry<'a>(&'a mut self, key: K) -> Entry<'a, K, V>
```

Le type a un paramètre de durée de vie car il s'agit en fait d'une sorte de référence empruntée à la carte. Tant qu'il existe, il a un accès exclusif à la carte. `Entry 'a mut Entry`

De retour dans « [Structs Containing References](#) », nous avons vu comment stocker des références dans un type et comment cela affecte les durées de vie. Maintenant, nous voyons à quoi cela ressemble du point de vue de l'utilisateur. C'est ce qui se passe avec . `Entry`

Malheureusement, il n'est pas possible de passer une référence de type à cette méthode si la carte comporte des clés. La méthode, dans ce cas, nécessite un réel . `&str String .entry() String`

Entry les valeurs fournissent trois méthodes pour traiter les entrées vacantes :

```
map.entry(key).or_insert(value)
```

S'assure qu'il contient une entrée avec le donné, en insérant une nouvelle entrée avec le donné si nécessaire. Il renvoie une référence à la valeur nouvelle ou existante. map key value mut

Supposons que nous devions compter les votes. Nous pouvons écrire

:

```
let mut vote_counts: HashMap<String, usize> = HashMap::new();
for name in ballots {
 let count = vote_counts.entry(name).or_insert(0);
 *count += 1;
}
```

.or\_insert() renvoie une référence, de sorte que le type de est .mut count &mut usize

```
map.entry(key).or_default()
```

Garantit qu'il contient une entrée avec la clé donnée, en insérant une nouvelle entrée avec la valeur renvoyée par si nécessaire. Cela ne fonctionne que pour les types qui implémentent . Par exemple, cette méthode renvoie une référence à la valeur nouvelle ou

existante. map Default::default() Default or\_insert mut

```
map.entry(key).or_insert_with(default_fn)
```

C'est la même chose, sauf que s'il doit créer une nouvelle entrée, il appelle pour produire la valeur par défaut. S'il y a déjà une entrée pour dans le , alors n'est pas

utilisé. default\_fn() key map default\_fn

Supposons que nous voulions savoir quels mots apparaissent dans quels fichiers. Nous pouvons écrire :

```
// This map contains, for each word, the set of files it appears in
let mut word_occurrence: HashMap<String, HashSet<String>> =
 HashMap::new();
for file in files {
 for word in read_words(file)? {
 let set = word_occurrence
 .entry(word)
 .or_insert_with(HashSet::new);
 set.insert(file.clone());
 }
}
```

```
 }
}
```

Entry fournit également un moyen pratique de modifier uniquement les champs existants.

```
map.entry(key).and_modify(closure)
```

Appelle si une entrée avec la clé existe, en passant une référence modifiable à la valeur. Il renvoie le , afin qu'il puisse être enchaîné avec d'autres méthodes. closure key Entry

Par exemple, nous pourrions l'utiliser pour compter le nombre d'occurrences de mots dans une chaîne :

```
// This map contains all the words in a given string,
// along with the number of times they occur.
let mut word_frequency: HashMap<&str, u32> = HashMap::new();
for c in text.split_whitespace() {
 word_frequency.entry(c)
 .and_modify(|count| *count += 1)
 .or_insert(1);
}
```

Le type est un enum, défini ainsi pour (et de même pour )  
:Entry HashMap BTreeMap

```
// (in std::collections::hash_map)
pub enum Entry<'a, K, V> {
 Occupied(OccupiedEntry<'a, K, V>),
 Vacant(VacantEntry<'a, K, V>)
}
```

Les types ont des méthodes pour insérer, supprimer et accéder aux entrées sans répéter la recherche initiale. Vous pouvez les trouver dans la documentation en ligne. Les méthodes supplémentaires peuvent parfois être utilisées pour éliminer une recherche redondante ou deux, mais et couvrir les cas

```
courants. OccupiedEntry VacantEntry .or_insert() .or_insert_with()
```

## Itération de carte

Il existe plusieurs façons d'itérer sur une carte :

- L'itération par valeur () produit des paires. Cela consomme la carte. `for (k, v) in map (K, V)`
- L'itération sur une référence partagée () produit des paires. `for (k, v) in &map (&K, &V)`
- L'itération sur une référence () produit des paires. (Encore une fois, il n'y a aucun moyen d'accéder aux clés stockées dans une carte, car les entrées sont organisées par leurs clés.) `mut for (k, v) in &mut map (&K, &mut V) mut`

Comme les vecteurs, les cartes ont des méthodes qui renvoient des itérateurs par référence, tout comme l'itération sur ou . De plus, `: .iter() .iter_mut() &map &mut map`

`map.keys()`

Renvoie un itérateur sur les seules clés, par référence.

`map.values()`

Renvoie un itérateur sur les valeurs, par référence.

`map.values_mut()`

Renvoie un itérateur sur les valeurs, par référence. `mut`

`map.into_iter(),, map.into_keys() map.into_values()`

Consommez la carte, en renvoyant un itérateur sur des tuples de clés et de valeurs, de clés ou de valeurs, respectivement. `(K, V)`

Tous les itérateurs visitent les entrées de la carte dans un ordre arbitraire. les itérateurs les visitent dans l'ordre par clé. `HashMap` `BTreeMap`

## HashSet<T> et BTreeSet<T>

*Les ensembles* sont des collections de valeurs organisées pour un test d'appartenance rapide :

```
let b1 = large_vector.contains(&"needle"); // slow, checks every element
let b2 = large_hash_set.contains(&"needle"); // fast, hash lookup
```

Un ensemble ne contient jamais plusieurs copies de la même valeur.

Les cartes et les ensembles ont des méthodes différentes, mais dans les coulisses, un ensemble est comme une carte avec seulement des clés, plutôt que des paires clé-valeur. En fait, les deux types d'ensembles de Rust, et , sont implémentés sous forme d'enveloppes minces autour de et `. HashSet<T> BTreeSet<T> HashMap<T, ()> BTreeMap<T, ()>`

`HashSet::new()`, `BTreeSet::new()`

Créez de nouveaux ensembles.

`iter.collect()`

Peut être utilisé pour créer un nouvel ensemble à partir de n'importe quel itérateur. Si produit des valeurs plus d'une fois, les doublons sont supprimés. `iter`

`HashSet::with_capacity(n)`

Crée un vide avec de l'espace pour au moins des valeurs. `HashSet n`

`HashSet<T>` et ont toutes les méthodes de base en commun: `BTreeSet<T>`

`set.len()`

Renvoie le nombre de valeurs dans . `set`

`set.is_empty()`

Renvoie si l'ensemble ne contient aucun élément. `true`

`set.contains(&value)`

Renvoie si l'ensemble contient le fichier . `true value`

`set.insert(value)`

Ajoute un à l'ensemble. Renvoie si une valeur a été ajoutée, si elle était déjà membre de l'ensemble. `value true false`

`set.remove(&value)`

Supprime un de l'ensemble. Renvoie si une valeur a été supprimée, si elle n'était pas déjà membre de l'ensemble. `value true false`

`set.retain(test)`

Supprime tous les éléments qui ne réussissent pas le test donné.

L'argument est une fonction ou une fermeture qui implémente .

Pour chaque élément de , cet appel , et s'il retourne , l'élément est

supprimé de l'ensemble et supprimé. `test FnMut(&T) ->`

`bool set test(&value) false`

En dehors de la performance, c'est comme écrire :

```
set = set.into_iter().filter(test).collect();
```

Comme pour les cartes, les méthodes qui recherchent une valeur par référence sont génériques sur les types qui peuvent être empruntés à . Pour plus de détails, voir « [Emprunter et empruntermut](#) ». T

# Définir l'itération

Il existe deux façons d'itérer sur les ensembles :

- L'itération par valeur (« ») produit les membres de l'ensemble (et consomme l'ensemble). `for v in set`
- L'itération par référence partagée (« ») produit des références partagées aux membres de l'ensemble. `for v in &set`

L'itération sur un ensemble par référence n'est pas prise en charge. Il n'y a aucun moyen d'obtenir une référence à une valeur stockée dans un ensemble. `mut mut`

`set.iterator()`

Renvoie un itérateur sur les membres de par référence. `set`

`HashSet` les itérateurs, comme les itérateurs, produisent leurs valeurs dans un ordre arbitraire. les itérateurs produisent des valeurs dans l'ordre, comme un vecteur trié. `HashMap` `BTreeSet`

## Lorsque des valeurs égales sont différentes

Les ensembles ont quelques méthodes étranges que vous devez utiliser uniquement si vous vous souciez des différences entre les valeurs « égales ».

De telles différences existent souvent. Deux valeurs identiques, par exemple, stockent leurs caractères à différents endroits en mémoire : `String`

```
let s1 = "hello".to_string();
let s2 = "hello".to_string();
println!("{:p}", &s1 as &str); // 0x7f8b32060008
println!("{:p}", &s2 as &str); // 0x7f8b32060010
```

Habituellement, nous ne nous en soucions pas.

Mais si jamais vous le faites, vous pouvez accéder aux valeurs réelles stockées dans un ensemble en utilisant les méthodes suivantes. Chacun renvoie un qui est s'il ne contient pas de valeur correspondante  
`:Option None` `set`

`set.get(&value)`

Renvoie une référence partagée au membre de qui est égal à , le cas échéant.

Renvoie un fichier . `set value Option<&T>`

```
set.take(&value)
```

Comme , mais il renvoie la valeur supprimée, le cas échéant. Renvoie un fichier  
.set.remove(&value) Option<T>

```
set.replace(value)
```

Comme , mais si contient déjà une valeur égale à , cela remplace et renvoie l'ancienne valeur. Renvoie un fichier  
.set.insert(value) set value Option<T>

## Opérations de l'ensemble

Jusqu'à présent, la plupart des méthodes d'ensemble que nous avons vues sont axées sur une seule valeur dans un seul ensemble. Les ensembles ont également des méthodes qui fonctionnent sur des ensembles entiers :

```
set1.intersection(&set2)
```

Renvoie un itérateur sur toutes les valeurs qui se trouvent dans les deux et .set1 set2

Par exemple, si nous voulons imprimer les noms de tous les étudiants qui suivent à la fois des cours de chirurgie cérébrale et de science des fusées, nous pourrions écrire:

```
for student in &brain_class {
 if rocket_class.contains(student) {
 println!("{}", student);
 }
}
```

Ou, plus court :

```
for student in brain_class.intersection(&rocket_class) {
 println!("{}", student);
}
```

Étonnamment, il y a un opérateur pour cela.

&set1 & &set2 renvoie un nouvel ensemble qui est l'intersection de et . Il s'agit de l'opérateur binaire binaire ET, appliqué à deux références. Cela trouve des valeurs qui sont dans les deux et : set1 set2 set1 set2

```
let overachievers = &brain_class & &rocket_class;
```

```
set1.union(&set2)
```

Renvoie un itérateur sur les valeurs qui se trouvent dans l'un ou l'autre ou , ou les deux. `set1 set2`

`&set1 | &set2` renvoie un nouvel ensemble contenant toutes ces valeurs. Il recherche les valeurs qui se trouvent dans *ou*. `set1 set2`

```
set1.difference(&set2)
```

Renvoie un itérateur sur les valeurs qui se trouvent dans mais pas dans. `set1 set2`

`&set1 - &set2` renvoie un nouvel ensemble contenant toutes ces valeurs.

```
set1.symmetric_difference(&set2)
```

Renvoie un itérateur sur les valeurs qui se trouvent dans l'un ou l'autre ou , mais pas les deux. `set1 set2`

`&set1 ^ &set2` renvoie un nouvel ensemble contenant toutes ces valeurs.

Et il existe trois méthodes pour tester les relations entre les ensembles :

```
set1.is_disjoint(set2)
```

True si et n'ont pas de valeurs en commun : l'intersection entre elles est vide. `set1 set2`

```
set1.is_subset(set2)
```

True if est un sous-ensemble de —c'est-à-dire que toutes les valeurs dans sont également dans . `set1 set2 set1 set2`

```
set1.is_superset(set2)
```

C'est l'inverse : c'est vrai si est un surensemble de . `set1 set2`

Les ensembles prennent également en charge les tests d'égalité avec ; deux ensembles sont égaux s'ils contiennent les mêmes valeurs. `== !=`

## Hachage

`std::hash::Hash` est le trait de bibliothèque standard pour les types hashables. les clés et les éléments doivent implémenter à la fois et `.HashMap HashSet Hash Eq`

La plupart des types intégrés qui implémentent implémentent également . Les types entiers, , et sont tous hashables; il en va de même pour les tu-

ples, les tableaux, les tranches et les vecteurs, tant que leurs éléments sont hachables. `Eq Hash char String`

L'un des principes de la bibliothèque standard est qu'une valeur doit avoir le même code de hachage, quel que soit l'endroit où vous la stockez ou la façon dont vous la pointez. Par conséquent, une référence a le même code de hachage que la valeur à laquelle elle fait référence et a le même code de hachage que la valeur encadrée. Un vecteur a le même code de hachage que la tranche contenant toutes ses données, . A a le même code de hachage que a avec les mêmes caractères. `Box vec &vec[...] String &str`

Les structs et les enums ne sont pas implémentés par défaut, mais une implémentation peut être dérivée : `Hash`

```
// The ID number for an object in the British Museum's collection.
#[derive(Clone, PartialEq, Eq, Hash)]
enum MuseumNumber {
 ...
}
```

Cela fonctionne tant que les champs du type sont tous hachissables.

Si vous implémentez à la main pour un type, vous devez également implémenter à la main. Par exemple, supposons que nous ayons un type qui représente des trésors historiques inestimables : `PartialEq Hash`

```
struct Artifact {
 id: MuseumNumber,
 name: String,
 cultures: Vec<Culture>,
 date: RoughTime,
 ...
}
```

Deux s sont considérés comme égaux s'ils ont le même ID : `Artifact`

```
impl PartialEq for Artifact {
 fn eq(&self, other: &Artifact) -> bool {
 self.id == other.id
 }
}

impl Eq for Artifact {}
```

Puisque nous comparons les artefacts uniquement sur la base de leur ID, nous devons les hacher de la même manière:

```
use std::hash::{Hash, Hasher};

impl Hash for Artifact {
 fn hash<H: Hasher>(&self, hasher: &mut H) {
 // Delegate hashing to the MuseumNumber.
 self.id.hash(hasher);
 }
}
```

(Sinon, ne fonctionnerait pas correctement; comme toutes les tables de hachage, il nécessite que si .) HashSet<Artifact> hash(a) == hash(b) a == b

Cela nous permet de créer un de : HashSet Artifacts

```
let mut collection = HashSet::<Artifact>::new();
```

Comme le montre ce code, même lorsque vous implémentez à la main, vous n'avez pas besoin de savoir quoi que ce soit sur les algorithmes de hachage. reçoit une référence à un , qui représente l'algorithme de hachage. Il vous suffit d'alimenter toutes les données pertinentes pour l'opérateur. Le calcule un code de hachage à partir de tout ce que vous lui donnez. Hash .hash() Hasher Hasher == Hasher

## Utilisation d'un algorithme de hachage personnalisé

La méthode est générique, de sorte que les implémentations montrées précédemment peuvent alimenter les données à n'importe quel type qui implémente . C'est ainsi que Rust prend en charge les algorithmes de hachage enfichables. hash Hash Hasher

Un troisième trait, , est le trait pour les types qui représentent l'état initial d'un algorithme de hachage. Chacun est à usage unique, comme un itérateur : vous l'utilisez une fois et vous le jetez. A est réutilisable. std::hash::BuildHasher Hasher BuildHasher

Chaque contient un qu'il utilise chaque fois qu'il a besoin de calculer un code de hachage. La valeur contient la clé, l'état initial ou d'autres

paramètres dont l'algorithme de hachage a besoin chaque fois qu'il s'exécute. `HashMap` `BuildHasher` `BuildHasher`

Le protocole complet pour le calcul d'un code de hachage ressemble à ceci :

```
use std::hash::{Hash, Hasher, BuildHasher};\n\nfn compute_hash<B, T>(builder: &B, value: &T) -> u64\n where B: BuildHasher, T: Hash\n{\n let mut hasher = builder.build_hasher(); // 1. start the algorithm\n value.hash(&mut hasher); // 2. feed it data\n hasher.finish() // 3. finish, producing a\n}\n\n
```

`HashMap` appelle ces trois méthodes chaque fois qu'il doit calculer un code de hachage. Toutes les méthodes sont ininflammables, donc c'est très rapide.

L'algorithme de hachage par défaut de Rust est un algorithme bien connu appelé SipHash-1-3. SipHash est rapide et très bon pour minimiser les collisions de hachage. En fait, il s'agit d'un algorithme cryptographique : il n'existe aucun moyen efficace connu de générer des collisions SipHash-1-3. Tant qu'une clé différente et imprévisible est utilisée pour chaque table de hachage, Rust est protégé contre une sorte d'attaque par déni de service appelée HashDoS, où les attaquants utilisent délibérément des collisions de hachage pour déclencher les pires performances dans le pire des cas sur un serveur.

Mais peut-être que vous n'en avez pas besoin pour votre application. Si vous stockez de nombreuses petites clés, telles que des entiers ou des chaînes très courtes, il est possible d'implémenter une fonction de hachage plus rapide, au détriment de la sécurité HashDoS. La caisse implémente l'un de ces algorithmes, le hachage Fowler-Noll-Vo (FNV). Pour l'essayer, ajoutez cette ligne à votre `Cargo.toml`: `fnv`

```
[dependencies]\nfnv = "1.0"\n\n
```

Importez ensuite la carte et définissez les types à partir de : `fnv`

```
use fnv::{FnvHashMap, FnvHashSet};\n\n
```

Vous pouvez utiliser ces deux types comme remplacements sans rendez-vous pour et . Un coup d'œil à l'intérieur du code source révèle comment ils sont définis : `HashMap` `HashSet` `fnv`

```
// A `HashMap` using a default FNV hasher.
pub type FnvHashMap<K, V> = HashMap<K, V, FnvBuildHasher>;

// A `HashSet` using a default FNV hasher.
pub type FnvHashSet<T> = HashSet<T, FnvBuildHasher>;
```

La norme et les collections acceptent un paramètre de type supplémentaire facultatif spécifiant l'algorithme de hachage ; et sont des alias de type générique pour et , spécifiant un hasher FNV pour ce paramètre. `HashMap` `HashSet` `FnvHashMap` `FnvHashSet` `HashMap` `Hash` `Set`

## Au-delà des collections standard

La création d'un nouveau type de collection personnalisé dans Rust est à peu près la même que dans n'importe quelle autre langue. Vous organisez les données en combinant les parties fournies par le langage : structs et enums, collections standard, s, es, etc. Pour obtenir un exemple, voir le type défini dans [« Generic Enums »](#). `Option` `Box` `BinaryTree<T>`

Si vous avez l'habitude d'implémenter des structures de données en C++, en utilisant des pointeurs bruts, la gestion manuelle de la mémoire, le placement et les appels de destructeur explicites pour obtenir les meilleures performances possibles, vous trouverez sans aucun doute Rust sûr plutôt limitatif. Tous ces outils sont intrinsèquement dangereux. Ils sont disponibles dans Rust, mais seulement si vous optez pour un code dangereux. [Le chapitre 22](#) montre comment; il inclut un exemple qui utilise du code non sécurisé pour implémenter une collection personnalisée sécurisée. `new`

Pour l'instant, nous allons simplement nous prélasser dans la lueur chaleureuse des collections standard et de leurs API sûres et efficaces. Comme la plupart des bibliothèques standard Rust, ils sont conçus pour s'assurer que le besoin d'écrire est aussi rare que possible. `unsafe`



# Chapitre 17. Chaînes et texte

*La chaîne est une structure de données austère et partout où elle est transmise, il y a beaucoup de duplication de processus. C'est un véhicule parfait pour cacher des informations.*

—Alan Perlis, épigramme #34

Nous avons utilisé les principaux types textuels de Rust, , et , tout au long du livre. Dans [« String Types »](#), nous avons décrit la syntaxe des littéraux de caractères et de chaînes et montré comment les chaînes sont représentées en mémoire. Dans ce chapitre, nous abordons la gestion du texte plus en détail. `String` `str` `char`

Dans ce chapitre :

- Nous vous donnons quelques informations sur Unicode qui devraient vous aider à comprendre la conception de la bibliothèque standard.
- Nous décrivons le type, représentant un seul point de code Unicode. `char`
- Nous décrivons les et types, représentant des séquences possédées et empruntées de caractères Unicode. Ceux-ci ont une grande variété de méthodes pour construire, rechercher, modifier et itérer sur leur contenu. `String` `str`
- Nous couvrons les fonctions de formatage de chaîne de Rust, comme les macros et les macros. Vous pouvez écrire vos propres macros qui fonctionnent avec des chaînes de mise en forme et les étendre pour prendre en charge vos propres types. `println!` `format!`
- Nous donnons un aperçu du support d'expression régulière de Rust.
- Enfin, nous expliquons pourquoi la normalisation Unicode est importante et montrons comment le faire dans Rust.

## Un peu d'arrière-plan Unicode

Ce livre parle de Rust, pas d'Unicode, qui a déjà des livres entiers qui lui sont consacrés. Mais les types de caractères et de chaînes de Rust sont conçus autour d'Unicode. Voici quelques morceaux d'Unicode qui aident à expliquer Rust.

### ASCII, Latin-1 et Unicode

Unicode et ASCII correspondent pour tous les points de code ASCII, de à : par exemple, les deux attribuent le caractère au point de code . De même, Unicode attribue les mêmes caractères que le jeu de caractères ISO/IEC 8859-1, un surensemble ASCII de huit bits à utiliser avec les langues d'Europe occidentale. Unicode appelle cette plage de points de code le *bloc de code Latin-1*, nous nous référerons donc à ISO/IEC 8859-1 par le nom plus évocateur *Latin-1*. 0 0x7f \* 42 0 0xff

Étant donné qu'Unicode est un sur-ensemble de Latin-1, la conversion de Latin-1 en Unicode ne nécessite même pas de table :

```
fn latin1_to_char(latin1: u8) -> char {
 latin1 as char
}
```

La conversion inverse est également triviale, en supposant que les points de code se situent dans la plage Latin-1:

```
fn char_to_latin1(c: char) -> Option<u8> {
 if c as u32 <= 0xff {
 Some(c as u8)
 } else {
 None
 }
}
```

UTF-8

Rust et types représentent le texte à l'aide du formulaire de codage UTF-8. UTF-8 code un caractère sous la forme d'une séquence d'un à quatre octets ([Figure 17-1](#)). `String str`

Graphique 17-1. L'encodage UTF-8

Il existe deux restrictions sur les séquences UTF-8 bien formées. Premièrement, seul l'encodage le plus court pour un point de code donné est considéré comme bien formé; vous ne pouvez pas dépenser quatre octets

pour coder un point de code qui tiendrait dans trois. Cette règle garantit qu'il existe exactement un codage UTF-8 pour un point de code donné. Deuxièmement, UTF-8 bien formé ne doit pas encoder les nombres de bout en bout ou au-delà : ceux-ci sont soit réservés à des fins non caractéristiques, soit entièrement en dehors de la plage d'Unicode. 0xd800 0xffff 0x10ffff

La figure 17-2 en donne quelques exemples.

| UTF-8 encoding (one to four bytes long)             | Code Point Represented                | Range            |
|-----------------------------------------------------|---------------------------------------|------------------|
| 0 0 1 0 1 0 1 0                                     | 0b0101010 == 0x2a                     | '*'              |
| 1 1 0 0 1 1 1 0 1 0 1 1 1 1 0 0                     | 0b01110_111100 == 0x3bc               | 'μ'              |
| 1 1 1 0 1 0 0 1 1 0 0 1 0 0 0 1 1 0                 | 0b1001_001100_000110 == 0x9306        | '鏽' (sabi: rust) |
| 1 1 1 1 0 0 0 0 1 0 0 1 1 1 1 1 1 0 1 0 0 0 0 0 0 0 | 0b000_011111_100110_000000 == 0x1f980 | '🦀' (crab emoji) |

Graphique 17-2. Exemples UTF-8

Notez que, même si l'emoji crabe a un encodage dont l'octet principal ne contribue que des zéros au point de code, il a toujours besoin d'un codage de quatre octets: les codages UTF-8 à trois octets ne peuvent transmettre que des points de code de 16 bits et mesurent 17 bits de long. 0x1f980

Voici un exemple rapide d'une chaîne contenant des caractères avec des codages de longueurs variables :

```
assert_eq!("うどん: udon".as_bytes(),
 &[0xe3, 0x81, 0x86, // う
 0xe3, 0x81, 0xa9, // ど
 0xe3, 0x82, 0x93, // ん
 0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
]);
```

La figure 17-2 montre également certaines propriétés très utiles d'UTF-8 :

- Étant donné que UTF-8 code les points de code à travers comme rien de plus que les octets à travers , une plage d'octets contenant du texte ASCII est valide UTF-8. Et si une chaîne UTF-8 inclut uniquement des caractères ASCII, l'inverse est également vrai : le codage UTF-8 est ascii valide. 0 0x7f 0 0x7f

Il n'en va pas de même pour Latin-1 : par exemple, Latin-1 code comme l'octet , que UTF-8 interpréterait comme le premier octet d'un codage de trois octets. é 0xe9

- En regardant les bits supérieurs de n'importe quel octet, vous pouvez immédiatement dire s'il s'agit du début de l'encodage UTF-8 d'un caractère ou d'un octet au milieu d'un.
- Le premier octet d'un encodage vous indique à lui seul la longueur complète de l'encodage, via ses bits de début.
- Étant donné qu'aucun codage ne dépasse quatre octets, le traitement UTF-8 ne nécessite jamais de boucles illimitées, ce qui est agréable lorsque vous travaillez avec des données non fiables.
- Dans UTF-8 bien formé, vous pouvez toujours dire sans ambiguïté où les codages des caractères commencent et se terminent, même si vous partez d'un point arbitraire au milieu des octets. Les premiers octets UTF-8 et les octets suivants sont toujours distincts, de sorte qu'un encodage ne peut pas démarrer au milieu d'un autre. Le premier octet détermine la longueur totale de l'encodage, de sorte qu'aucun codage ne peut être un préfixe d'un autre. Cela a beaucoup de belles conséquences. Par exemple, la recherche d'un caractère de délimiteur ASCII dans une chaîne UTF-8 ne nécessite qu'une simple analyse de l'octet du délimiteur. Il ne peut jamais apparaître comme une partie d'un encodage multi-octets, il n'est donc pas nécessaire de suivre la structure UTF-8. De même, les algorithmes qui recherchent une chaîne d'octets dans une autre fonctionneront sans modification sur les chaînes UTF-8, même si certains n'examinent même pas chaque octet du texte recherché.

Bien que les codages à largeur variable soient plus compliqués que les codages à largeur fixe, ces caractéristiques rendent UTF-8 plus confortable à utiliser que prévu. La bibliothèque standard gère la plupart des aspects pour vous.

## Directionnalité du texte

Alors que des écritures comme le latin, le cyrillique et le thaï sont écrites de gauche à droite, d'autres écritures comme l'hébreu et l'arabe sont écrites de droite à gauche. Unicode stocke les caractères dans l'ordre dans lequel ils seraient normalement écrits ou lus, de sorte que les octets initiaux d'une chaîne contenant, par exemple, du texte hébreu encodent le caractère qui serait écrit à droite:

```
assert_eq!("בָּרְכָה" .chars().next() , Some('ב'));
```

# Caractères (char)

Un Rust est une valeur 32 bits contenant un point de code Unicode. A est garanti de tomber dans la plage de à ou dans la gamme à ; toutes les méthodes de création et de manipulation des valeurs garantissent que cela est vrai. Le type implémente et , ainsi que tous les traits habituels pour la comparaison, le hachage et la mise en

```
forme. char char 0 0xd7ff 0xe000 0x10ffff char char Copy Clone
```

e

Une tranche de chaîne peut produire un itérateur sur ses caractères avec : `:slice.chars()`

```
assert_eq!("力二".chars().next(), Some('力'));
```

Dans les descriptions qui suivent, la variable est toujours de type `.ch char`

## Classification des caractères

Le type dispose de méthodes pour classer les caractères en quelques catégories courantes, comme indiqué dans [le tableau 17-1](#). Ceux-ci tirent tous leurs définitions d'Unicode. `char`

Tableau 17-1. Méthodes de classification pour le type char

| Méthode                                     | Description                                                                                                                              | Exemples                                                                                        |
|---------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------|
| <code>ch.is_n<br/>umeric<br/>( )</code>     | Caractère numérique. Cela inclut les catégories générales Unicode « Nombre; chiffre » et « Nombre; lettre » mais pas « Nombre; autres ». | '4'.is_<br>numeric<br>( )<br>'↑'.is_<br>numeric<br>( )<br>'⑧'.is_<br>numeric<br>( )             |
| <code>ch.is_a<br/>lphabet<br/>ic()</code>   | Un caractère alphabétique : propriété dérivée « Alphabétique » d'Unicode.                                                                | 'q'.is_<br>alphabe<br>tic()<br>'七'.is_<br>alphabe<br>tic()                                      |
| <code>ch.is_a<br/>lphanum<br/>eric()</code> | Numérique ou alphabétique, comme défini précédemment.                                                                                    | '9'.is_<br>alphanum<br>eric()<br>'餰'.is_<br>alphanum<br>eric()<br>!*'.is<br>_alphanu<br>meric() |

| Méthode                                   | Description                                                                  | Exemples                                                                                             |
|-------------------------------------------|------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------|
| <code>ch.is_w<br/>hitespa<br/>ce()</code> | Un caractère d'espace blanc : propriété de caractère Unicode « WSpace=Y ».   | ' '.is_<br>whitesp<br>ace()<br>'\n'.is_<br>_whitesp<br>ace()<br>'\u{A<br>0}'.is_w<br>hitespa<br>ce() |
| <code>ch.is_c<br/>ontrol<br/>( )</code>   | Un caractère de contrôle : catégorie générale « Autre, contrôle » d'Unicode. | '\n'.is_<br>_contro<br>l()<br>'\u{8<br>5}'.is_c<br>ontrol<br>( )                                     |

Un ensemble parallèle de méthodes se limite à ASCII uniquement, renvoyant pour tout non-ASCII ([Tableau 17-2](#)). `false char`

Table 17-2. ASCII classification methods for `char`

| Method                            | Description                                                                 | Examples                                                                                                         |
|-----------------------------------|-----------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------|
| <code>ch.is_ascii()</code>        | An ASCII character: one whose code point falls between and inclusive. 0 127 | 'n'.is_ascii()<br>!ñ'.is_ascii()                                                                                 |
| <code>ch.is_alpha()</code>        | An upper- or lowercase ASCII letter, in the range or. 'A'..='Z' 'a'..='z'   | 'n'.is_alpha()<br>i_alpha()<br>c()<br>'l'.is_alpha()<br>i_alpha()<br>ic()<br>!ñ'.is_alpha()<br>i_alpha()<br>ic() |
| <code>ch.is_digit()</code>        | Un chiffre ASCII, dans la plage . '0'..='9'                                 | '8'.is_digit()<br>i_digit()<br>!-.is_ascii_digit()<br>!'@'.is_ascii_digit()                                      |
| <code>ch.is_hexdigit()</code>     | Tout caractère des plages , ou . '0'..='9' 'A'..='F' 'a'..='f'              |                                                                                                                  |
| <code>ch.is_alphanumeric()</code> | Un chiffre ASCII ou une lettre majuscule ou minuscule.                      | 'q'.is_alpha()<br>i_alphanumeric()<br>'0'.is_alpha()<br>i_alphanumeric()                                         |

| <b>Method</b>                                                                                   | <b>Description</b>                                                                                 | <b>Examples</b>                                                                                                 |
|-------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------|
| <code>ch.is_a<br/>scii_con<br/>trol()</code>                                                    | Un caractère de contrôle ASCII, y compris 'DEL'.                                                   | <code>'\n'.is_asc<br/>ii_control<br/>()<br/>\x7f'.is_a<br/>scii_contro<br/>l()</code>                           |
| <code>ch.is_a<br/>scii_gra<br/>phic()</code>                                                    | Tout caractère ASCII qui laisse de l'encre sur la page : ni un espace ni un caractère de contrôle. | <code>'Q'.is_asic<br/>i_graphic()<br/>'~'.is_asic<br/>i_graphic()<br/>!' '.is_asic<br/>ii_graphic<br/>()</code> |
| <code>ch.is_a<br/>scii_upp<br/>ercase<br/>(),<br/>ch.is_a<br/>scii_low<br/>ercase<br/>()</code> | Lettres ASCII majuscules et minuscules.                                                            | <code>'z'.is_asic<br/>i_lowercase<br/>()<br/>'Z'.is_asic<br/>i_uppercase<br/>()</code>                          |
| <code>ch.is_a<br/>scii_pun<br/>ctuatio<br/>n()</code>                                           | Tout caractère graphique ASCII qui n'est ni alphabétique ni un chiffre.                            |                                                                                                                 |

| Method                                | Description                                                                                                        | Examples                                                                                   |
|---------------------------------------|--------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------|
| ch.is_a<br>scii_whi<br>tespace<br>( ) | Un caractère d'espace ASCII : espace, tabulation horizontale, saut de ligne, saut de formulaire ou retour chariot. | ' '.is_ascii_whitespace()<br>\n'.is_ascii_whitespace()<br>! '\u{A0}'.is_ascii_whitespace() |

Tout le... sont également disponibles sur le type d'octet : `is_ascii_u8`

```
assert!(32u8.is_ascii_whitespace());
assert!(b'9'.is_ascii_digit());
```

Lorsque vous utilisez ces fonctions, faites attention à implémenter une spécification existante comme une norme de langage de programmation ou un format de fichier, car les classifications peuvent différer de manière surprenante. Par exemple, notez que et diffèrent dans leur traitement de certains

personnages: `is whitespace` `is_ascii_whitespace`

```
let line_tab = '\u{000b}'; // 'line tab', AKA 'vertical tab'
assert_eq!(line_tab.is_whitespace(), true);
assert_eq!(line_tab.is_ascii_whitespace(), false);
```

La fonction implémente une définition d'espace blanc commune à de nombreuses normes Web, alors qu'elle suit la norme Unicode. `char::is_ascii_whitespace` `char::is_whitespace`

## Manipulation des chiffres

Pour gérer les chiffres, vous pouvez utiliser les méthodes suivantes :

`ch.to_digit(radix)`

Décide s'il s'agit d'un chiffre en base . Si c'est le cas, il renvoie , où est un fichier . Sinon, il renvoie . Cela ne reconnaît que les chiffres ASCII, et non la classe plus large de caractères couverts par . Le paramètre peut aller de 2 à 36. Pour les

rayons supérieurs à 10, les lettres ASCII de l'un ou l'autre cas sont considérées comme des chiffres avec des valeurs comprises entre 10 et

```
35. ch radix Some(num) num u32 None char::is_numeric radix x
```

```
std::char::from_digit(num, radix)
```

Fonction libre qui convertit la valeur numérique en si possible. Si peut être représenté par un seul chiffre dans , renvoie , où est le chiffre. Lorsqu'il est supérieur à 10, peut être une lettre minuscule. Sinon, il renvoie

```
.u32 num char num radix from_digit Some(ch) ch radix ch N one
```

C'est l'inverse de . Si est , alors est . S'il s'agit d'un chiffre ASCII ou d'une lettre minuscule, l'inverse est également

```
valable.to_digit std::char::from_digit(num, radix) Some(ch) ch.to_digit(radix) Some(num) ch
```

```
ch.is_digit(radix)
```

Renvoie si est un chiffre ASCII en base . Cela équivaut à

```
.true ch radix ch.to_digit(radix) != None
```

Ainsi, par exemple :

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std::char::from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

## Conversion de casse pour les caractères

Pour la gestion de la casse de caractères :

```
ch.is_lowercase(), ch.is_uppercase()
```

Indiquez s'il s'agit d'un caractère alphabétique minuscule ou majuscule. Ceux-ci suivent les propriétés dérivées en minuscules et en majuscules d'Unicode, de sorte qu'ils couvrent les alphabets non latins comme le grec et le cyrillique et donnent également les résultats attendus pour ASCII. ch

```
ch.to_lowercase(), ch.to_uppercase()
```

Renvoyer des itérateurs qui produisent les caractères des équivalents minuscules et majuscules de , selon les algorithmes de conversion de casse par défaut Unicode : ch

```

let mut upper = 's'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

```

Ces méthodes renvoient un itérateur au lieu d'un seul caractère, car la conversion de casse en Unicode n'est pas toujours un processus un à un :

```

// The uppercase form of the German letter "sharp S" is "SS":
let mut upper = 'ß'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

// Unicode says to lowercase Turkish dotted capital 'İ' to 'i'
// followed by ``\u{307}``, COMBINING DOT ABOVE, so that a
// subsequent conversion back to uppercase preserves the dot.
let ch = 'İ'; // ``\u{130}``
let mut lower = ch.to_lowercase();
assert_eq!(lower.next(), Some('i'));
assert_eq!(lower.next(), Some('\u{307}'));
assert_eq!(lower.next(), None);

```

Pour plus de commodité, ces itérateurs implémentent le trait, de sorte que vous pouvez les transmettre directement à une ou une macro. `std::fmt::Display` `println!` `write!`

## Conversions vers et depuis des entiers

L'opérateur de Rust convertira `a` en n'importe quel type entier, masquant silencieusement tous les bits supérieurs : `as char`

```

assert_eq!('B' as u32, 66);
assert_eq!('餾' as u8, 66); // upper bits truncated
assert_eq!('二' as i8, -116); // same

```

L'opérateur convertira n'importe quelle valeur en `a`, et implémente également, mais les types entiers plus larges peuvent représenter des points de code non valides, donc pour ceux que vous devez utiliser, qui renvoie

```
: as u8 char char From<u8> std::char::from_u32 Option<char>
```

```
assert_eq!(char::from(66), 'B');
assert_eq!(std::char::from_u32(0x9942), Some('餳'));
assert_eq!(std::char::from_u32(0xd800), None); // reserved for UTF-16
```

## String et str

Les rouilles et les types de rouille sont garantis pour ne contenir que des UTF-8 bien formés. La bibliothèque garantit cela en limitant les façons dont vous pouvez créer et les valeurs et les opérations que vous pouvez effectuer sur eux, de sorte que les valeurs soient bien formées lorsqu'elles sont introduites et le restent lorsque vous travaillez avec elles. Toutes leurs méthodes protègent cette garantie: aucun fonctionnement sûr sur eux ne peut introduire UTF-8 mal formé. Cela simplifie le code qui fonctionne avec le texte. `String` `str` `String` `str`

Rust place les méthodes de gestion de texte sur l'un ou l'autre ou selon que la méthode a besoin d'un tampon redimensionnable ou qu'il s'agit d'un contenu juste pour utiliser le texte en place. Depuis les déréférences à `,` chaque méthode définie sur est également directement disponible sur. Cette section présente les méthodes des deux types, regroupées par fonction approximative. `str` `String` `String` `&str` `str` `String`

Ces méthodes indexent le texte par décalage d'octets et mesurent sa longueur en octets plutôt qu'en caractères. En pratique, compte tenu de la nature d'Unicode, l'indexation par caractère n'est pas aussi utile qu'il n'y paraît, et les décalages d'octets sont plus rapides et plus simples. Si vous essayez d'utiliser un décalage d'octets qui atterrit au milieu de l'encodage UTF-8 d'un personnage, la méthode panique, de sorte que vous ne pouvez pas introduire UTF-8 mal formé de cette façon.

`A` est implémenté comme un wrapper autour de `a` qui garantit que le contenu du vecteur est toujours bien formé UTF-8. Rust ne changera jamais pour utiliser une représentation plus compliquée, vous pouvez donc supposer que les caractéristiques de performance des actions. `String` `Vec<u8>` `String` `String` `Vec`

Dans ces explications, les variables ont les types [donnés dans le tableau 17-3.](#)

Tableau 17-3. Types de variables utilisées dans les explications

| Variable | Type présumé                                                                                                                |
|----------|-----------------------------------------------------------------------------------------------------------------------------|
| string   | String                                                                                                                      |
| slice    | &str ou quelque chose qui fait référence à l'un d'entre eux, comme ou String Rc<String>                                     |
| ch       | char                                                                                                                        |
| n        | usize, une longueur                                                                                                         |
| i, j     | usize, un décalage d'octets                                                                                                 |
| range    | Plage de décalages d'octets, soit entièrement bornés comme , soit partiellement délimités comme , , ou usize i...j i...j .. |
| patter n | Tout type de motif: , , , , ou char String &str &[char] FnMut(char) -> bool                                                 |

Nous décrivons les types de modèles dans [« Modèles pour la recherche de texte ».](#)

## Création de valeurs de chaîne

Il existe plusieurs façons courantes de créer des valeurs : String

*String::new()*

Renvoie une chaîne fraîche et vide. Il n'a pas de tampon alloué au tas, mais en allouera un si nécessaire.

*String::with\_capacity(n)*

Renvoie une chaîne vide avec un tampon pré-alloué pour contenir au moins des octets. Si vous connaissez la longueur de la chaîne que vous générerez à l'avance, ce constructeur vous permet d'obtenir la taille de la mémoire tampon correctement dès le début, au lieu de redimensionner la mémoire tampon au fur et à mesure que vous construisez la chaîne. La chaîne continuera à développer sa mémoire tampon si nécessaire si sa longueur dépasse les octets. Comme les vecteurs, les chaînes ont , et les méthodes, mais généralement la logique

```
d'allocation par défaut est
correcte. n n capacity reserve shrink_to_fit
str_slice.to_string()
```

Alloue un produit dont le contenu est une copie de . Nous avons utilisé des expressions comme tout au long du livre pour faire des s à partir de littéraux de chaîne. String str\_slice "literal  
text".to\_string() String  
**iter.collect()**

Construit une chaîne en concaténant les éléments d'un itérateur, qui peuvent être , ou des valeurs. Par exemple, pour supprimer tous les espaces d'une chaîne, vous pouvez écrire : char &str String

```
let spacey = "man hat tan";
let spaceless: String =
 spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

L'utilisation de cette méthode tire parti de la mise en œuvre du trait de caractère. collect String std::iter::FromIterator

```
slice.to_owned()
```

Renvoie une copie de la tranche sous la forme d'un fichier . Le type ne peut pas implémenter : le trait nécessiterait sur a de renvoyer une valeur, mais n'est pas dimensionné. Cependant, implémente , ce qui permet à l'implémenteur de spécifier son équivalent propre. String str Clone clone &str str str &str ToOwned

## Simple Inspection

Ces méthodes obtiennent des informations de base à partir de tranches de chaîne :

```
slice.len()
```

Longueur de , en octets. slice

```
slice.is_empty()
```

True si . slice.len() == 0

```
slice[range]
```

Renvoie une tranche empruntant la partie donnée de . Les plages partiellement délimitées et non bornées sont OK ; par exemple: slice

```

let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
assert_eq!(&full[2..4], "ok");
assert_eq!(full[..].len(), 11);
assert_eq!(full[5..].contains("boo"), false);

```

Notez que vous ne pouvez pas indexer une tranche de chaîne avec une seule position, comme . La récupération d'un seul caractère à un décalage d'octet donné est un peu maladroite : vous devez produire un itérateur sur la tranche et lui demander d'analyser l'UTF-8 d'un caractère : `slice[i] chars`

```

let parenthesized = "Rust (餰)";
assert_eq!(parenthesized[6..].chars().next(), Some('餰'));

```

Cependant, vous devriez rarement avoir besoin de le faire. Rust a des façons beaucoup plus agréables d'itérer sur les tranches, que nous décrivons dans [« Itérer sur le texte »](#).

`slice.split_at(i)`

Renvoie un tuple de deux tranches partagées empruntées à : la portion jusqu'au décalage d'octets et la partie qui la suit. En d'autres termes, cela renvoie `.slice i (slice[..i], slice[i..])`

`slice.is_char_boundary(i)`

True si le décalage d'octet se situe entre les limites de caractères et convient donc comme décalage dans `.i slice`

Naturellement, les tranches peuvent être comparées pour l'égalité, ordonnées et hachées. La comparaison ordonnée traite simplement la chaîne comme une séquence de points de code Unicode et les compare dans l'ordre lexicographique.

## Ajout et insertion de texte

Les méthodes suivantes ajoutent du texte à un `:String`

`string.push(ch)`

Ajoute le caractère à la fin `.ch string`

`string.push_str(slice)`

Ajoute le contenu complet de `.slice`

```
string.extend(iter)
```

Ajoute les éléments produits par l'itérateur à la chaîne. L'itérateur peut produire , ou des valeurs. Ce sont les implémentations de

```
: iter char str String String std::iter::Extend
```

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");
```

```
string.insert(i, ch)
```

Insère le caractère unique au décalage d'octet dans . Cela implique de déplacer tous les caractères après pour faire de la place pour , de sorte que la construction d'une chaîne de cette façon peut nécessiter un temps quadratique dans la longueur de la chaîne. ch i string i ch

```
string.insert_str(i, slice)
```

Cela fait la même chose pour , avec la même mise en garde de performance. slice

String implémente , ce qui signifie que les et les macros peuvent ajouter du texte mis en forme à s

```
: std::fmt::Write write! writeln! String
```

```
use std::fmt::Write;
```

```
let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know", "rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n\
 His house is in the village though;\n");
```

Puisque et sont conçus pour écrire dans des flux de sortie, ils renvoient un , ce que Rust se plaint si vous ignorez. Ce code utilise l'opérateur pour le gérer, mais écrire sur un est en fait infaillible, donc dans ce cas, l'appel serait OK aussi. write! writeln! Result ? String .unwrap()

Puisque implémente et , vous pouvez écrire du code comme ceci: String Add<&str> AddAssign<&str>

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");
```

```
right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

Lorsqu'il est appliqué à des chaînes, l'opérateur prend son opérande gauche par valeur, de sorte qu'il peut réellement le réutiliser à la suite de l'ajout. Par conséquent, si la mémoire tampon de l'opérande gauche est suffisamment grande pour contenir le résultat, aucune allocation n'est nécessaire.

+ String

Dans un manque regrettable de symétrie, l'opérande gauche de ne peut pas être un , vous ne pouvez donc pas écrire: + &str

```
let parenthetical = "(" + string + ")";
```

Vous devez plutôt écrire :

```
let parenthetical = ".to_string() + &string + ")";
```

Cependant, cette restriction décourage la construction de chaînes à partir de la fin vers l'arrière. Cette approche fonctionne mal car le texte doit être déplacé à plusieurs reprises vers la fin de la mémoire tampon.

Construire des chaînes du début à la fin en ajoutant de petits morceaux, cependant, est efficace. A se comporte comme un vecteur, doublant toujours au moins la taille de son tampon lorsqu'il a besoin de plus de capacité. Cela permet de conserver les frais généraux de recopie proportionnellement à la taille finale. Néanmoins, l'utilisation de la création de chaînes avec la bonne taille de tampon pour commencer évite le redimensionnement et peut réduire le nombre d'appels à l'allocateur de tas.

String String::with\_capacity

## Suppression et remplacement de texte

String dispose de quelques méthodes pour supprimer du texte (celles-ci n'affectent pas la capacité de la chaîne; utilisez si vous avez besoin de libérer de la mémoire): shrink\_to\_fit

```
string.clear()
```

Réinitialise la chaîne vide. string

```
string.truncate(n)
```

Supprime tous les caractères après le décalage d'octets , en laissant avec une longueur d'au plus . Si est plus court que les octets, cela n'a aucun effet.

```
n string n string n
```

```
string.pop()
```

Supprime le dernier caractère de , le cas échéant, et le renvoie sous la forme d'un fichier.

```
string Option<char>
```

```
string.remove(i)
```

Supprime le caractère au décalage d'octet et le renvoie, en déplaçant tous les caractères suivants vers l'avant. Cela prend du temps linéaire dans le nombre de caractères suivants.

```
i string
```

```
string.drain(range)
```

Renvoie un itérateur sur la plage donnée d'index d'octets et supprime les caractères une fois l'itérateur supprimé. Caractères après que la plage est décalée vers l'avant :

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

Si vous souhaitez simplement supprimer la plage, vous pouvez simplement laisser tomber l'itérateur immédiatement, sans en tirer d'éléments:

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

```
string.replace_range(range, replacement)
```

Remplace la plage donnée par la tranche de chaîne de remplacement donnée. La tranche n'a pas besoin d'avoir la même longueur que la plage remplacée, mais à moins que la plage remplacée ne se termine par , cela nécessitera de déplacer tous les octets après la fin de la plage:

```
string string
```

```
let mut beverage = "a piña colada".to_string();
beverage.replace_range(2..7, "kahlua"); // 'ñ' is two bytes!
assert_eq!(beverage, "a kahlua colada");
```

## Conventions de recherche et d'itération

Les fonctions de bibliothèque standard de Rust pour la recherche de texte et l’itération sur le texte suivent certaines conventions de dénomination pour les rendre plus faciles à retenir:

*r*

La plupart des opérations traitent le texte du début à la fin, mais les opérations dont les noms commencent par `r` fonctionnent de bout en début. Par exemple, `rsplit` est la version de bout en bout de `split`. Dans certains cas, le changement de direction peut affecter non seulement l’ordre dans lequel les valeurs sont produites, mais aussi les valeurs elles-mêmes. Voir le diagramme de [la figure 17-3](#) pour un exemple de ceci.

*n*

Les itérateurs dont les noms se terminent par `n` se limitent à un nombre donné de correspondances.

*\_indices*

Les itérateurs dont les noms se terminent par `_indices` produisent, avec leurs valeurs d’itération habituelles, les décalages d’octets dans la trame à laquelle ils apparaissent.

La bibliothèque standard ne fournit pas toutes les combinaisons pour chaque opération. Par exemple, de nombreuses opérations n’ont pas besoin d’une variante, car il est assez facile de simplement mettre fin à l’itération plus tôt.

## Modèles de recherche de texte

Lorsqu’une fonction de bibliothèque standard doit rechercher, faire correspondre, diviser ou découper du texte, elle accepte plusieurs types différents pour représenter ce qu’il faut rechercher :

```
let haystack = "One fine day, in the middle of the night";

assert_eq!(haystack.find(','), Some(12));
assert_eq!(haystack.find("night"), Some(35));
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

Ces types sont appelés *modèles* et la plupart des opérations les prennent en charge :

```

assert_eq!(## Elephants"
 .trim_start_matches(|ch: char| ch == '#' || ch.is_whitespace())
 "Elephants");

```

La bibliothèque standard prend en charge quatre principaux types de modèles :

- A en tant que motif correspond à ce caractère. `char`
- A ou ou comme un motif correspond à une sous-chaîne égale au motif. `String &str &&str`
- Une fermeture en tant que motif correspond à un caractère unique pour lequel la fermeture renvoie true. `FnMut(char) -> bool`
- A en tant que modèle (pas un , mais une tranche de valeurs) correspond à n'importe quel caractère unique qui apparaît dans la liste. Notez que si vous écrivez la liste en tant que littéral de tableau, vous devrez peut-être appeler pour obtenir le bon type : `&[char] &str char as_ref()`

```

let code = "\t function noodle() { ";
assert_eq!(code.trim_start_matches([' ', '\t']).as_ref(),
 "function noodle() { ");
// Shorter equivalent: &[' ', '\t'][..]

```

Sinon, Rust sera confondu par le type de tableau à taille fixe , qui n'est malheureusement pas un type de modèle. `&[char; 2]`

Dans le code de la bibliothèque, un modèle est un type qui implémente le trait. Les détails de ne sont pas encore stables, vous ne pouvez donc pas l'implémenter pour vos propres types dans Rust stable, mais la porte est ouverte pour permettre des expressions régulières et d'autres modèles sophistiqués à l'avenir. Rust garantit que les types de modèles pris en charge aujourd'hui continueront à fonctionner à l'avenir. `std::str::Pattern`

## Recherche et remplacement

Rust a quelques méthodes pour rechercher des motifs dans les tranches et éventuellement les remplacer par du nouveau texte:

`slice.contains(pattern)`

Renvoie true si contient une correspondance pour `.slice pattern`

```
slice.starts_with(pattern), slice.ends_with(pattern)
```

Renvoyer true si le texte initial ou final de ' correspond à

```
:slice pattern
```

```
assert!("2017".starts_with(char::is_numeric));
```

```
slice.find(pattern), slice.rfind(pattern)
```

Renvoyer si contient une correspondance pour , où est le décalage d'octet auquel le motif apparaît. La méthode renvoie la première correspondance, la dernière

```
:Some(i) slice pattern i find rfind
```

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

```
slice.replace(pattern, replacement)
```

Renvoie un nouveau formé en remplaçant avec empreusement tous les matchs par : String pattern replacement

```
assert_eq!("The only thing we have to fear is fear itself"
 .replace("fear", "spin"),
 "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
 .replace(|ch:char| !ch.is_alphanumeric(), ""),
 "BorrowandBorrowMut");
```

Parce que le remplacement est fait avec empreusement, le comportement de 'sur les matchs qui se chevauchent peut être surprenant. Ici, il y a quatre instances du modèle, mais la deuxième et la quatrième ne correspondent plus après que la première et la troisième sont remplacées: .replace() "aba"

```
assert_eq!("cabababababbage"
 .replace("aba", "***"),
 "c***b***babbage")
```

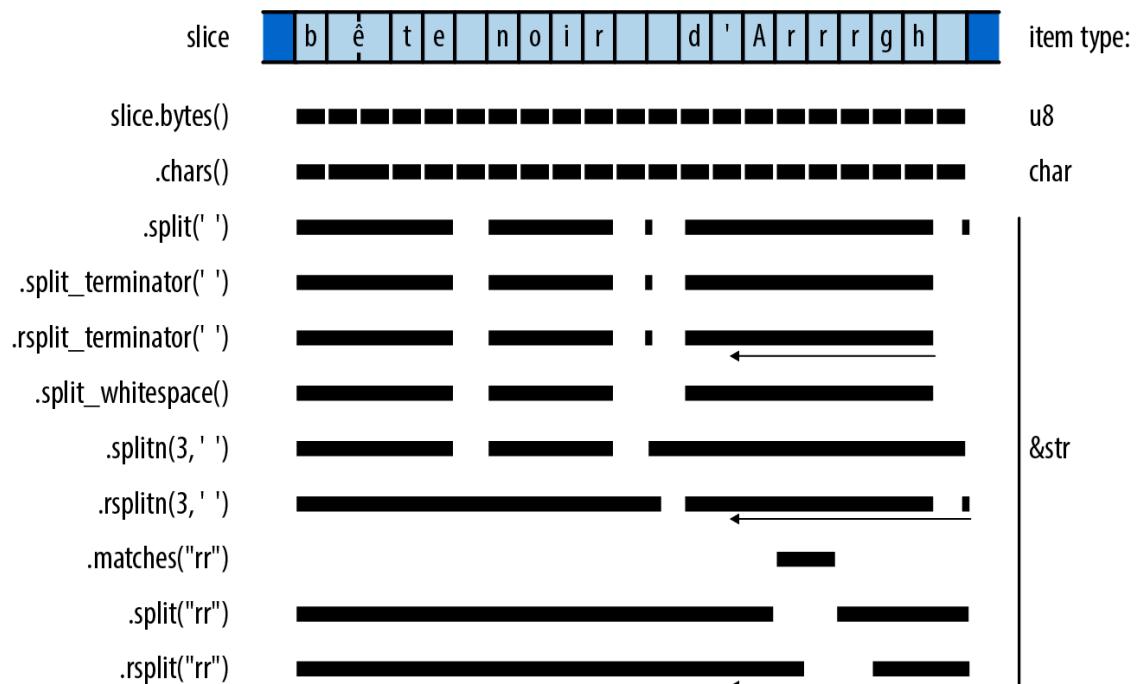
```
slice.replace(pattern, replacement, n)
```

Cela fait de même, mais remplace tout au plus les premiers matchs. n

# Itération sur le texte

La bibliothèque standard offre plusieurs façons d’itérer sur le texte d’une tranche. [La figure 17-3 en](#) montre des exemples.

Vous pouvez considérer les familles comme étant complémentaires les unes des autres: les divisions sont les fourchettes entre les matchs. `split match`



Graphique 17-3. Quelques façons d’itérer sur une tranche

La plupart de ces méthodes renvoient des itérateurs réversibles (c'est-à-dire qu'ils implémentent) : l'appel de leur méthode d'adaptateur vous donne un itérateur qui produit les mêmes éléments, mais dans l'ordre inverse. `DoubleEndedIterator .rev()`

`slice.chars()`

Renvoie un itérateur sur les caractères de `'slice`

`slice.char indices()`

Renvoie un itérateur sur les caractères de et leurs décalages d'octets : `:slice`

```
assert_eq!("élan".char_indices().collect::<Vec<_>>(),
 vec![(0, 'é'), // has a two-byte UTF-8 encoding
 (2, 'l'),
 (3, 'a'),
 (4, 'n')));
```

Notez que cela n'est pas équivalent à , car il fournit le décalage d'octet de chaque caractère dans la tranche, au lieu de simplement numérotter les caractères. `.chars().enumerate()`

#### `slice.bytes()`

Renvoie un itérateur sur les octets individuels de , exposant le codage UTF-8 : `slice`

```
assert_eq!("élan".bytes().collect::<Vec<_>>(),
 vec![195, 169, b'l', b'a', b'n']);
```

#### `slice.lines()`

Renvoie un itérateur sur les lignes de . Les lignes sont terminées par ou . Chaque article produit est un emprunt auprès de . Les éléments n'incluent pas les caractères de fin des lignes. `slice "\n" "\r\n" &str slice`

#### `slice.split(pattern)`

Renvoie un itérateur sur les parties de séparées par des correspondances de . Cela produit des chaînes vides entre les correspondances immédiatement adjacentes, ainsi que pour les correspondances au début et à la fin de . `slice pattern slice`

L'itérateur renvoyé n'est pas réversible si est un . De tels modèles peuvent produire différentes séquences de correspondances en fonction de la direction à partir de laquelle vous scannez, ce que les itérateurs réversibles sont interdits de faire. Au lieu de cela, vous pourrez peut-être utiliser la méthode décrite ci-dessous. `pattern &str rsplit`

#### `slice.rsplit(pattern)`

Cette méthode est la même, mais scanne de bout en bout, produisant des correspondances dans cet ordre. `slice`

```
slice.split_terminator(pattern),
slice.rsplit_terminator(pattern)
```

Ceux-ci sont similaires, sauf que le motif est traité comme un terminal, pas un séparateur : si les correspondances à la toute fin de , les itérateurs ne produisent pas de tranche vide représentant la chaîne vide entre cette correspondance et la fin de la tranche, comme et le font. Par exemple: `pattern slice split rsplit`

```

// The ':' characters are separators here. Note the final "".
assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
 vec!["jimb", "1000", "Jim Blandy", ""]);

// The '\n' characters are terminators here.
assert_eq!("127.0.0.1 localhost\n\
 127.0.0.1 www.reddit.com\n\
 .split_terminator('\n').collect::<Vec<_>>(),
 vec!["127.0.0.1 localhost",
 "127.0.0.1 www.reddit.com"]);
 // Note, no final ""!
```

### `slice.splitn(n, pattern), slice.rsplitn(n, pattern)`

Ceux-ci sont comme et , sauf qu'ils divisent la chaîne en au plus des tranches, à la première ou à la dernière correspondance pour `.split` `rsplit` `n` `n-1` `pattern`

### `slice.split_whitespace(), slice.split_ascii_whitespace()`

Renvoyer un itérateur sur les parties séparées par des espaces blancs de . Une série de plusieurs caractères d'espace blanc est considérée comme un séparateur unique. L'espace blanc de fin est ignoré. `slice`

La méthode utilise la définition Unicode d'espace blanc, telle qu'implémentée par la méthode sur . La méthode utilise à la place, ce qui ne reconnaît que les caractères d'espace

`ASCII.split_whitespace` `is_whitespace` `char` `split_ascii_whitespace` `char::is_ascii_whitespace`

```
let poem = "This is just to say\n\
 I have eaten\n\
 the plums\n\
 again\n";
```

```
assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
 vec!["This", "is", "just", "to", "say",
 "I", "have", "eaten", "the", "plums",
 "again"]);
```

### `slice.matches(pattern)`

Renvoie un itérateur sur les correspondances pour dans la tranche. est le même, mais itére de bout en bout. `pattern slice.rmatches(pattern)`

```
slice.match_indices(pattern),
slice.rmatch_indices(pattern)
```

Ceux-ci sont similaires, sauf que les éléments produits sont des paires, où est le décalage d'octets auquel la correspondance commence et est la tranche correspondante. `(offset, match) offset match`

## Coupe

*Couper une* chaîne revient à supprimer du texte, généralement des espaces blancs, du début ou de la fin de la chaîne. Il est souvent utile pour nettoyer les entrées lues à partir d'un fichier où l'utilisateur peut avoir mis en retrait du texte pour plus de lisibilité ou accidentellement laissé un espace blanc de fin sur une ligne.

```
slice.trim()
```

Renvoie une sous-section qui omet tout espace blanc de début et de fin. omet uniquement l'espace blanc de début, uniquement l'espace de fin : `slice slice.trim_start() slice.trim_end()`

```
assert_eq!("\t*.\rs ".trim(), "*.\rs");
assert_eq!("\t*.\rs ".trim_start(), "*.\rs ");
assert_eq!("\t*.\rs ".trim_end(), "\t*.\rs");
```

```
slice.trim_matches(pattern)
```

Renvoie une sous-section qui omet toutes les correspondances du début et de la fin. Les méthodes font de même pour les matchs de début ou de fin uniquement

```
:slice pattern trim_start_matches trim_end_matches
```

```
assert_eq!("001990".trim_start_matches('0'), "1990");
```

```
slice.strip_prefix(pattern), slice.strip_suffix(pattern)
```

Si commence par , renvoie la colonne en maintenant la tranche enfoncée avec le texte correspondant supprimé. Sinon, il renvoie . La méthode est similaire, mais vérifie une correspondance à la fin de la chaîne. `slice pattern strip_prefix Some None strip_suffix`

Ceux-ci sont comme et , sauf qu'ils renvoient un , et qu'une seule copie de est

```
supprimée: trim_start_matches trim_end_matches Option pattern
```

```
let slice = "banana";
assert_eq!(slice.strip_suffix("na"),
 Some("bana"))
```

## Conversion de casse pour les chaînes

Les méthodes et renvoyer une chaîne fraîchement allouée contenant le texte de converti en majuscules ou minuscules. Le résultat peut ne pas être de la même longueur que ; voir [« Conversion de casse pour les caractères »](#) pour plus de

détails. `slice.to_uppercase()` `slice.to_lowercase()` `slice` `slice`

## Analyse d'autres types à partir de chaînes

Rust fournit des caractéristiques standard pour analyser les valeurs des chaînes et produire des représentations textuelles des valeurs.

Si un type implémente le trait, il fournit un moyen standard d'analyser une valeur à partir d'une tranche de chaîne : `std::str::FromStr`

```
pub trait FromStr: Sized {
 type Err;
 fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

Tous les types de machines habituels implémentent : `FromStr`

```
use std::str::FromStr;

assert_eq!(usize::from_str("3628800"), Ok(3628800));
assert_eq!(f64::from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64::from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

Le type implémente également , pour les chaînes avec un seul caractère : `char` `FromStr`

```

assert_eq!(char::from_str("é"), Ok('é'));
assert!(char::from_str("abcdefg").is_err());

```

Le type, une détention d'une adresse Internet IPv4 ou IPv6, implémente également:

```
std::net::IpAddr enum FromStr
```

```

use std::net::IpAddr;

let address = IpAddr::from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
 IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34, 0x7a50]));

```

Les tranches de chaîne ont une méthode qui analyse la tranche dans le type de votre choix, en supposant qu'elle implémente . Comme avec , vous devrez parfois préciser quel type vous voulez, donc ce n'est pas toujours beaucoup plus lisible que d'appeler

directement: parse FromStr Iterator::collect parse from\_str

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

## Conversion d'autres types en chaînes

Il existe trois façons principales de convertir des valeurs non textuelles en chaînes :

- Les types qui ont un formulaire imprimé naturel lisible par l'homme peuvent implémenter le trait, ce qui vous permet d'utiliser le spécificateur de format dans la macro : std::fmt::Display {} format!

```

assert_eq!(format!("{} , wow", "doge"), "doge, wow");
assert_eq!(format!("{}" , true), "true");
assert_eq!(format!("({:.3}, {:.3})" , 0.5, f64::sqrt(3.0)/2.0),
 "(0.500, 0.866)");

// Using `address` from above.
let formatted_addr: String = format!("{}" , address);
assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");

```

Tous les types numériques de machine de Rust implémentent , tout comme les caractères, les chaînes et les tranches. Les types de pointeurs intelligents , , et implémenter si lui-même le fait: leur forme affichée est simplement celle de leur référent. Les conteneurs aiment et ne sont pas

implémentés, car il n'existe pas de forme naturelle lisible par l'homme pour ces

```
types. Display Box<T> Rc<T> Arc<T> Display T Vec HashMap Dis
play
```

- Si un type implémente , la bibliothèque standard implémente automatiquement le trait pour cela, dont la seule méthode peut être plus pratique lorsque vous n'avez pas besoin de la flexibilité de

```
:Display std::str::ToString to_string format!
```

```
// Continued from above.
assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
```

Le trait est antérieur à l'introduction et est moins flexible. Pour vos propres types, vous devez généralement implémenter au lieu de

```
.ToString Display Display ToString
```

- Chaque type public de la bibliothèque standard implémente , ce qui prend une valeur et la formate sous forme de chaîne d'une manière utile aux programmeurs. Le moyen le plus simple d'utiliser pour produire une chaîne consiste à utiliser le spécificateur de format de la macro :std::fmt::Debug Debug format! {::?}

```
// Continued from above.
let addresses = vec![address,
 IpAddr::from_str("192.168.0.1")?];
assert_eq!(format!("{}:{:?}", addresses),
 "[fe80::3ea9:f4ff:fe34:7a50, 192.168.0.1]");
```

Cela tire parti d'une implémentation générale de pour , pour tout ce qui implémente lui-même . Tous les types de collection de Rust ont de telles implémentations. Debug Vec<T> T Debug

Vous devez également implémenter pour vos propres types. Habituellement, il est préférable de laisser Rust dériver une implémentation, comme nous l'avons fait pour le type dans le [chapitre](#)

[12](#): Debug Complex

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im: f64 }
```

Les traits de mise en forme ne sont que deux parmi plusieurs que la macro et ses parents utilisent pour mettre en forme les valeurs sous forme de texte. Nous couvrirons les autres et expliquerons comment les

implémenter tous dans [« Formatage des valeurs »](#). Display Debug format!

## Emprunt en tant qu'autres types textuels

Vous pouvez emprunter le contenu d'une tranche de plusieurs manières différentes :

- Les tranches et s implémentent , , et . De nombreuses fonctions de bibliothèque standard utilisent ces traits comme limites sur leurs types de paramètres, de sorte que vous pouvez leur transmettre directement des tranches et des chaînes, même lorsqu'elles veulent vraiment un autre type. Voir [« AsRef et AsMut »](#) pour une explication plus détaillée.`String AsRef<str> AsRef<[u8]> AsRef<Path> AsRef<OsStr>`
- Les tranches et les chaînes implémentent également le trait. et utiliser pour faire fonctionner s bien comme des clés dans une table. Voir [« Borrow and BorrowMut »](#) pour plus de détails.`std::borrow::Borrow<str> HashMap BTreeMap Borrow String`

## Accès au texte en UTF-8

Il existe deux façons principales d'obtenir les octets représentant le texte, selon que vous souhaitez vous approprier les octets ou simplement les emprunter :

`slice.as_bytes()`

Emprunte les octets de . Comme il ne s'agit pas d'une référence modifiable, on peut supposer que ses octets resteront bien formés UTF-8.`slice & [u8] slice`

`string.into_bytes()`

Prend possession et renvoie un octet de la chaîne par valeur. Il s'agit d'une conversion bon marché, car elle remet simplement ce que la chaîne utilisait comme tampon. Comme il n'existe plus, il n'est pas nécessaire que les octets continuent d'être bien formés UTF-8, et l'appelant est libre de le modifier à sa guise.`string Vec<u8> Vec<u8> string Vec<u8>`

## Production de texte à partir de données UTF-8

Si vous avez un bloc d'octets qui, selon vous, contient des données UTF-8, vous disposez de quelques options pour les convertir en s ou en tranches,

selon la façon dont vous souhaitez gérer les erreurs : String

### `str::from_utf8(byte_slice)`

Prend une tranche d'octets et renvoie un : soit s'il contient utf-8 bien formé, soit une erreur dans le cas contraire. `&[u8] Result Ok(&str) byte_slice`

### `String::from_utf8(vec)`

Tente de construire une chaîne à partir d'une valeur transmise par. S'il contient UTF-8 bien formé, renvoie , où a pris possession de pour une utilisation comme tampon. Aucune allocation de tas ou copie du texte n'a lieu. `Vec<u8> vec from_utf8 Ok(string) string vec`

Si les octets ne sont pas valides UTF-8, cela renvoie , où est une valeur d'erreur. L'appel vous redonne le vecteur d'origine , afin qu'il ne soit pas perdu lorsque la conversion échoue

```
:Err(e) e FromUtf8Error e.into_bytes() vec
```

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(), Some("鍾".to_string()));

let bad_utf8: Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// Since String::from_utf8 failed, it didn't consume the original
// vector, and the error value hands it back to us unharmed.
assert_eq!(result.unwrap_err().into_bytes(),
 vec![0x9f, 0xf0, 0xa6, 0x80]);
```

### `String::from_utf8_lossy(byte_slice)`

Tente de construire un ou à partir d'une tranche d'octets partagée. Cette conversion réussit toujours, remplaçant tout UTF-8 mal formé par des caractères de remplacement Unicode. La valeur renvoyée est une valeur qui emprunte un directement à s'il contient utf-8 bien formé ou possède un nouveau alloué avec des caractères de remplacement substitués aux octets mal formés. Par conséquent, lorsqu'il est bien formé, aucune allocation ou copie de tas n'a lieu.

Nous discutons plus en détail dans [« Différer](#)

[l'allocation »](#). `String &str &`

```
[u8] Cow<str> &str byte_slice String byte_slice Cow<str>
```

### `String::from_utf8_unchecked(vec)`

Si vous savez pertinemment que votre contient UTF-8 bien formé, alors vous pouvez appeler cette fonction dangereuse. Cela se termine simplement par un et

le renvoie, sans examiner les octets du tout. Vous êtes responsable de vous assurer que vous n'avez pas introduit UTF-8 mal formé dans le système, c'est pourquoi cette fonction est marquée . Vec<u8> vec String unsafe str::from\_utf8\_unchecked(byte\_slice)

De même, cela prend un et le renvoie sous la forme d'un , sans vérifier s'il contient UTF-8 bien formé. Comme pour , vous êtes responsable de vous assurer que c'est sûr. &[ u8 ] &str String::from\_utf8\_unchecked

## Report de l'allocation

Supposons que vous souhaitiez que votre programme accueille l'utilisateur. Sous Unix, vous pouvez écrire :

```
fn get_name() -> String {
 std::env::var("USER") // Windows uses "USERNAME"
 .unwrap_or("whoever you are").to_string()
}

println!("Greetings, {}!", get_name());
```

Pour les utilisateurs d'Unix, cela les accueille par nom d'utilisateur. Pour les utilisateurs de Windows et les personnes tragiquement anonymes, il fournit un texte de stock alternatif.

La fonction renvoie un — et a de bonnes raisons de le faire que nous n'aborderons pas ici. Mais cela signifie que le texte de stock alternatif doit également être retourné en tant que fichier . C'est décevant : lorsqu'une chaîne statique renvoie une chaîne statique, aucune allocation ne devrait être nécessaire. std::env::var String String get\_name

Le nœud du problème est que parfois la valeur de retour de devrait être une propriété, parfois elle devrait être un , et nous ne pouvons pas savoir lequel ce sera jusqu'à ce que nous exécutions le programme. Ce caractère dynamique est l'indice à considérer pour utiliser , le type de clone sur écriture qui peut contenir des données possédées ou empruntées. get\_name String &'static str std::borrow::Cow

Comme expliqué dans [« Borrow and ToOwned at Work: The Humble Cow »](#), est un enum avec deux variantes: et . détient une référence , et détiennent la version propriétaire de : pour , pour , et ainsi de suite. Que ce soit ou , un peut toujours produire un pour vous d'utiliser. En fait, déréférence à , se comportant comme une sorte de pointeur

```
intelligent. Cow<'a, T> Owned Borrowed Borrowed &'a
T Owned &T String &str Vec<i32> &
[i32] Owned Borrowed Cow<'a, T> &T Cow<'a, T> &T
```

Modification pour renvoyer un résultat dans les éléments suivants

```
: get_name Cow
```

```
use std::borrow::Cow;

fn get_name() -> Cow<'static, str> {
 std::env::var("USER")
 .map(|v| Cow::Owned(v))
 .unwrap_or(Cow::Borrowed("whoever you are"))
}
```

Si cela réussit à lire la variable d'environnement, le renvoie le résultat sous la forme d'un fichier . En cas d'échec, le renvoie sa statique sous la forme d'un fichier . L'appelant peut rester inchangé

```
: "USER" map String Cow::Owned unwrap_or &str Cow::Borrowed

println!("Greetings, {}!", get_name());
```

Tant qu'il implémente le trait, l'affichage d'un produit les mêmes résultats que l'affichage d'un . T std::fmt::Display Cow<'a, T> T

Cow est également utile lorsque vous avez besoin ou non de modifier un texte que vous avez emprunté. Lorsqu'aucun changement n'est nécessaire, vous pouvez continuer à l'emprunter. Mais le comportement homonyme de clone sur écriture peut vous donner une copie propre et mutable de la valeur à la demande. s'assure que is , en appliquant l'implémentation de la valeur si nécessaire, puis renvoie une référence modifiable à la valeur. Cow Cow to\_mut Cow Cow::Owned ToOwned

Donc, si vous constatez que certains de vos utilisateurs, mais pas tous, ont des titres par lesquels ils préféreraient être adressés, vous pouvez dire:

```
fn get_title() -> Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
 name.to_mut().push_str(", ");
 name.to_mut().push_str(title);
}
```

```
println!("Greetings, {}!", name);
```

Cela peut produire une sortie comme suit :

```
$ cargo run
Greetings, jimb, Esq.!
$
```

Ce qui est bien ici, c'est que, si renvoie une chaîne statique et renvoie , le porte simplement la chaîne statique jusqu'au . Vous avez réussi à reporter l'allocation à moins que ce ne soit vraiment nécessaire, tout en écrivant du code simple. `get_name()` `get_title` `None Cow println!`

Étant donné qu'elle est fréquemment utilisée pour les chaînes, la bibliothèque standard dispose d'un support spécial pour . Il fournit et convertit à la fois et , afin que vous puissiez écrire de manière plus laconique: `Cow Cow<'a, str> From Into String &str get_name`

```
fn get_name() -> Cow<'static, str> {
 std::env::var("USER")
 .map(|v| v.into())
 .unwrap_or("whoever you are".into())
}
```

`Cow<'a, str>` implémente également et , donc pour ajouter le titre au nom, vous pouvez écrire: `std::ops::Add std::ops::AddAssign`

```
if let Some(title) = get_title() {
 name += ", ";
 name += title;
}
```

Ou, puisque a peut être la destination d'une macro : `String write!`

```
use std::fmt::Write;

if let Some(title) = get_title() {
 write!(name.to_mut(), ", {}", title).unwrap();
}
```

Comme précédemment, aucune allocation ne se produit tant que vous n'avez pas essayé de modifier le fichier .Cow

Gardez à l'esprit que tout ne doit pas être : vous pouvez utiliser pour emprunter du texte préalablement calculé jusqu'au moment où une copie devient nécessaire. `Cow<..., str> 'static Cow`

## Chaînes en tant que collections génériques

`String` implémente les deux et : renvoie une chaîne vide et peut ajouter des caractères, des tranches de chaîne, des `s` ou des chaînes à la fin d'une chaîne. Il s'agit de la même combinaison de traits implementés par les autres types de collection de Rust comme et pour les modèles de construction génériques tels que et

```
.std::default::Default std::iter::Extend default extend Cow
<..., str> Vec HashMap collect partition
```

Le type implémente également , renvoyant une tranche vide. C'est pratique dans certains cas de coin; par exemple, il vous permet de dériver pour des structures contenant des tranches de chaîne. `&str Default Default`

## Mise en forme des valeurs

Tout au long du livre, nous avons utilisé des macros de mise en forme de texte comme : `println!`

```
println!(" {:.3}\u00b5s: relocated {} at {:#x} to {:#x}, {} bytes",
 0.84391, "object",
 140737488346304_usize, 6299664_usize, 64);
```

Cet appel produit la sortie suivante :

```
0.844\u00b5s: relocated object at 0x7fffffffdcc0 to 0x602010, 64 bytes
```

Le littéral de chaîne sert de modèle pour la sortie : chacun dans le modèle est remplacé par la forme formatée de l'un des arguments suivants. La chaîne de modèle doit être une constante afin que Rust puisse la comparer aux types d'arguments au moment de la compilation. Chaque argument doit être utilisé ; Rust signale une erreur de compilation dans le cas contraire. { ... }

Plusieurs fonctionnalités de bibliothèque standard partagent ce petit langage pour la mise en forme des chaînes :

- La macro l'utilise pour construire `s`. `format!` `String`
- Les macros et les macros écrivent du texte formaté dans le flux de sortie standard. `println!` `print!`
- Les macros et l'écrivent dans un flux de sortie désigné. `writeln!` `write!`
- La macro l'utilise pour construire une expression (idéalement informative) de la consternation terminale. `panic!`

Les installations de formatage de Rust sont conçues pour être ouvertes.

Vous pouvez étendre ces macros pour prendre en charge vos propres types en implémentant les traits de mise en forme du module. Et vous pouvez utiliser la macro et le type pour créer vos propres fonctions et macros prenant en charge le langage de

`formatage`. `std::fmt::format_args!` `std::fmt::Arguments`

Les macros de mise en forme empruntent toujours des références partagées à leurs arguments ; ils ne s'en approprient jamais et ne les transforment jamais.

Les formulaires du modèle sont *appelés paramètres de format* et ont le formulaire . Les deux parties sont facultatives; est fréquemment utilisé.

`{...} {which:how} {}`

La valeur *qui* sélectionne l'argument suivant le modèle doit prendre la place du paramètre. Vous pouvez sélectionner des arguments par index ou par nom. Les paramètres *sans valeur* sont simplement associés à des arguments de gauche à droite.

La valeur *how* indique comment l'argument doit être mis en forme : combien de remplissage, à quelle précision, dans quel rayon numérique, etc. Si *comment* est présent, le colon avant qu'il ne soit nécessaire. [Le tableau 17-4](#) présente quelques exemples.

Tableau 17-4. Exemples de chaînes mises en forme

| Chaîne de modèle                 | Liste d'arguments                  | Résultat                       |
|----------------------------------|------------------------------------|--------------------------------|
| "number of {}:<br>{}"            | "elephants", 19                    | "number of elephants: 19"      |
| "from {1} to<br>{0}"             | "the grave", "the cradle"          | "from the cradle to the grave" |
| "v = {::?}"                      | vec![0,1,2,5,12,<br>29]            | "v = [0, 1, 2,<br>5, 12, 29]"  |
| "name = {::?}"                   | "Nemo"                             | "name = \"Nemo<br>\n"          |
| "{:8.2} km/s"                    | 11.186                             | " 11.19 km/s"                  |
| "{:20} {:02x}<br>{:02x}"         | "adc #42", 105,<br>42              | "adc #42<br>69 2a"             |
| "{1:02x} {2:02<br>x} {0}"        | "adc #42", 105,<br>42              | "69 2a adc #4<br>2"            |
| "{lsb:02x} {ms<br>b:02x} {insn}" | insn="adc #42",<br>lsb=105, msb=42 | "69 2a adc #4<br>2"            |
| "{:02?}"                         | [110, 11, 9]                       | "[110, 11, 09]"                |
| "{:02x?}"                        | [110, 11, 9]                       | "[6e, 0b, 09]"                 |

Si vous souhaitez inclure ou des caractères dans votre sortie, doublez les caractères du modèle : { }

```
assert_eq!(format!("{{a, c}} ⊂ {{a, b, c}}"),
 "{a, c} ⊂ {a, b, c}");
```

## Mise en forme des valeurs de texte

Lors de la mise en forme d'un type textuel comme ou ( est traité comme une chaîne à un seul caractère), la valeur *how* d'un paramètre comporte

plusieurs parties, toutes facultatives : `&str String char`

- Limite de longueur de texte. Rust tronque votre argument s'il est plus long que cela. Si vous ne spécifiez aucune limite, Rust utilise le texte intégral.
- Largeur de champ minimale. Après toute troncature, si votre argument est plus court que cela, Rust le tamponne à droite (par défaut) avec des espaces (par défaut) pour créer un champ de cette largeur. S'il est omis, Rust ne répond pas à votre argumentation.
- Un alignement. Si votre argument doit être rembourré pour respecter la largeur minimale du champ, cela indique où votre texte doit être placé dans le champ. , et placez votre texte au début, au milieu et à la fin, respectivement. `< ^ >`
- Caractère de remplissage à utiliser dans ce processus de remplissage. S'il est omis, Rust utilise des espaces. Si vous spécifiez le caractère de remplissage, vous devez également spécifier l'alignement.

[Le tableau 17-5](#) illustre quelques exemples montrant comment écrire les choses et leurs effets. Tous utilisent le même argument à huit caractères, . "bookends"

Tableau 17-5. Mettre en forme des directives de chaîne pour le texte

| Fonctionnalités utilisées                  | Chaîne de modèle | Résultat           |
|--------------------------------------------|------------------|--------------------|
| Faire défaut                               | "{ }"            | "bookends"         |
| Largeur minimale du champ                  | "{ : 4 }"        | "bookends"         |
|                                            | "{ : 12 }"       | "bookends"<br>"    |
| Limite de longueur de texte                | "{ : . 4 }"      | "book"             |
|                                            | "{ : . 12 }"     | "bookends"         |
| Largeur du champ, limite de longueur       | "{ : 12 . 20 }"  | "bookends"<br>"    |
|                                            | "{ : 4 . 20 }"   | "bookends"         |
|                                            | "{ : 4 . 6 }"    | "booken"           |
|                                            | "{ : 6 . 4 }"    | "book"             |
| Aligné à gauche, largeur                   | "{ : <12 }"      | "bookends"<br>"    |
| Centré, largeur                            | "{ : ^12 }"      | " bookends<br>"    |
| Aligné à droite, largeur                   | "{ : >12 }"      | " booken<br>ds"    |
| Pad avec , centré, largeur '='             | "{ : =^12 }"     | "==bookends<br>==" |
| Pad , aligné à droite, largeur, limite '*' | "{ : *>12 . 4 }" | "*****bo<br>ok"    |

Le formateur de Rust a une compréhension naïve de la largeur: il suppose que chaque caractère occupe une colonne, sans égard pour combiner des caractères, des katakana demi-largeur, des espaces de largeur nulle ou les autres réalités désordonnées d'Unicode. Par exemple:

```
assert_eq!(format!("{:4}", "th\u{e9}"), "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

Bien qu'Unicode indique que ces chaînes sont toutes deux équivalentes à , le formateur de Rust ne sait pas que des caractères comme , COMBINING ACUTE ACCENT, ont besoin d'un traitement spécial. Il tamponne correctement la première chaîne, mais suppose que la seconde a quatre colonnes de large et n'ajoute aucun rembourrage. Bien qu'il soit facile de voir comment Rust pourrait s'améliorer dans ce cas spécifique, la véritable mise en forme de texte multilingue pour tous les scripts Unicode est une tâche monumentale, mieux gérée en s'appuyant sur les boîtes à outils de l'interface utilisateur de votre plate-forme, ou peut-être en générant HTML et CSS et en faisant en sorte qu'un navigateur Web règle tout. Il y a une caisse populaire, , qui gère certains aspects de cela. "thé" '\u{301}' unicode-width

Avec et , vous pouvez également passer des types de pointeurs intelligents de macros de mise en forme avec des référents textuels, comme ou , sans cérémonie. &str String Rc<String> Cow<'a, str>

Étant donné que les chemins de noms de fichiers ne sont pas nécessairement utf-8 bien formés, ce n'est pas tout à fait un type textuel; vous ne pouvez pas passer un macro directement à une macro de mise en forme. Cependant, la méthode d'a renvoie une valeur que vous pouvez mettre en forme et qui règle les choses d'une manière adaptée à la plate-forme : std::path::Path std::path::Path Path display

```
println!("processing file: {}", path.display());
```

## Mise en forme des nombres

Lorsque l'argument de mise en forme a un type numérique comme ou , la valeur du paramètre comporte les parties suivantes, toutes facultatives : usize f64

- Un *rembourrage* et un *alignement*, qui fonctionnent comme ils le font avec les types textuels.
- Un caractère, demandant que le signe du numéro soit toujours affiché, même lorsque l'argument est positif. +
- Caractère demandant un préfixe radix explicite comme ou . Voir la puce « notation » qui conclut cette liste. # 0x 0b
- Caractère demandant que la largeur minimale du champ soit satisfaite en incluant des zéros de début dans le nombre, au lieu de l'approche de remplissage habituelle. 0
- Largeur *de champ minimale*. Si le nombre formaté n'est pas au moins aussi large, Rust le tamponne à gauche (par défaut) avec des espaces (par défaut) pour créer un champ de la largeur donnée.
- *Précision* pour les arguments à virgule flottante, indiquant le nombre de chiffres que Rust doit inclure après la virgule. La rouille arrondit ou s'étend à zéro selon les besoins pour produire exactement autant de chiffres fractionnaires. Si la précision est omise, Rust essaie de représenter avec précision la valeur en utilisant le moins de chiffres possible. Pour les arguments de type entier, la précision est ignorée.
- Une *notation*. Pour les types entiers, cela peut être pour binaire, pour octal, ou pour hexadécimal avec des lettres minuscules ou majuscules. Si vous avez inclus le caractère, il s'agit d'un préfixe radix explicite de style Rust, , , ou . Pour les types à virgule flottante, un radix de ou demande une notation scientifique, avec un coefficient normalisé, en utilisant ou pour l'exposant. Si vous ne spécifiez aucune notation, Rust met en forme les nombres en décimal. b o x x # 0b 0o 0x 0X e E e E

[Le tableau 17-6 montre](#) quelques exemples de mise en forme de la valeur . i32 1234

Tableau 17-6. Mettre en forme des directives de chaîne pour les entiers

| Fonctionnalités utilisées       | Chaîne de modèle | Résultat        |
|---------------------------------|------------------|-----------------|
| Faire défaut                    | "{ }"            | "1234"          |
| Signe forcé                     | "{:+}"           | "+1234"         |
| Largeur minimale du champ       | "{:12}"          | "1234"          |
|                                 | "{:2}"           | "1234"          |
| Signe, largeur                  | "{:+12}"         | "1234"          |
| Zéros de début, largeur         | "{:012}"         | "000000001234"  |
| Signe, zéros, largeur           | "{:+012}"        | "+000000001234" |
| Aligné à gauche, largeur        | "{:<12}"         | "1234"          |
| Centré, largeur                 | "{:^12}"         | "1234"          |
| Aligné à droite, largeur        | "{:>12}"         | "1234"          |
| Aligné à gauche, signe, largeur | "{:<+12}"        | +1234           |
| Centré, signe, largeur          | "{:^+12}"        | +1234           |
| Aligné à droite, signe, largeur | "{:>+12}"        | +1234"          |

| Fonctionnalités utilisées                            | Chaîne de modèle | Résultat            |
|------------------------------------------------------|------------------|---------------------|
| Rembourré avec , centré, largeur '= '                | "{:=^12}"        | "====1234<br>=====" |
| Notation binaire                                     | "{:b}"           | "10011010<br>010"   |
| Largeur, notation octale                             | "{:12o}"         | "<br>2322"          |
| Signe, largeur, notation hexadécimale                | "{:+12x}"        | "<br>+4d2"          |
| Signe, largeur, hexagone avec chiffres majuscules    | "{:+12X}"        | "<br>+4D2"          |
| Signe, préfixe radix explicite, largeur, hexadécimal | "{:+#12x}"       | "<br>+0<br>x4d2"    |
| Signe, radix, zéros, largeur, hexagonal              | "{:+#012x}"      | "+0x00000<br>04d2"  |
|                                                      | "{:+#06x}"       | "+0x4d2"            |

Comme le montrent les deux derniers exemples, la largeur minimale du champ s'applique au nombre entier, au signe, au préfixe radix et à tout.

Les nombres négatifs incluent toujours leur signe. Les résultats sont similaires à ceux présentés dans les exemples de « signes forcés ».

Lorsque vous demandez des zéros de début, les caractères d'alignement et de remplissage sont simplement ignorés, car les zéros développent le nombre pour remplir le champ entier.

En utilisant l'argument , nous pouvons montrer des effets spécifiques aux types à virgule flottante ([tableau 17-7](#)). 1234.5678

Tableau 17-7. Mettre en forme des directives de chaîne pour les nombres à virgule flottante

| Fonctionnalités utilisées          | Chaîne de modèle | Résultat       |
|------------------------------------|------------------|----------------|
| Faire défaut                       | "{}"             | "1234.5678"    |
| Précision                          | "{:.2}"          | "1234.57"      |
|                                    | "{:.6}"          | "1234.567800"  |
| Largeur minimale du champ          | "{:12}"          | "1234.5678"    |
| Minimum, précision                 | "{:12.2}"        | "1234.57"      |
|                                    | "{:12.6}"        | "1234.567800"  |
| Zéros de début, minimum, précision | "{:012.6}"       | "01234.567800" |
| Scientifique                       | "{:e}"           | "1.2345678e3"  |
| Scientifique, précision            | "{:.3e}"         | "1.235e3"      |
| Scientifique, minimum, précision   | "{:12.3e}"       | "1.235e3"      |
|                                    | "{:12.3E}"       | "1.235E3"      |

## Mise en forme d'autres types

Au-delà des chaînes et des nombres, vous pouvez mettre en forme plusieurs autres types de bibliothèque standard :

- Les types d'erreur peuvent tous être formatés directement, ce qui facilite leur inclusion dans les messages d'erreur. Chaque type d'erreur doit implémenter le trait, ce qui étend le trait de mise en forme par défaut . En conséquence, tout type qui implémente est prêt à formater. `std::error::Error std::fmt::Display Error`
- Vous pouvez formater des types d'adresses de protocole Internet comme et . `std::net::IpAddr std::net::SocketAddr`
- Le booléen et les valeurs peuvent être mis en forme, bien que ce ne soient généralement pas les meilleures chaînes à présenter directement aux utilisateurs finaux. `true false`

Vous devez utiliser les mêmes types de paramètres de format que pour les chaînes. Les contrôles de limite de longueur, de largeur de champ et d'alignement fonctionnent comme prévu.

## Mise en forme des valeurs pour le débogage

Pour faciliter le débogage et la journalisation, le paramètre formate tout type public dans la bibliothèque standard Rust d'une manière destinée à être utile aux programmeurs. Vous pouvez l'utiliser pour inspecter des vecteurs, des tranches, des tuples, des tables de hachage, des threads et des centaines d'autres types. `{ :? }`

Par exemple, vous pouvez écrire ce qui suit :

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606, -122.6819273));
map.insert("Taipei", (25.0375167, 121.5637));
println!("{:?}", map);
```

Cette impression est :

```
{"Taipei": (25.0375167, 121.5637), "Portland": (45.5237606, -122.6819273)}
```

Les types savent déjà comment se formater, sans effort de votre part. `HashMap (f64, f64)`

Si vous incluez le caractère dans le paramètre format, Rust imprimera joliment la valeur. Changer ce code pour dire conduit à cette sortie: # `println!("{:#?}", map)`

```

{
 "Taipei": (
 25.0375167,
 121.5637
),
 "Portland": (
 45.5237606,
 -122.6819273
)
}

```

Ces formes exactes ne sont pas garanties et changent parfois d'une version Rust à l'autre.

Le débogage de la mise en forme imprime généralement des nombres en décimal, mais vous pouvez placer un ou avant le point d'interrogation pour demander hexadécimal à la place. La syntaxe de zéro et de largeur de champ est également respectée. Par exemple, vous pouvez écrire :x x

```

println!("ordinary: {:02?}", [9, 15, 240]);
println!("hex: {:02x?}", [9, 15, 240]);

```

Cette impression est :

```

ordinary: [09, 15, 240]
hex: [09, 0f, f0]

```

Comme nous l'avons mentionné, vous pouvez utiliser la syntaxe pour faire fonctionner vos propres types avec #[derive(Debug)] {::?}

```

#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im: f64 }

```

Avec cette définition en place, nous pouvons utiliser un format pour imprimer des valeurs :{::?} Complex

```

let third = Complex { re: -0.5, im: f64::sqrt(0.75) };
println!("{:?}", third);

```

Cette impression est :

```

Complex { re: -0.5, im: 0.8660254037844386 }

```

C'est bien pour le débogage, mais ce serait peut-être bien si vous pouviez les imprimer sous une forme plus traditionnelle, comme . Dans [« For-mater vos propres types »](#), nous allons montrer comment faire exactement cela. {} -0.5 + 0.8660254037844386i

## Mise en forme des pointeurs pour le débogage

Normalement, si vous passez n'importe quel type de pointeur à une macro de mise en forme (une référence, un , un ), la macro suit simplement le pointeur et met en forme son référent ; le pointeur lui-même n'est pas intéressant. Mais lorsque vous déboguez, il est parfois utile de voir le pointeur : une adresse peut servir de « nom » approximatif pour une valeur individuelle, ce qui peut être éclairant lors de l'examen de structures avec des cycles ou du partage. Box Rc

La notation met en forme les références, les zones et d'autres types de type pointeur en tant qu'adresses : { :p }

```
use std::rc::Rc;
let original = Rc::new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("text: {}, {}, {}", original, cloned, impostor);
println!("pointers: {:p}, {:p}, {:p}", original, cloned, impostor);
```

Ce code imprime :

```
text: mazurka, mazurka, mazurka
pointers: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

Bien sûr, les valeurs de pointeur spécifiques varient d'une exécution à l'autre, mais même ainsi, la comparaison des adresses indique clairement que les deux premières sont des références à la même , tandis que la troisième pointe vers une valeur distincte. String

Les adresses ont tendance à ressembler à de la soupe hexadécimale, donc des visualisations plus raffinées peuvent en valoir la peine, mais le style peut toujours être une solution efficace rapide et sale. { :p }

## Référence aux arguments par index ou par nom

Un paramètre de format peut sélectionner explicitement l'argument qu'il utilise. Par exemple:

```
assert_eq!(format!("{}{},{}{}", "zeroth", "first", "second"),
 "first,zeroth,second");
```

Vous pouvez inclure des paramètres de format après deux points :

```
assert_eq!(format!("{}:{}{:b},{}{:>10}", "first", 10, 100),
 "0x0064,1010,=====first");
```

Vous pouvez également sélectionner des arguments par nom. Cela rend les modèles complexes avec de nombreux paramètres beaucoup plus lisibles. Par exemple:

```
assert_eq!(format!("{}{}{:2} @ {}{}{:2}", price:5.2,
 price=3.25,
 quantity=3,
 description="Maple Turmeric Latte"),
 "Maple Turmeric Latte..... 3 @ 3.25");
```

(Les arguments nommés ici ressemblent à des arguments de mots-clés en Python, mais il ne s'agit que d'une caractéristique spéciale des macros de mise en forme, qui ne fait pas partie de la syntaxe d'appel de fonction de Rust.)

Vous pouvez mélanger des paramètres indexés, nommés et positionnels (c'est-à-dire sans index ni nom) en une seule utilisation de macro de mise en forme. Les paramètres positionnels sont associés à des arguments de gauche à droite comme si les paramètres indexés et nommés n'étaient pas là :

```
assert_eq!(format!("{} {} {} {}", "people", "eater", "purple", mode="flying"),
 "flying purple people eater");
```

Les arguments nommés doivent apparaître à la fin de la liste.

## Largeurs et précisions dynamiques

La largeur minimale de champ, la limite de longueur de texte et la précision numérique d'un paramètre ne doivent pas toujours être des valeurs

fixes ; vous pouvez les choisir au moment de l'exécution.

Nous avons examiné des cas comme cette expression, qui vous donne la chaîne justifiée à droite dans un champ de 20 caractères de large

```
:content
```

```
format!("{:>20}", content)
```

Mais si vous souhaitez choisir la largeur du champ au moment de l'exécution, vous pouvez écrire :

```
format!("{:>1$}", content, get_width())
```

L'écriture pour la largeur minimale du champ indique d'utiliser la valeur du deuxième argument comme largeur. L'argument cité doit être un fichier . Vous pouvez également faire référence à l'argument par son nom

```
:1$ format! usize
```

```
format!("{:>width$}", content, width=get_width())
```

La même approche fonctionne également pour la limite de longueur du texte :

```
format!("{:>width$.limit$}", content,
 width=get_width(), limit=get_limit())
```

Au lieu de la limite de longueur de texte ou de la précision en virgule flottante, vous pouvez également écrire , ce qui indique de prendre l'argument positionnel suivant comme précision. Les clips suivants à au plus des caractères : \* content get\_limit()

```
format!("{:.*}", get_limit(), content)
```

L'argument pris comme précision doit être un . Il n'existe pas de syntaxe correspondante pour la largeur du champ. usize

## Formatage de vos propres types

Les macros de mise en forme utilisent un ensemble de traits définis dans le module pour convertir les valeurs en texte. Vous pouvez faire en sorte

que les macros de formatage de Rust formatent vos propres types en implémentant vous-même un ou plusieurs de ces traits. `std::fmt`

La notation d'un paramètre de format indique le trait que le type de son argument doit implémenter, comme illustré dans [le tableau 17-8](#).

Tableau 17-8. Notation de directive de chaîne de format

| Notation | Exemple       | Trait                      | But                                                  |
|----------|---------------|----------------------------|------------------------------------------------------|
| aucun    | { }           | std::fm<br>t::Display<br>y | Texte, chiffres, erreurs : le trait fourre-tout      |
| b        | {bits:<br>#b} | std::fm<br>t::Binary       | Nombres en binaire                                   |
| o        | {:#5o}        | std::fm<br>t::Octal        | Nombres en octal                                     |
| x        | {:4x}         | std::fm<br>t::LowerH<br>ex | Nombres en chiffres hexadécimaux en minuscules       |
| X        | {:016<br>X}   | std::fm<br>t::UpperH<br>ex | Nombres en chiffres hexadécimaux, majuscules         |
| e        | {:.3e}        | std::fm<br>t::LowerE<br>xp | Nombres à virgule flottante en notation scientifique |
| E        | {:.3E}        | std::fm<br>t::UpperE<br>xp | Idem, majuscule E                                    |
| ?        | {:#?}         | std::fm<br>t::Debug        | Vue de débogage, pour les développeurs               |
| p        | {:p}          | std::fm<br>t::Pointe<br>r  | Pointeur comme adresse, pour les développeurs        |

Lorsque vous placez l'attribut sur une définition de type afin de pouvoir utiliser le paramètre format, vous demandez simplement à Rust d'implémenter le trait pour vous. `#[derive(Debug)] {:#?} std::fmt::Debug`

Les traits de mise en forme ont tous la même structure, ne différant que par leurs noms. Nous utiliserons en tant que représentant

```
: std::fmt::Display
```

```
trait Display {
 fn fmt(&self, dest: &mut std::fmt::Formatter)
 -> std::fmt::Result;
}
```

Le travail de la méthode consiste à produire une représentation correctement formatée et à écrire ses caractères sur . En plus de servir de flux de sortie, l'argument contient également des détails analysés à partir du paramètre de format, tels que l'alignement et la largeur minimale du champ. `fmt self dest dest`

Par exemple, plus haut dans ce chapitre, nous avons suggéré qu'il serait bien que les valeurs s'impriment elles-mêmes sous la forme habituelle. Voici une implémentation qui fait cela : `Complex a + bi Display`

```
use std::fmt;

impl fmt::Display for Complex {
 fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
 let im_sign = if self.im < 0.0 { '-' } else { '+' };
 write!(dest, "{} {} {}i", self.re, im_sign, f64::abs(self.im))
 }
}
```

Cela tire parti du fait qu'il s'agit lui-même d'un flux de sortie, de sorte que la macro peut faire la majeure partie du travail pour nous. Avec cette implémentation en place, nous pouvons écrire ce qui suit: `Formatter write!`

```
let one_twenty = Complex { re: -0.5, im: 0.866 };
assert_eq!(format!("{}", one_twenty),
 "-0.5 + 0.866i");

let two_forty = Complex { re: -0.5, im: -0.866 };
assert_eq!(format!("{}", two_forty),
 "-0.5 - 0.866i");
```

Il est parfois utile d'afficher des nombres complexes sous forme polaire : si vous imaginez une ligne tracée sur le plan complexe de l'origine au

nombre, la forme polaire donne la longueur de la ligne et son angle dans le sens des aiguilles d'une montre par rapport à l'axe des x positif. Le caractère d'un paramètre de format sélectionne généralement une autre forme d'affichage ; la mise en œuvre pourrait le traiter comme une demande d'utilisation de la forme polaire :# Display

```
impl fmt::Display for Complex {
 fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
 let (re, im) = (self.re, self.im);
 if dest.alternate() {
 let abs = f64::sqrt(re * re + im * im);
 let angle = f64::atan2(im, re) / std::f64::consts::PI * 180;
 write!(dest, "{} °", abs, angle)
 } else {
 let im_sign = if im < 0.0 { '-' } else { '+' };
 write!(dest, "{} {} {}i", re, im_sign, f64::abs(im))
 }
 }
}
```

À l'aide de cette implémentation :

```
let ninety = Complex { re: 0.0, im: 2.0 };
assert_eq!(format!("{}", ninety),
 "0 + 2i");
assert_eq!(format!("{:#}", ninety),
 "2 ↞ 90°");
```

Bien que les méthodes des traits de mise en forme renvoient une valeur (un type typique spécifique au module), vous ne devez propager les échecs qu'à partir des opérations sur le , comme le fait l'implémentation avec ses appels à ; vos fonctions de formatage ne doivent jamais générer d'erreurs elles-mêmes. Cela permet aux macros de renvoyer simplement un au lieu d'un , car l'ajout du texte formaté à un n'échoue jamais. Il garantit également que toutes les erreurs que vous obtenez ou reflètent des problèmes réels du flux d'E/S sous-jacent, et non des problèmes de formatage.

```
fmt fmt::Result Result Formatter fmt::Display write!
 format! String Result<String,
 ...> String write! writeln!
```

Formatter a beaucoup d'autres méthodes utiles, y compris certaines pour gérer les données structurées comme les cartes, les listes, etc., que

nous ne couvrirons pas ici; consultez la documentation en ligne pour tous les détails.

## Utilisation du langage de mise en forme dans votre propre code

Vous pouvez écrire vos propres fonctions et macros qui acceptent les modèles de format et les arguments en utilisant la macro de Rust et le type. Par exemple, supposons que votre programme doit consigner les messages d'état au fur et à mesure de son exécution et que vous souhaitez utiliser le langage de formatage de texte de Rust pour les produire. Ce qui suit serait un début: `format_args! std::fmt::Arguments`

```
fn logging_enabled() -> bool { ... }

use std::fs::OpenOptions;
use std::io::Write;

fn write_log_entry(entry: std::fmt::Arguments) {
 if logging_enabled() {
 // Keep things simple for now, and just
 // open the file every time.
 let mut log_file = OpenOptions::new()
 .append(true)
 .create(true)
 .open("log-file-name")
 .expect("failed to open log file");

 log_file.write_fmt(entry)
 .expect("failed to write to log");
 }
}
```

Vous pouvez appeler comme ça: `write_log_entry`

```
write_log_entry(format_args!("Hark! {}\\n", mysterious_value));
```

Au moment de la compilation, la macro analyse la chaîne de modèle et la compare aux types des arguments, signalant une erreur en cas de problème. Au moment de l'exécution, il évalue les arguments et crée une valeur contenant toutes les informations nécessaires à la mise en forme du texte : une forme prédéfinie du modèle, ainsi que des références partagées aux valeurs de l'argument. `format_args! Arguments`

Construire une valeur est bon marché: c'est juste rassembler quelques conseils. Aucun travail de formatage n'a encore lieu, seulement la collecte des informations nécessaires pour le faire plus tard. Cela peut être important : si la journalisation n'est pas activée, tout temps passé à convertir des nombres en décimales, à remplir des valeurs, etc. serait gaspillé. Arguments

Le type implémente le trait, dont la méthode prend un et effectue la mise en forme. Il écrit les résultats dans le flux sous-jacent.

```
File std::io::Write write_fmt Argument
```

Cet appel à n'est pas joli. C'est là qu'une macro peut vous aider

```
:write_log_entry
```

```
macro_rules! log { // no ! needed after name in macro definitions
 ($format:tt, $($arg:expr),*) => (
 write_log_entry(format_args!($format, $($arg),*))
)
}
```

Nous couvrons les macros en détail au [chapitre 21](#). Pour l'instant, croyez que cela définit une nouvelle macro qui transmet ses arguments à et appelle ensuite votre fonction sur la valeur résultante. Les macros de formatage comme , et sont toutes à peu près la même

```
idée. log! format_args! write_log_entry Arguments println! writeln! format!
```

Vous pouvez utiliser comme suit: log!

```
log!("O day and night, but this is wondrous strange! {:?}\n",
 mysterious_value);
```

Idéalement, cela a l'air un peu mieux.

## Expressions régulières

La caisse externe est la bibliothèque d'expressions régulières officielle de Rust. Il fournit les fonctions habituelles de recherche et de correspondance. Il a un bon support pour Unicode, mais il peut également rechercher des chaînes d'octets. Bien qu'il ne prenne pas en charge certaines fonctionnalités que vous trouverez souvent dans d'autres packages d'expression régulière, comme les backreferences et les modèles de

recherche, ces simplifications permettent de s'assurer que les recherches prennent du temps linéaire dans la taille de l'expression et dans la longueur du texte recherché. Ces garanties, entre autres, rendent l'utilisation sûre même avec des expressions non fiables recherchant du texte non fiable. `regex regex regex`

Dans ce livre, nous ne fournirons qu'un aperçu de ; vous devriez consulter sa documentation en ligne pour plus de détails. `regex`

Bien que la caisse ne soit pas dans , elle est maintenue par l'équipe de la bibliothèque Rust, le même groupe responsable de . Pour utiliser , placez la ligne suivante dans la section du fichier *Cargo.toml* de votre caisse

```
: regex std std regex [dependencies]
```

```
regex = "1"
```

Dans les sections suivantes, nous supposerons que vous avez ce changement en place.

## Utilisation de base de Regex

Une valeur représente une expression régulière analysée prête à l'emploi. Le constructeur tente d'analyser a en tant qu'expression régulière et renvoie un `:Regex` `Regex::new &str Result`

```
use regex::Regex;

// A semver version number, like 0.2.1.
// May contain a pre-release version suffix, like 0.2.1-alpha.
// (No build metadata suffix, for brevity.)
//
// Note use of r"..." raw string syntax, to avoid backslash blizzard.
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+)([-.[:alnum:]]*)")?;

// Simple search, with a Boolean result.
let haystack = r#"regex = "0.2.5"#;
assert!(semver.is_match(haystack));
```

La méthode recherche la première correspondance dans une chaîne et renvoie une valeur contenant des informations de correspondance pour chaque groupe de l'expression `:Regex::captures regex::Captures`

```
// You can retrieve capture groups:
let captures = semver.captures(haystack)
 .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

L'indexation d'une valeur panique si le groupe demandé ne correspond pas. Pour tester si un groupe particulier correspond, vous pouvez appeler , qui renvoie un fichier . Une valeur enregistre la correspondance d'un seul groupe

```
: Captures Captures::get Option<regex::Match> Match

assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

Vous pouvez itérer sur toutes les correspondances d'une chaîne :

```
let haystack = "In the beginning, there was 1.0.0. \
 For a while, we used 1.0.1-beta, \
 but in the end, we settled on 1.2.4.";

let matches: Vec<&str> = semver.find_iter(haystack)
 .map(|match_| match_.as_str())
 .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

L'itérateur produit une valeur pour chaque correspondance sans chevauchement de l'expression, en travaillant du début à la fin de la chaîne. La méthode est similaire, mais produit des valeurs enregistrant tous les groupes de capture. La recherche est plus lente lorsque les groupes de capture doivent être signalés, donc si vous n'en avez pas besoin, il est préférable d'utiliser l'une des méthodes qui ne les renvoie pas. find\_iter Match captures\_iter Captures

## Construire les valeurs Regex paresseusement

Le constructeur peut être coûteux : la construction d'une expression régulière de 1 200 caractères peut prendre près d'une milliseconde sur

une machine de développement rapide, et même une expression triviale prend des microsecondes. Il est préférable de garder la construction hors des boucles de calcul lourdes; au lieu de cela, vous devriez construire votre une fois, puis réutiliser le même. `Regex::new Regex Regex Regex`

La caisse offre un bon moyen de construire des valeurs statiques par-  
resseusement la première fois qu'elles sont utilisées. Pour commencer,  
notez la dépendance dans votre fichier `Cargo.toml`: `lazy_static`

```
[dependencies]
lazy_static = "1"
```

Cette caisse fournit une macro pour déclarer ces variables :

```
use lazy_static::lazy_static;

lazy_static! {
 static ref SEMVER: Regex
 = Regex::new(r"(\d+)\.(\d+)\.(\d+)([-_.[:alnum:]]*)")?
 .expect("error parsing regex");
}
```

La macro se développe à une déclaration d'une variable statique nommée , mais son type n'est pas exactement . Au lieu de cela, il s'agit d'un type généré par macro qui implémente et expose donc toutes les mêmes méthodes qu'un fichier . La première fois est déréférencée, l'initialiseur est évalué et la valeur est enregistrée pour une utilisation ultérieure.

Puisqu'il s'agit d'une variable statique, et pas seulement d'une variable locale, l'initialiseur s'exécute au plus une fois par exécution de programme. `SEMVER Regex Deref<Target=Regex> Regex SEMVER SEMVER`

Avec cette déclaration en place, l'utilisation est simple: `SEMVER`

```
use std::io::BufRead;

let stdin = std::io::stdin();
for line_result in stdin.lock().lines() {
 let line = line_result?;
 if let Some(match_) = SEMVER.find(&line) {
 println!("{}", match_.as_str());
 }
}
```

Vous pouvez placer la déclaration dans un module, ou même à l'intérieur de la fonction qui utilise le , si c'est l'étendue la plus appropriée. L'expression régulière n'est toujours compilée qu'une seule fois par exécution de programme. `lazy_static! Regex`

## Normalisation

La plupart des utilisateurs considéreraient que le mot Français pour le thé, *thé*, est long de trois caractères. Cependant, Unicode a en fait deux façons de représenter ce texte :

- Dans la forme *composée*, « *thé* » comprend les trois caractères , , et , où est un seul caractère Unicode avec point de code  
`. 't' 'h' 'é' 'é' 0xe9`
- Dans la forme *décomposée*, « *thé* » comprend les quatre caractères , , , et , où le est le caractère ASCII simple, sans accent, et le point de code est le caractère « COMBINING ACUTE ACCENT », qui ajoute un accent aigu à n'importe quel caractère qu'il suit. `'t' 'h' 'e' '\u{301}' 'e' 0x301`

Unicode ne considère ni la forme composée ni la forme décomposée de é comme la « bonne » ; elle les considère plutôt comme des représentations équivalentes du même caractère abstrait. Unicode indique que les deux formulaires doivent être affichés de la même manière, et les méthodes de saisie de texte sont autorisées à produire l'un ou l'autre, de sorte que les utilisateurs ne sauront généralement pas quel formulaire ils consultent ou tapent. (Rust vous permet d'utiliser des caractères Unicode directement dans les littéraux de chaîne, de sorte que vous pouvez simplement écrire si vous ne vous souciez pas de l'encodage que vous obtenez. Ici, nous allons utiliser les échappements pour plus de clarté.) "thé" \u

Cependant, considéré comme rouille ou valeurs, et sont complètement distincts. Ils ont des longueurs différentes, se comparent comme inégaux, ont des valeurs de hachage différentes et s'ordonnent différemment par rapport aux autres chaînes : `&str String "th\u{e9}" "the\u{301}"`

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" > "the\u{301}");
```

```
// A Hasher is designed to accumulate the hash of a series of values,
// so hashing just one is a bit clunky.
```

```

use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
 let mut s = DefaultHasher::new();
 t.hash(&mut s);
 s.finish()
}

// These values may change in future Rust releases.
assert_eq!(hash("th\u{e9}"), 0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);

```

De toute évidence, si vous avez l'intention de comparer le texte fourni par l'utilisateur ou de l'utiliser comme clé dans une table de hachage ou un arbre B, vous devrez d'abord mettre chaque chaîne sous une forme canonique.

Heureusement, Unicode spécifie des formulaires *normalisés* pour les chaînes. Chaque fois que deux chaînes doivent être traitées comme équivalentes selon les règles d'Unicode, leurs formes normalisées sont identiques caractère par caractère. Lorsqu'ils sont codés en UTF-8, ils sont identiques octet pour octet. Cela signifie que vous pouvez comparer des chaînes normalisées avec , les utiliser comme clés dans un ou , et ainsi de suite, et vous obtiendrez la notion d'égalité

d'Unicode. == HashMap HashSet

L'échec de la normalisation peut même avoir des conséquences sur la sécurité. Par exemple, si votre site Web normalise les noms d'utilisateur dans certains cas mais pas dans d'autres, vous pourriez vous retrouver avec deux utilisateurs distincts nommés , que certaines parties de votre code traitent comme le même utilisateur, mais d'autres distinguent, ce qui entraîne l'extension incorrecte des privilèges de l'un à l'autre. Bien sûr, il existe de nombreuses façons d'éviter ce genre de problème, mais l'histoire montre qu'il existe également de nombreuses façons de ne pas le faire. bananasflambé

## Formulaires de normalisation

Unicode définit quatre formes normalisées, chacune étant appropriée pour différentes utilisations. Il y a deux questions auxquelles il faut répondre :

- Tout d'abord, préférez-vous que les personnages soient aussi *composés* que possible ou aussi *décomposés* que possible ?

Par exemple, la représentation la plus composée du mot vietnamien *Phở* est la chaîne à trois caractères , où la marque tonale ' et la marque voyelle ' sont appliquées au caractère de base « o » dans un seul caractère Unicode, , qu'Unicode nomme consciencieusement PETITE LETTRE LATINE O AVEC CORNE ET CROCHET CI-DESSUS. "Ph\u{1edf}" '\u{1edf}'

La représentation la plus décomposée divise la lettre de base et ses deux marques en trois caractères Unicode distincts : , (COMBINAISON DE CORNE) et (COMBINAISON DE CROCHET CI-DESSUS), ce qui donne . (Chaque fois que les marques combinées apparaissent en tant que caractères distincts, plutôt que dans le cadre d'un caractère composé, toutes les formes normalisées spécifient un ordre fixe dans lequel elles doivent apparaître, de sorte que la normalisation est bien spécifiée même lorsque les caractères ont plusieurs accents.) 'o' '\u{31b}' '\u{309}' "Pho\u{31b}\u{309}"

La forme composée a généralement moins de problèmes de compatibilité, car elle correspond plus étroitement aux représentations que la plupart des langues utilisaient pour leur texte avant l'établissement d'Unicode. Il peut également mieux fonctionner avec des fonctionnalités de formatage de chaîne naïves comme la macro de Rust. La forme décomposée, en revanche, peut être meilleure pour l'affichage du texte ou la recherche, car elle rend la structure détaillée du texte plus explicite. format !

- La deuxième question est la suivante : si deux séquences de caractères représentent le même texte fondamental mais diffèrent dans la façon dont le texte doit être formaté, voulez-vous les traiter comme équivalentes ou les garder distinctes ?

Unicode a des caractères distincts pour le chiffre ordinaire, le chiffre en exposant (ou ) et le chiffre encerclé (ou ), mais déclare que les trois sont *équivalents* à la compatibilité. De même, Unicode a un seul caractère pour la ligature *ffi* (), mais déclare que c'est une compatibilité équivalente à la séquence à trois caractères

. 5 5 '\u{2075}' ⑤ '\u{2464}' '\u{fb03}' ffi

L'équivalence de compatibilité est logique pour les recherches : une recherche pour , en utilisant uniquement des caractères ASCII, doit correspondre à la chaîne , qui utilise la ligature *ffi*. L'application de la décomposition de compatibilité à cette dernière chaîne remplacerait la

ligature par les trois lettres simples, ce qui faciliterait la recherche.

Mais la normalisation du texte à une forme équivalente à la compatibilité peut perdre des informations essentielles, il ne doit donc pas être appliqué négligemment. Par exemple, il serait incorrect dans la plupart des contextes de stocker en tant que

```
. "difficult" "di\u{fb03}cult" "ffi" "2⁵" "2⁵"
```

Unicode Normalization Form C et Normalization Form D (NFC et NFD) utilisent les formes composées au maximum et décomposées au maximum de chaque caractère, mais n'essayez pas d'unifier des séquences équivalentes de compatibilité. Les formulaires de normalisation NFKC et NFKD sont comme NFC et NFD, mais normalisent toutes les séquences équivalentes de compatibilité à un simple représentant de leur classe.

Le « Character Model For the World Wide Web » du World Wide Web Consortium recommande d'utiliser NFC pour tous les contenus. L'annexe Unicode Identifier and Pattern Syntax suggère d'utiliser NFKC pour les identificateurs dans les langages de programmation et propose des principes pour adapter le formulaire si nécessaire.

## La caisse de normalisation unicode

La caisse de Rust fournit un trait qui ajoute des méthodes pour mettre le texte dans l'une des quatre formes normalisées. Pour l'utiliser, ajoutez la ligne suivante à la section de votre fichier *Cargo.toml*:

```
unicode-normalization &str [dependencies]
```

```
unicode-normalization = "0.1.17"
```

Avec cette déclaration en place, a quatre nouvelles méthodes qui renvoient des itérateurs sur une forme normalisée particulière de la chaîne : `&str`

```
use unicode_normalization::UnicodeNormalization;

// No matter what representation the left-hand string uses
// (you shouldn't be able to tell just by looking),
// these assertions will hold.
assert_eq!("Phő".nfd().collect::<String>(), "Pho\u{31b}\u{309}");
assert_eq!("Phő".nfc().collect::<String>(), "Ph\u{1edf}");
```

```
// The left-hand side here uses the "ffi" ligature character.
assert_eq!("① Di\u{fb03}culty".nfkc().collect::<String>(), "1 Difficu
```

Prendre une chaîne normalisée et la normaliser à nouveau sous la même forme est garanti pour renvoyer un texte identique.

Bien que toute sous-chaîne d'une chaîne normalisée soit elle-même normalisée, la concaténation de deux chaînes normalisées n'est pas nécessairement normalisée : par exemple, la deuxième chaîne peut commencer par combiner des caractères qui doivent être placés avant de combiner des caractères à la fin de la première chaîne.

Tant qu'un texte n'utilise pas de points de code non attribués lorsqu'il est normalisé, Unicode promet que sa forme normalisée ne changera pas dans les futures versions de la norme. Cela signifie que les formulaires normalisés peuvent généralement être utilisés en toute sécurité dans le stockage persistant, même si la norme Unicode évolue.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

# Chapitre 18. Entrées et sorties

*Doolittle: Quelles preuves concrètes avez-vous que vous existez?*

*Bombe #20: Hmm... puits... Je pense, donc je le suis.*

*Doolittle : C'est bien. C'est très bien. Mais comment savez-vous que quelque chose d'autre existe?*

*Bombe #20 : Mon appareil sensoriel me le révèle.*

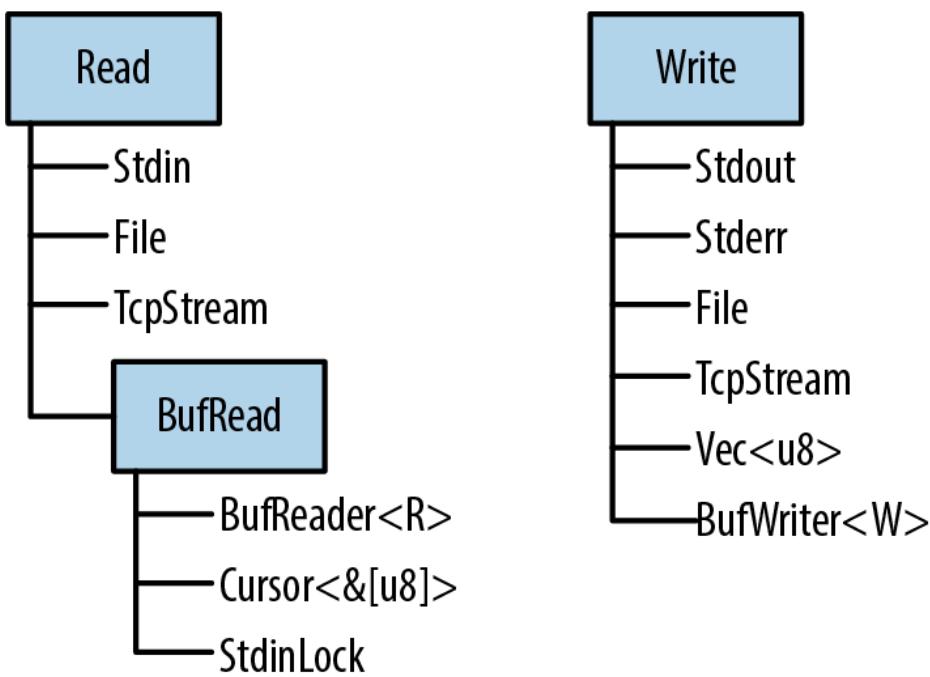
—Étoile noire

Les fonctionnalités de bibliothèque standard de Rust pour l'entrée et la sortie sont organisées autour de trois traits, , , et : `Read BufRead Write`

- Les valeurs implémentées ont des méthodes pour l'entrée orientée octet. On les appelle des *lecteurs*. `Read`
- Les valeurs *implémentées sont des lecteurs mis en mémoire tampon*. Ils prennent en charge toutes les méthodes de , plus les méthodes de lecture de lignes de texte et ainsi de suite. `BufRead Read`
- Les valeurs implémentées prennent en charge à la fois la sortie de texte orientée octet et UTF-8. On les appelle *des écrivains*. `Write`

[La figure 18-1](#) montre ces trois traits et quelques exemples de types de lecteurs et d'écrivains.

Dans ce chapitre, nous expliquerons comment utiliser ces traits et leurs méthodes, couvrirons les types de lecteur et d'écrivain illustrés dans la figure et montrerons d'autres façons d'interagir avec les fichiers, le terminal et le réseau.



Graphique 18-1. Les trois principaux traits d'E/S de Rust et les types sélectionnés qui les implémentent

## Lecteurs et écrivains

*Les lecteurs* sont des valeurs à partir desquelles votre programme peut lire des octets. En voici quelques exemples :

- Fichiers ouverts à l'aide de `std::fs::File::open(filename)`
- `std::net::TcpStreams`, pour recevoir des données sur le réseau
- `std::io::stdin()`, pour la lecture à partir du flux d'entrée standard du processus
- `std::io::Cursor<&[u8]>` et les valeurs, qui sont des lecteurs qui « lisent » à partir d'un tableau d'octets ou d'un vecteur qui est déjà en mémoire `std::io::Cursor<Vec<u8>>`

*Les rédacteurs* sont des valeurs sur lesquelles votre programme peut écrire des octets. En voici quelques exemples :

- Fichiers ouverts à l'aide de `std::fs::File::create(filename)`
- `std::net::TcpStreams`, pour l'envoi de données sur le réseau
- `std::io::stdout()` et , pour écrire au terminal `std::io::stderr()`
- `Vec<u8>`, un écrivain dont les méthodes s'ajoutent au vecteur `write`
- `std::io::Cursor<Vec<u8>>`, qui est similaire mais vous permet à la fois de lire et d'écrire des données, et de chercher à différentes positions dans le vecteur
- `std::io::Cursor<&mut [u8]>`, qui ressemble beaucoup à , sauf qu'il ne peut pas faire croître la mémoire tampon, car il ne s'agit que d'une tranche d'un tableau d'octets existant `std::io::Cursor<Vec<u8>>`

Comme il existe des traits standard pour les lecteurs et les écrivains ( et ), il est assez courant d'écrire du code générique qui fonctionne sur une variété de canaux d'entrée ou de sortie. Par exemple, voici une fonction qui copie tous les octets de n'importe quel lecteur vers n'importe quel écrivain :

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

```
const DEFAULT_BUF_SIZE: usize = 8 * 1024;

pub fn copy<R: ?Sized, W: ?Sized>(reader: &mut R, writer: &mut W)
 -> io::Result<u64>
 where R: Read, W: Write
{
 let mut buf = [0; DEFAULT_BUF_SIZE];
 let mut written = 0;
 loop {
 let len = match reader.read(&mut buf) {
 Ok(0) => return Ok(written),
 Ok(len) => len,
 Err(ref e) if e.kind() == ErrorKind::Interrupted => continue,
 Err(e) => return Err(e),
 };
 writer.write_all(&buf[..len])?;
 written += len as u64;
 }
}
```

Il s'agit de l'implémentation de la bibliothèque standard de Rust. Comme il est générique, vous pouvez l'utiliser pour copier des données de a à a , de à un en mémoire ,

```
etc. std::io::copy() File TcpStream Stdin Vec<u8>
```

Si le code de gestion des erreurs ici n'est pas clair, revenez [au chapitre 7](#). Nous utiliserons le type constamment dans les pages à venir; il est important d'avoir une bonne compréhension de son fonctionnement. Result

Les trois traits , , et , avec , sont si couramment utilisés qu'il existe un module contenant uniquement ces

```
traits: std::io Read BufRead Write Seek prelude
```

```
use std::io::prelude::*;


```

Vous le verrez une ou deux fois dans ce chapitre. Nous prenons également l'habitude d'importer le module lui-même: `std::io`

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

Le mot-clé `self` ici est déclaré comme alias pour le module. De cette façon, et peut être écrit de manière plus concise au fur et à mesure, et ainsi de suite.

```
self io std::io std::io::Result std::io::Error io::Result io::Error
```

## Lecteurs

`std::io::Read` dispose de plusieurs méthodes de lecture des données. Tous prennent le lecteur lui-même par référence. `mut`

```
reader.read(&mut buffer)
```

Lit certains octets de la source de données et les stocke dans le fichier. Le type de l'argument est `.buffer`. Cela se lit jusqu'à des octets.

```
buffer &mut [u8] buffer.len()
```

Le type de retour est `Result<u64>`, qui est un alias de type pour `An`. En cas de réussite, la valeur est le nombre d'octets lus, qui peut être égal ou inférieur à `len`, même s'il y a plus de données à venir, au gré de la source de données. `Ok(0)` signifie qu'il n'y a plus d'entrée à lire.

```
io::Result<u64> Result<u64>,
io::Error> u64 buffer.len() Ok(0)
```

En cas d'erreur, renvoie `Err(err)`, où `err` est une valeur `An` est imprimable, pour le bénéfice des humains; pour les programmes, il dispose d'une méthode qui renvoie un code d'erreur de type `Code`. Les membres de cet enum ont des noms comme `ConnectionReset` et `Interrupted`. La plupart indiquent des erreurs graves qui ne peuvent pas être ignorées, mais un type d'erreur doit être traité spécialement. `PermissionDenied` correspond au code d'erreur Unix `EINTR`, ce qui signifie que la lecture a été interrompue par un signal. À moins que le programme ne soit conçu pour faire quelque chose d'intelligent avec les signaux, il devrait simplement réessayer la lecture. Le code de `copy`, dans la section précédente, en montre un exemple.

```
.read() Err(err) err io::Error io::Error .kind()
) io::ErrorKind PermissionDenied ConnectionReset io::ErrorKind::Interrupted EINTR copy()
```

Comme vous pouvez le voir, la méthode est de très bas niveau, héritant même des bizarries du système d'exploitation sous-jacent. Si vous implémentez le trait pour un nouveau type de source de données, cela vous donne beaucoup de marge de manœuvre. Si vous essayez de lire des données, c'est pénible. Par conséquent, Rust fournit plusieurs méthodes de commodité de niveau supérieur. Tous ont des implémentations par défaut en termes de . Ils gèrent tous , donc vous n'avez pas à le

```
faire. .read() Read .read() ErrorKind::Interrupted
```

```
reader.read_to_end(&mut byte_vec)
```

Lit toutes les entrées restantes de ce lecteur, en l'ajoutant à , qui est un fichier . Renvoie un , le nombre d'octets  
lus. byte\_vec Vec<u8> io::Result<usize>

Il n'y a pas de limite à la quantité de données que cette méthode empilera dans le vecteur, alors ne l'utilisez pas sur une source non fiable. (Vous pouvez imposer une limite à l'aide de la méthode décrite dans la liste suivante.) .take()

```
reader.read_to_string(&mut string)
```

C'est la même chose, mais ajoute les données au fichier . Si le flux n'est pas valide UTF-8, cela renvoie une  
erreur. String ErrorKind::InvalidData

Dans certains langages de programmation, l'entrée d'octets et l'entrée de caractères sont gérées par différents types. De nos jours, UTF-8 est si dominant que Rust reconnaît cette norme de facto et prend en charge UTF-8 partout. D'autres jeux de caractères sont pris en charge avec la caisse open source. encoding

```
reader.read_exact(&mut buf)
```

Lit exactement suffisamment de données pour remplir le tampon donné. Le type d'argument est . Si le lecteur manque de données avant de lire des octets, cela renvoie une erreur. & [ u8 ] buf.len() ErrorKind::UnexpectedEof

Ce sont les principales méthodes du trait. En outre, il existe trois méthodes d'adaptateur qui prennent la valeur by, la transformant en un itérateur ou un lecteur différent : Read reader

```
reader.bytes()
```

Renvoie un itérateur sur les octets du flux d'entrée. Le type d'élément est , de sorte qu'une vérification d'erreur est requise pour chaque octet. De plus, cela appelle une fois par octet, ce qui sera très inefficace si le lecteur n'est pas mis en mémoire tampon. `io::Result<u8> reader.read()`

`reader.chain(reader2)`

Renvoie un nouveau lecteur qui produit toutes les entrées de , suivies de toutes les entrées de . `reader reader2`

`reader.take(n)`

Renvoie un nouveau lecteur qui lit à partir de la même source que , mais qui est limité aux octets d'entrée. `reader n`

Il n'existe aucune méthode pour fermer un lecteur. Les lecteurs et les rédacteurs implémentent généralement de sorte qu'ils sont fermés automatiquement. `Drop`

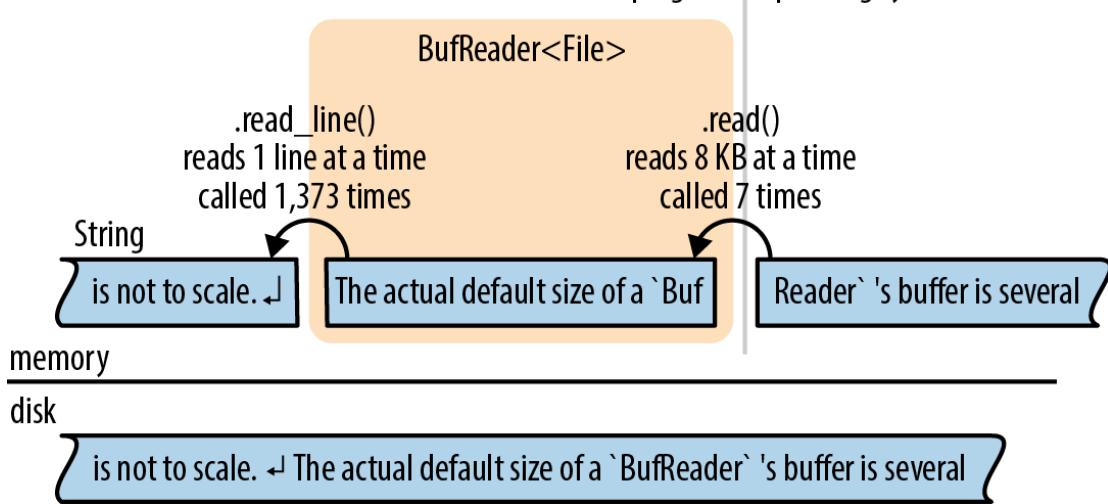
## Lecteurs tamponnés

Pour plus d'efficacité, les lecteurs et les rédacteurs peuvent être *mis en mémoire tampon*, ce qui signifie simplement qu'ils ont un morceau de mémoire (un tampon) qui contient des données d'entrée ou de sortie en mémoire. Cela permet d'économiser sur les appels système, comme illustré à [la figure 18-2](#). L'application lit les données du , dans cet exemple en appelant sa méthode. Le à son tour obtient son entrée en plus gros morceaux du système

d'exploitation. `BufReader .read_line() BufReader`

Cette image n'est pas à l'échelle. La taille par défaut réelle de la mémoire tampon d'un tampon est de plusieurs kilo-octets, de sorte qu'un seul système peut servir des centaines d'appels. Cela est important car les appels système sont lents. `BufReader read .read_line()`

(Comme le montre l'image, le système d'exploitation dispose également d'une mémoire tampon, pour la même raison : les appels système sont lents, mais la lecture des données d'un disque est plus lente.)



Graphique 18-2. Un lecteur de fichiers mis en mémoire tampon

Les lecteurs mis en mémoire tampon implémentent les deux et un deuxième trait, , qui ajoute les méthodes suivantes : `Read BufRead`

```
reader.read_line(&mut line)
```

Lit une ligne de texte et l'ajoute à , qui est un fichier . Le caractère de nouvelle ligne à la fin de la ligne est inclus dans . Si l'entrée a des terminaisons de ligne de style Windows, , les deux caractères sont inclus dans .  
`line String '\n' line "\r\n" line`

La valeur renvoyée est un , le nombre d'octets lus, y compris la fin de la ligne, le cas échéant. `io::Result<usize>`

Si le lecteur est à la fin de l'entrée, cela laisse inchangé et renvoie  
`.line Ok(0)`

```
reader.lines()
```

Renvoie un itérateur sur les lignes de l'entrée. Le type d'élément est . Les caractères de nouvelle ligne *ne sont pas* inclus dans les chaînes. Si l'entrée a des terminaisons de ligne de style Windows, les deux caractères sont supprimés. `io::Result<String> "\r\n"`

Cette méthode est presque toujours ce que vous voulez pour la saisie de texte. Les deux sections suivantes montrent quelques exemples de son utilisation.

```
reader.read_until(stop_byte, &mut byte_vec),
reader.split(stop_byte)
```

Ceux-ci sont comme et , mais orientés octets, produisant s au lieu de s. Vous choisissez le délimiteur  
`..read_line() .lines() Vec<u8> String stop_byte`

`BufRead` fournit également une paire de méthodes de bas niveau et , pour un accès direct à la mémoire tampon interne du lecteur. Pour en savoir plus sur ces méthodes, consultez la documentation en ligne. `.fill_buf()` `.consume(n)`

Les deux sections suivantes couvrent plus en détail les lecteurs tamponnés.

## Lignes de lecture

Voici une fonction qui implémente l'utilitaire Unix. Il recherche de nombreuses lignes de texte, généralement acheminées à partir d'une autre commande, pour une chaîne donnée : `grep`

```
use std::io;
use std::io::prelude::*;

fn grep(target: &str) -> io::Result<()> {
 let stdin = io::stdin();
 for line_result in stdin.lock().lines() {
 let line = line_result?;
 if line.contains(target) {
 println!("{}", line);
 }
 }
 Ok(())
}
```

Puisque nous voulons appeler `lock()`, nous avons besoin d'une source d'entrée qui implémente `BufRead`. Dans ce cas, nous appelons `io::stdin()` pour obtenir les données qui nous sont acheminées. Cependant, la bibliothèque standard Rust protège avec un mutex. Nous appelons `lock()` pour l'usage exclusif du fil actuel; il renvoie une valeur qui implémente `BufRead`. À la fin de la boucle, le mutex est lâché, libérant le mutex. (Sans mutex, deux threads essayant de lire en même temps provoqueraient un comportement indéfini. C a le même problème et le résout de la même manière: toutes les fonctions d'entrée et de sortie standard C obtiennent un verrou dans les coulisses. La seule différence est que dans Rust, le verrou fait partie de

```
l'API.) .lines() BufRead io::stdin() stdin .lock() stdin StdinLock BufRead StdinLock stdin
```

Le reste de la fonction est simple : elle appelle `lock()` et boucle sur l'itérateur résultant. Étant donné que cet itérateur produit des valeurs, nous utilisons

l'opérateur pour vérifier les erreurs. .lines() Result ?

Supposons que nous voulions pousser notre programme un peu plus loin et ajouter la prise en charge de la recherche de fichiers sur le disque.

Nous pouvons rendre cette fonction générique: grep

```
fn grep<R>(target: &str, reader: R) -> io::Result<()>
 where R: BufRead
{
 for line_result in reader.lines() {
 let line = line_result?;
 if line.contains(target) {
 println!("{}", line);
 }
 }
 Ok(())
}
```

Maintenant, nous pouvons le passer soit un ou un tamponné: StdinLock File

```
let stdin = io::stdin();
grep(&target, stdin.lock())?; // ok

let f = File::open(file)?;
grep(&target, BufReader::new(f))?; // also ok
```

Notez que a n'est pas automatiquement mis en mémoire tampon. implémente mais pas . Cependant, il est facile de créer un lecteur tamponné pour un , ou tout autre lecteur sans tampon. fait cela. (Pour définir la taille de la mémoire tampon, utilisez

```
.)File File Read BufRead File BufReader::new(reader) BufRead
er::with_capacity(size, reader)
```

Dans la plupart des langues, les fichiers sont mis en mémoire tampon par défaut. Si vous voulez une entrée ou une sortie sans tampon, vous devez trouver comment désactiver la mise en mémoire tampon. Dans Rust, et sont deux fonctionnalités de bibliothèque distinctes, parce que parfois vous voulez des fichiers sans mise en mémoire tampon, et parfois vous voulez mettre en mémoire tampon sans fichiers (par exemple, vous pouvez vouloir mettre en mémoire tampon l'entrée du réseau). File BufReader

Le programme complet, y compris la gestion des erreurs et l'analyse des arguments bruts, est illustré ici :

```
// grep - Search stdin or some files for lines matching a given string.

use std::error::Error;
use std::io::{self, BufReader};
use std::io::prelude::*;
use std::fs::File;
use std::path::PathBuf;

fn grep<R>(target: &str, reader: R) -> io::Result<()>
 where R: BufRead
{
 for line_result in reader.lines() {
 let line = line_result?;
 if line.contains(target) {
 println!("{}", line);
 }
 }
 Ok(())
}

fn grep_main() -> Result<(), Box
```

```

fn main() {
 let result = grep_main();
 if let Err(err) = result {
 eprintln!("{}: {}", file, err);
 std::process::exit(1);
 }
}

```

## Collecte de lignes

Plusieurs méthodes de lecture, y compris `.lines()`, renvoient des itérateurs qui produisent des valeurs. La première fois que vous souhaitez rassembler toutes les lignes d'un fichier en un seul grand vecteur, vous rencontrerez un problème pour vous débarrasser du type `Result<String>`:

```

// ok, but not what you want
let results: Vec<io::Result<String>> = reader.lines().collect();

// error: can't convert collection of Results to Vec<String>
let lines: Vec<String> = reader.lines().collect();

```

Le deuxième essai ne compile pas : qu'adviendrait-il des erreurs ? La solution simple consiste à écrire une boucle et à vérifier chaque élément pour les erreurs: `for`

```

let mut lines = vec![];
for line_result in reader.lines() {
 lines.push(line_result?);
}

```

Pas mal; mais ce serait bien de l'utiliser ici, et il s'avère que nous le pouvons. Il suffit de savoir quel type demander : `.collect()`

```
let lines = reader.lines().collect::<io::Result<Vec<String>>>();
```

Comment cela fonctionne-t-il ? La bibliothèque standard contient une implémentation de `FromIterator` (facile à négliger dans la documentation en ligne - qui rend cela possible : `FromIterator<Result<T, E>>`)

```

impl<T, E, C> FromIterator<Result<T, E>> for Result<C, E>
 where C: FromIterator<T>
{

```

```
...
}
```

Cela nécessite une lecture attentive, mais c'est une belle astuce. Supposons qu'il s'agisse de n'importe quel type de collection, comme ou . Tant que nous savons déjà comment construire un à partir d'un itérateur de valeurs, nous pouvons construire un à partir d'un itérateur produisant des valeurs. Nous avons juste besoin de tirer des valeurs de l'itérateur et de construire la collection à partir des résultats, mais si jamais nous voyons un , arrêtez-vous et transmettez-

```
le. C Vec HashSet C T Result<C, E> Result<T, E> Ok Err
```

En d'autres termes, est un type de collection, de sorte que la méthode peut créer et remplir des valeurs de ce type. `io::Result<Vec<String>> .collect()`

## Écrivains

Comme nous l'avons vu, la saisie se fait principalement à l'aide de méthodes. La sortie est un peu différente.

Tout au long du livre, nous avons utilisé pour produire une sortie en texte brut: `println!()`

```
println!("Hello, world!");

println!("The greatest common divisor of {:+?} is {}",
 numbers, d);

println!(); // print a blank line
```

Il y a aussi une macro, qui n'ajoute pas de caractère de nouvelle ligne à la fin, et des macros qui écrivent dans le flux d'erreur standard. Les codes de mise en forme pour tous ces éléments sont les mêmes que ceux de la macro, décrits dans [« Valeurs de mise en forme »](#). `print!`

```
() eprintln! eprint! format!
```

Pour envoyer la sortie à un enregistreur, utilisez les macros et. Ils sont les mêmes que et , à l'exception de deux différences : `write!()` `writeln!()` `print!()` `println!()`

```
writeln!(io::stderr(), "error: world not helloable")?;
```

```
writeln!(&mut byte_vec, "The greatest common divisor of {} is {}", numbers, d);
```

Une différence est que les macros prennent chacune un premier argument supplémentaire, un écrivain. L'autre est qu'ils renvoient un , donc les erreurs doivent être gérées. C'est pourquoi nous avons utilisé l'opérateur à la fin de chaque ligne. `write Result ?`

Les macros ne renvoient pas de ; ils paniquent simplement si l'écriture échoue. Comme ils écrivent sur le terminal, c'est rare. `print Result`

Le trait a ces méthodes: `Write`

```
writer.write(&buf)
```

Écrit une partie des octets de la tranche dans le flux sous-jacent. Il renvoie un fichier . En cas de succès, cela donne le nombre d'octets écrits, qui peut être inférieur à , au gré du flux. `buf io::Result<usize> buf.len()`

Comme , il s'agit d'une méthode de bas niveau que vous devriez éviter d'utiliser directement. `Reader::read()`

```
writer.write_all(&buf)
```

Écrit tous les octets de la tranche . Retourne. `buf Result<()>`

```
writer.flush()
```

Vide toutes les données mises en mémoire tampon dans le flux sous-jacent. Retourne. `Result<()>`

Notez que si les macros et vident automatiquement le flux stdout et stderr, les macros et ne le font pas. Vous devrez peut-être appeler manuellement lorsque vous les utilisez. `println! eprintln! print! eprint! flush()`

Comme les lecteurs, les écrivains sont fermés automatiquement lorsqu'ils sont abandonnés.

Tout comme ajoute un tampon à n'importe quel lecteur, ajoute un tampon à n'importe quel

```
écrivain: BufReader::new(reader) BufWriter::new(writer)
```

```
let file = File::create("tmp.txt")?;
let writer = BufWriter::new(file);
```

Pour définir la taille de la mémoire tampon, utilisez

```
.BufWriter::with_capacity(size, writer)
```

Lorsque `a` est supprimé, toutes les données mises en mémoire tampon restantes sont écrites dans l'enregistreur sous-jacent. Toutefois, si une erreur se produit pendant cette écriture, l'erreur est *ignorée*. (Étant donné que cela se produit à l'intérieur de la méthode de `drop()`, il n'y a pas d'endroit utile pour signaler l'erreur.) Pour vous assurer que votre application remarque toutes les erreurs de sortie, mettez manuellement en mémoire tampon les rédacteurs avant de les supprimer.

```
BufWriter BufWriter .drop() .flush()
```

## Fichiers

Nous avons déjà vu deux façons d'ouvrir un fichier :

```
File::open(filename)
```

Ouvre un fichier existant pour lecture. Il renvoie un `File`, et c'est une erreur si le fichier n'existe pas.

```
io::Result<File>
```

```
File::create(filename)
```

Crée un nouveau fichier pour l'écriture. Si un fichier existe avec le nom de fichier donné, il est tronqué.

Notez que le type se trouve dans le module de système de fichiers, et non dans `.File`

```
std::fs std::io
```

Lorsque ni l'un ni l'autre de ces éléments ne correspond à la facture, vous pouvez l'utiliser pour spécifier le comportement souhaité exact

```
:OpenOptions
```

```
use std::fs::OpenOptions;
```

```
let log = OpenOptions::new()
 .append(true) // if file exists, add to the end
 .open("server.log")?;
```

```
let file = OpenOptions::new()
 .write(true)
 .create_new(true) // fail if file exists
 .open("new_file.txt")?;
```

Les méthodes `append`, `create_new`, et ainsi de suite sont conçues pour être enchaînées comme ceci: chacune renvoie `OpenOptions`. Ce modèle de conception de chaînage de

méthode est assez commun pour avoir un nom dans Rust: il s'appelle un *constructeur*. est un autre exemple. Pour plus de détails sur , consultez la documentation en

```
ligne. .append() .write() .create_new() self std::process::Command OpenOptions
```

Une fois qu'un a été ouvert, il se comporte comme n'importe quel autre lecteur ou écrivain. Vous pouvez ajouter un tampon si nécessaire. Le sera fermé automatiquement lorsque vous le déposerez. `File`

## Recherche

`File` implémente également le trait, ce qui signifie que vous pouvez sauter dans un plutôt que de lire ou d'écrire en un seul passage du début à la fin. est défini comme suit : `Seek`

```
pub trait Seek {
 fn seek(&mut self, pos: SeekFrom) -> io::Result<u64>;
}

pub enum SeekFrom {
 Start(u64),
 End(i64),
 Current(i64)
}
```

Grâce à l'enum, la méthode est joliment expressive : utiliser pour rebobiner au début et utiliser pour revenir en arrière de quelques octets, et ainsi de

```
suite. seek file.seek(SeekFrom::Start(0)) file.seek(SeekFrom::Current(-8))
```

La recherche dans un fichier est lente. Que vous utilisiez un disque dur ou un disque SSD, une recherche prend autant de temps que la lecture de plusieurs mégaoctets de données.

## Autres types de lecteurs et d'écrivains

Jusqu'à présent, ce chapitre a utilisé comme exemple un cheval de bataille, mais il existe de nombreux autres types de lecteurs et d'écrivains utiles: `File`

```
io::stdin()
```

Renvoie un lecteur pour le flux d'entrée standard. Son type est . Comme cela est partagé par tous les threads, chaque lecture acquiert et libère un mutex. `io:::Stdin`

`Stdin` possède une méthode qui acquiert le mutex et renvoie un , un lecteur tamponné qui maintient le mutex jusqu'à ce qu'il soit abandonné. Les opérations individuelles sur le évitent donc la surcharge mutex. Nous avons montré un exemple de code utilisant cette méthode dans [« Reading Lines »](#).

```
lock() io:::StdinLock StdinLock
```

Pour des raisons techniques, ne fonctionne pas. Le verrou contient une référence à la valeur, ce qui signifie que la valeur doit être stockée quelque part pour qu'elle vive assez longtemps

```
: io:::stdin().lock() Stdin Stdin
```

```
let stdin = io:::stdin();
let lines = stdin.lock().lines(); // ok
```

```
io:::stdout(), io:::stderr()
```

Types de retour et d'écriture pour les flux de sortie standard et d'erreur standard. Ceux-ci aussi ont des mutex et des méthodes. `Stdout Stderr .lock()`

```
Vec<u8>
```

Implémente. L'écriture sur un étend le vecteur avec les nouvelles données. `write Vec<u8>`

( `String` , cependant, n'implémente pas . Pour créer une chaîne à l'aide de , écrivez d'abord dans un , puis utilisez pour convertir le vecteur en

```
chaîne.) write Write Vec<u8> String::from_utf8(vec)
```

```
Cursor:::new(buf)
```

Crée un , un lecteur mis en mémoire tampon qui lit à partir de . C'est ainsi que vous créez un lecteur qui lit à partir d'un fichier . L'argument peut être n'importe quel type qui implémente , de sorte que vous pouvez également passer un , , ou

```
. Cursor buf String buf AsRef<[u8]> &[u8] &str Vec<u8>
```

`Cursor` sont triviaux en interne. Ils n'ont que deux champs: lui-même et un entier, le décalage dans l'endroit où la lecture suivante commencera. La position est initialement 0. `buf`

Les curseurs implémentent , et . Si le type de est ou , alors le également implémente . L'écriture sur un curseur remplace les octets en commençant à la position actuelle. Si vous essayez d'écrire au-delà de la fin d'un , vous obtiendrez une écriture partielle ou un fichier . Utiliser un curseur pour écrire au-delà de la fin de a est bien, cependant: il fait croître le vecteur. et ainsi mettre en œuvre les quatre traits.

```
Read BufRead Seek buf &mut
[u8] Vec<u8> Cursor Write buf &mut
[u8] io::Error Vec<u8> Cursor<&mut
[u8]> Cursor<Vec<u8>> std::io::prelude
```

#### *std::net::TcpStream*

Représente une connexion réseau TCP. Puisque TCP permet la communication bidirectionnelle, c'est à la fois un lecteur et un écrivain.

La fonction associée au type tente de se connecter à un serveur et renvoie un fichier . TcpStream::connect(("hostname", PORT)) io::Result<TcpStream>

#### *std::process::Command*

Prend en charge la génération d'un processus enfant et la tuyauterie des données vers son entrée standard, comme suit :

```
use std::process::{Command, Stdio};

let mut child =
 Command::new("grep")
 .arg("-e")
 .arg("a.*e.*i.*o.*u")
 .stdin(Stdio::piped())
 .spawn()?

let mut to_child = child.stdin.take().unwrap();
for word in my_words {
 writeln!(to_child, "{}", word)?;
}
drop(to_child); // close grep's stdin, so it will exit
child.wait()?
```

Le type de est ; ici, nous avons utilisé lors de la configuration du processus enfant, donc est définitivement rempli quand réussit. Si nous ne l'avions pas fait, ce serait

```
.child.stdin Option<std::process::ChildStdin> .stdin(Stream::piped()) child.stdin .spawn() child.stdin None

Commande a également des méthodes similaires et , qui peuvent être utilisées pour demander des lecteurs dans et
.stdout() .stderr() child.stdout child.stderr
```

Le module offre également une poignée de fonctions qui renvoient des lecteurs et des écrivains triviaux: std::io

*io::sink()*

C'est l'écrivain no-op. Toutes les méthodes d'écriture renvoient , mais les données sont simplement ignorées. Ok

*io::empty()*

C'est le lecteur no-op. La lecture réussit toujours, mais renvoie la fin de l'entrée.

*io::repeat(byte)*

Renvoie un lecteur qui répète l'octet donné à l'infini.

## Données binaires, compression et sérialisation

De nombreuses caisses open source s'appuient sur le framework pour offrir des fonctionnalités supplémentaires. std::io

La caisse offre et des traits qui ajoutent des méthodes à tous les lecteurs et écrivains pour l'entrée et la sortie

binaires: byteorder ReadBytesExt WriteBytesExt

```
use byteorder::{ReadBytesExt, WriteBytesExt, LittleEndian};

let n = reader.read_u32::<LittleEndian>()?;
writer.write_i64::<LittleEndian>(n as i64)?;
```

La caisse fournit des méthodes d'adaptateur pour la lecture et l'écriture de données ped: flate2 gzip

```
use flate2::read::GzDecoder;
let file = File::open("access.log.gz")?;
let mut gzip_reader = GzDecoder::new(file);
```

La caisse, et ses caisses de format associées telles que , implémentent la sérialisation et la désérialisation: elles convertissent entre les structures Rust et les octets. Nous l'avons déjà mentionné une fois, dans [« Traits et](#)

[types d'autres personnes](#) ». Maintenant, nous pouvons regarder de plus près. `serde serde_json`

Supposons que nous ayons des données – la carte d'un jeu d'aventure textuel – stockées dans un : `HashMap`

```
type RoomId = String; // each room has a unique r
type RoomExits = Vec<(char, RoomId)>; // ...and a list of exits
type RoomMap = HashMap<RoomId, RoomExits>; // room names and exits, si

// Create a simple map.
let mut map = RoomMap::new();
map.insert("Cobble Crawl".to_string(),
 vec![('w', "Debris Room".to_string())]);
map.insert("Debris Room".to_string(),
 vec![('E', "Cobble Crawl".to_string()),
 ('w', "Sloping Canyon".to_string())]);
...
...
```

La transformation de ces données en JSON pour la sortie est une seule ligne de code :

```
serde_json::to_writer(&mut std::io::stdout(), &map)?;
```

En interne, utilise la méthode du trait. La bibliothèque attache ce trait à tous les types qu'elle sait sérialiser, et cela inclut tous les types qui apparaissent dans nos données : chaînes, caractères, tuples, vecteurs et

```
s. serde_json::to_writer serialize serde::Serialize HashMap
```

`serde` est flexible. Dans ce programme, la sortie est des données JSON, car nous avons choisi le sérialiseur. D'autres formats, comme Message-Pack, sont également disponibles. De même, vous pouvez envoyer cette sortie à un fichier, à un ou à tout autre scripteur. Le code précédent imprime les données sur `. Le voilà:`  `serde_json Vec<u8> stdout`

```
{"Debris Room": [["E", "Cobble Crawl"], ["W", "Sloping Canyon"]], "Cobble Cr
 [["W", "Debris Room"]] }
```

`serde` inclut également la prise en charge de la dérivation des deux traits clés : `serde`

```
#[derive(Serialize, Deserialize)]
struct Player {
```

```
 location: String,
 items: Vec<String>,
 health: u32
}
```

Cet attribut peut rendre vos compilations un peu plus longues, vous devez donc demander explicitement à le prendre en charge lorsque vous le répertoriez comme dépendance dans votre fichier *Cargo.toml*. Voici ce que nous avons utilisé pour le code précédent :#[derive] serde

```
[dependencies]
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
```

Consultez la documentation pour plus de détails. En bref, le système de build génère automatiquement des implémentations de et pour , de sorte que la sérialisation d'une valeur est simple

```
:serde serde::Serialize serde::Deserialize Player Player

 serde_json::to_writer(&mut std::io::stdout(), &player)?;
```

La sortie ressemble à ceci :

```
{"location": "Cobble Crawl", "items": ["a wand"], "health": 3}
```

## Fichiers et répertoires

Maintenant que nous avons montré comment travailler avec les lecteurs et les écrivains, les quelques sections suivantes couvrent les fonctionnalités de Rust pour travailler avec des fichiers et des répertoires, qui vivent dans les modules et. Toutes ces fonctionnalités impliquent de travailler avec des noms de fichiers, nous allons donc commencer par les types de noms de fichiers. std::path std::fs

### OsStr et chemin d'accès

Malheureusement, votre système d'exploitation ne force pas les noms de fichiers à être des Unicode valides. Voici deux commandes shell Linux qui créent des fichiers texte. Seul le premier utilise un nom de fichier UTF-8 valide :

```
$ echo "hello world" > ô.txt
$ echo "O brave new world, that has such filenames in't" > $'\xf4'.txt
```

Les deux commandes passent sans commentaire, car le noyau Linux ne connaît pas UTF-8 d’Ogg Vorbis. Pour le noyau, toute chaîne d’octets (à l’exclusion des octets nuls et des barres obliques) est un nom de fichier acceptable. C’est une histoire similaire sur Windows: presque n’importe quelle chaîne de « caractères larges » 16 bits est un nom de fichier acceptable, même les chaînes qui ne sont pas valides UTF-16. Il en va de même pour les autres chaînes gérées par le système d’exploitation, telles que les arguments de ligne de commande et les variables d’environnement.

Les chaînes Rust sont toujours des Unicode valides. Les noms de fichiers sont *presque* toujours Unicode dans la pratique, mais Rust doit faire face d’une manière ou d’une autre aux rares cas où ils ne le sont pas. C’est pourquoi Rust a et .  
std::ffi::OsStr OsString

OsStr est un type de chaîne qui est un sur-ensemble d’UTF-8. Son travail consiste à pouvoir représenter tous les noms de fichiers, les arguments de ligne de commande et les variables d’environnement sur le système actuel, *qu’ils soient Unicode valides ou non*. Sous Unix, an peut contenir n’importe quelle séquence d’octets. Sous Windows, an est stocké à l’aide d’une extension UTF-8 qui peut encoder n’importe quelle séquence de valeurs 16 bits, y compris des substituts inégalés. osstr osstr

Nous avons donc deux types de chaînes: pour les chaînes Unicode réelles; et pour toutes les absurdités que votre système d’exploitation peut faire. Nous en présenterons un de plus : , pour les noms de fichiers. Celui-ci est purement une commodité. est exactement comme , mais il ajoute de nombreuses méthodes pratiques liées au nom de fichier, que nous couvrirons dans la section suivante. À utiliser pour les chemins absous et relatifs. Pour un composant individuel d’un chemin d’accès, utilisez

```
.str OsStr std::path::Path Path OsStr Path OsStr
```

Enfin, pour chaque type de chaîne, il existe un type *de propriété* correspondant : un possède un tas alloué, un possède un tas alloué et un possède un tas alloué. [Le tableau 18-1](#) présente certaines des caractéristiques de chaque

```
type. String str std::ffi::OsString OsStr std::path::PathBuf P
ath
```

Tableau 18-1. Types de noms de fichiers

|                                                             | <b>Str</b>          | <b>OsStr</b>           | <b>Chemin</b>         |
|-------------------------------------------------------------|---------------------|------------------------|-----------------------|
| Type non dimensionné,<br>toujours transmis par<br>référence | Oui                 | Oui                    | Oui                   |
| Peut contenir n'importe quel<br>texte Unicode               | Oui                 | Oui                    | Oui                   |
| Ressemble à UTF-8,<br>normalement                           | Oui                 | Oui                    | Oui                   |
| Peut contenir des données non<br>Unicode                    | Non                 | Oui                    | Oui                   |
| Méthodes de traitement de<br>texte                          | Oui                 | Non                    | Non                   |
| Méthodes liées au nom de<br>fichier                         | Non                 | Non                    | Oui                   |
| Équivalent possédé, cultivable,<br>alloué en tas            | String              | OsString               | PathBuf               |
| Convertir en type possédé                                   | .to_string()<br>( ) | .to_os_string()<br>( ) | .to_path_buf()<br>( ) |

Ces trois types implémentent un trait commun, de sorte que nous pouvons facilement déclarer une fonction générique qui accepte « n'importe quel type de nom de fichier » comme argument. Cela utilise une technique que nous avons montrée dans [« AsRef et AsMut »](#): AsRef<Path>

```
use std::path::Path;
use std::io;

fn swizzle_file<P>(path_arg: P) -> io::Result<()>
 where P: AsRef<Path>
{
 let path = path_arg.as_ref();
```

```
...
}
```

Toutes les fonctions et méthodes standard qui prennent des arguments utilisent cette technique, de sorte que vous pouvez librement passer des littéraux de chaîne à n'importe lequel d'entre eux. path

## Méthodes Path et PathBuf

Path offre les méthodes suivantes, entre autres:

```
Path::new(str)
```

Convertit un ou en un fichier . Cela ne copie pas la chaîne. Le nouveau pointe vers les mêmes octets que l'original ou  
`:&str &OsStr &Path &Path &str &OsStr`

```
use std::path::Path;
let home_dir = Path::new("/home/fwolfe");
```

(La méthode similaire convertit a en un  
. OsStr::new(str) &str &OsStr

```
path.parent()
```

Renvoie le répertoire parent du chemin d'accès, le cas échéant. Le type de retour est . Option<&Path>

Cela ne copie pas le chemin d'accès. Le répertoire parent de est toujours une sous-chaîne de : path path

```
assert_eq!(Path::new("/home/fwolfe/program.txt").parent(),
 Some(Path::new("/home/fwolfe")));
```

```
path.file_name()
```

Renvoie le dernier composant de , le cas échéant. Le type de retour est . path Option<&OsStr>

Dans le cas typique, où se compose d'un répertoire, puis d'une barre oblique, puis d'un nom de fichier, cela renvoie le nom de fichier

: path

```
use std::ffi::OsStr;
assert_eq!(Path::new("/home/fwolfe/program.txt").file_name(),
 Some(OsStr::new("program.txt")));
```

```
path.is_absolute(), path.is_relative()
```

Ceux-ci indiquent si le fichier est absolu, comme le chemin Unix `/usr/bin/advent` ou le chemin Windows `C:\Program Files`, ou relatif, comme `src/main.rs`.

```
path1.join(path2)
```

Joint deux chemins, en renvoyant un nouveau : PathBuf

```
let path1 = Path::new("/usr/share/dict");
assert_eq!(path1.join("words"),
 Path::new("/usr/share/dict/words"));
```

Si est un chemin absolu, cela renvoie simplement une copie de , de sorte que cette méthode peut être utilisée pour convertir n'importe quel chemin en chemin absolu : path2 path2

```
let abs_path = std::env::current_dir()?.join(any_path);
```

```
path.components()
```

Renvoie un itérateur sur les composants du chemin d'accès donné, de gauche à droite. Le type d'élément de cet itérateur est , un enum qui peut représenter tous les différents éléments pouvant apparaître dans les noms de fichiers : std::path::Component

```
pub enum Component<'a> {
 Prefix(PrefixComponent<'a>), // a drive letter or share (on Windows)
 RootDir, // the root directory, `/` or `\\`
 CurDir, // the `.` special directory
 ParentDir, // the `..` special directory
 Normal(&'a OsStr) // plain file and directory names
}
```

Par exemple, le chemin Windows `||venice|Music|A Love Supreme|04-Psalm.mp3` se compose d'un `||venice|Music` représentant un , puis de deux composants représentant `A Love Supreme` et `04-Psalm.mp3`. Prefix RootDir Normal

Pour plus de détails, [consultez la documentation en ligne](#).

```
path.ancestors()
```

Renvoie un itérateur qui se déplace jusqu'à la racine. Chaque article produit est un : d'abord lui-même, puis son parent, puis son grand-parent, et ainsi de suite : path Path path

```

let file = Path::new("/home/jimb/calendars/calendar-18x18.pdf");
assert_eq!(file.ancestors().collect::<Vec<_>>(),
 vec![Path::new("/home/jimb/calendars/calendar-18x18.pdf",
 Path::new("/home/jimb/calendars"),
 Path::new("/home/jimb"),
 Path::new("/home"),
 Path::new("/")]);

```

C'est un peu comme appeler à plusieurs reprises jusqu'à ce qu'il revienne . L'élément final est toujours un chemin racine ou préfixe. parent None

Ces méthodes fonctionnent sur des chaînes en mémoire. ont également des méthodes qui interrogent le système de fichiers : , , , , et ainsi de suite. Consultez la documentation en ligne pour en savoir plus. Path .exists() .is\_file() .is\_dir() .read\_dir() .canonicalize()

Il existe trois méthodes pour convertir s en chaînes. Chacun d'eux permet la possibilité d'utf-8 non valide dans le : Path Path

*path.to\_str()*

Convertit a en chaîne, sous la forme d'un fichier . Si UTF-8 n'est pas valide, cela renvoie : Path Option<&str> path None

```

if let Some(file_str) = path.to_str() {
 println!("{}", file_str);
} // ...otherwise skip this weirdly named file

```

*path.to\_string\_lossy()*

C'est fondamentalement la même chose, mais il parvient à renvoyer une sorte de chaîne dans tous les cas. Si UTF-8 n'est pas valide, ces méthodes effectuent une copie, en remplaçant chaque séquence d'octets non valide par le caractère de remplacement Unicode, U+FFFD (' '). path

Le type de retour est : une chaîne empruntée ou possédée. Pour obtenir un à partir de cette valeur, utilisez sa méthode. (Pour en savoir plus sur , voir [« Emprunter et posséder au travail : l'humble vache ».](#)) std::borrow::Cow<str> String .to\_owned() Cow

*path.display()*

Ceci est pour les chemins d'impression:

```
println!("Download found. You put it in: {}", dir_path.display());
```

La valeur renvoyée n'est pas une chaîne, mais elle implémente , de sorte qu'elle peut être utilisée avec , et des amis. Si le chemin d'accès n'est pas valide UTF-8, la sortie peut contenir le caractère. std::fmt::Display format!() println!()

## Fonctions d'accès au système de fichiers

Le tableau 18-2 montre certaines des fonctions et leurs équivalents approximatifs sous Unix et Windows. Toutes ces fonctions renvoient des valeurs. Sauf indication contraire, ils le sont. std::fs::io::Result<()>

Tableau 18-2. Résumé des fonctions d'accès au système de fichiers

| Fonction rouille                    | Unix                                                             | Windows                                   |
|-------------------------------------|------------------------------------------------------------------|-------------------------------------------|
| Création et suppression             | <code>create_dir(path)</code>                                    | <code>mkdir()</code>                      |
|                                     |                                                                  | <code>CreateDirectory()</code>            |
|                                     | <code>create_dir_all(path)</code>                                | <code>comme mkdir -p</code>               |
|                                     | <code>remove_dir(path)</code>                                    | <code>rmdir()</code>                      |
|                                     |                                                                  | <code>RemoveDirectory()</code>            |
|                                     | <code>remove_dir_all(path)</code>                                | <code>comme rm -r</code>                  |
|                                     | <code>remove_file(path)</code>                                   | <code>unlink()</code>                     |
| Copier, déplacer et lier Inspection | <code>copy(src_path, dest_path) -&gt; Result&lt;u64&gt;</code>   | <code>comme cp -p</code>                  |
|                                     | <code>rename(src_path, dest_path)</code>                         | <code>renam e()</code>                    |
|                                     | <code>hard_link(src_path, dest_path)</code>                      | <code>link()</code>                       |
|                                     | <code>canonicalize(path) -&gt; Result&lt;PathBuf&gt;</code>      | <code>realpath()</code>                   |
|                                     | <code>metadata(path) -&gt; Result&lt;Metadata&gt;</code>         | <code>GetFileInformationByHandle()</code> |
|                                     | <code>symlink_metadata(path) -&gt; Result&lt;Metadata&gt;</code> | <code>lstat()</code>                      |
|                                     |                                                                  | <code>GetFileInformationByHandle()</code> |

| Fonction rouille                   | Unix       | Windows                 |
|------------------------------------|------------|-------------------------|
| read_dir(path) -> Result<ReadDir>  | opendir()  | FindFirstFile()         |
| read_link(path) -> Result<PathBuf> | readlink() | FSCTL_GET_REPARSE_POINT |
| set_permissions(path, perm)        | chmod()    | SetFileAttributes()     |

Autorisations

(Le nombre renvoyé par est la taille du fichier copié, en octets. Pour créer des liens symboliques, voir [« Fonctionnalités spécifiques à la plate-forme »](#).)

copy()

Comme vous pouvez le voir, Rust s'efforce de fournir des fonctions portables qui fonctionnent de manière prévisible sur Windows ainsi que sur macOS, Linux et d'autres systèmes Unix.

Un tutoriel complet sur les systèmes de fichiers dépasse le cadre de ce livre, mais si vous êtes curieux de connaître l'une de ces fonctions, vous pouvez facilement en trouver plus à leur sujet en ligne. Nous montrerons quelques exemples dans la section suivante.

Toutes ces fonctions sont implémentées en appelant le système d'exploitation. Par exemple, n'utilise pas simplement le traitement de chaîne pour éliminer et à partir du fichier . Il résout les chemins relatifs à l'aide du répertoire de travail actuel et recherche les liens symboliques. C'est une erreur si le chemin n'existe

```
pas. std::fs::canonicalize(path) . . . path
```

Type produit par et contenant des informations telles que le type et la taille du fichier, les autorisations et les horodatages. Comme toujours, consultez la documentation pour plus de détails.

```
Metadata std::fs::metadata(path) std::fs::symlink_metadata(path)
```

Pour plus de commodité, le type a quelques-uns d'entre eux intégrés comme méthodes: , par exemple, est la même chose que

```
.Path path.metadata() std::fs::metadata(path)
```

## Lecture de répertoires

Pour lister le contenu d'un répertoire, utilisez ou, de manière équivalente, la méthode d'un : `std::fs::read_dir .read_dir() Path`

```
for entry_result in path.read_dir()? {
 let entry = entry_result?;
 println!("{}", entry.file_name().to_string_lossy());
}
```

Notez les deux utilisations de dans ce code. La première ligne vérifie les erreurs d'ouverture du répertoire. La deuxième ligne vérifie les erreurs de lecture de l'entrée suivante. ?

Le type de est , et c'est une structure avec seulement quelques méthodes: `entry std::fs::DirEntry`

`entry.file_name()`

Nom du fichier ou du répertoire, sous la forme d'un fichier `OsString`

`entry.path()`

C'est la même chose, mais avec le chemin d'origine qui y est joint, produisant un nouveau . Si le répertoire que nous répertorions est , et est , alors renvoie `.PathBuf "/home/jimb" entry.file_name() ".emacs" entry.path() PathBuf::from("/home/jimb/.emacs")`

`entry.file_type()`

Renvoie un fichier . a , et

méthodes. `io::Result<FileType> FileType .is_file() .is_dir() .is_symlink()`

`entry.metadata()`

Obtient le reste des métadonnées relatives à cette entrée.

Les répertoires spéciaux et ne sont pas répertoriés lors de la lecture d'un répertoire. . . .

Voici un exemple plus substantiel. Le code suivant copie récursivement une arborescence de répertoires d'un emplacement à un autre sur le disque :

```
use std::fs;
use std::io;
use std::path::Path;

/// Copy the existing directory `src` to the target path `dst`.
fn copy_dir_to(src: &Path, dst: &Path) -> io::Result<()> {
```

```

 if !dst.is_dir() {
 fs::create_dir(dst)?;
 }

 for entry_result in src.read_dir()? {
 let entry = entry_result?;
 let file_type = entry.file_type()?;
 copy_to(&entry.path(), &file_type, &dst.join(entry.file_name()))
 }

 Ok(())
}

```

Une fonction distincte, , copie les entrées de répertoire individuelles  
`:copy_to`

```

/// Copy whatever is at `src` to the target path `dst`.
fn copy_to(src: &Path, src_type: &fs::FileType, dst: &Path)
 -> io::Result<()>
{
 if src_type.is_file() {
 fs::copy(src, dst)?;
 } else if src_type.is_dir() {
 copy_dir_to(src, dst)?;
 } else {
 return Err(io::Error::new(io::ErrorKind::Other,
 format!("don't know how to copy: {}",
 src.display())));
 }
 Ok(())
}

```

## Fonctionnalités spécifiques à la plate-forme

Jusqu'à présent, notre fonction peut copier des fichiers et des répertoires. Supposons que nous voulions également prendre en charge les liens symboliques sous Unix. `copy_to`

Il n'existe aucun moyen portable de créer des liens symboliques qui fonctionnent à la fois sous Unix et Windows, mais la bibliothèque standard offre une fonction spécifique à Unix : `symlink`

```
use std::os::unix::fs::symlink;
```

Avec cela, notre travail est facile. Il suffit d'ajouter une branche à l'expression dans : if `copy_to`

```
...
} else if src_type.is_symlink() {
 let target = src.read_link()?;
 symlink(target, dst)?;
...
}
```

Cela fonctionnera tant que nous compilerons notre programme uniquement pour les systèmes Unix, tels que Linux et macOS.

Le module contient diverses fonctionnalités spécifiques à la plate-forme, telles que . Le corps réel de dans la bibliothèque standard ressemble à ceci (en prenant une licence poétique): `std::os::symlink` `std::os::os_specific`

```
//! OS-specific functionality.
```

```
#[cfg(unix)] pub mod unix;
#[cfg(windows)] pub mod windows;
#[cfg(target_os = "ios")] pub mod ios;
#[cfg(target_os = "linux")] pub mod linux;
#[cfg(target_os = "macos")] pub mod macos;
...

```

L'attribut indique une compilation conditionnelle : chacun de ces modules n'existe que sur certaines plateformes. C'est pourquoi notre programme modifié, utilisant , compilera avec succès uniquement pour Unix: sur d'autres plates-formes, n'existe pas. #

```
[cfg] std::os::unix std::os::unix
```

Si nous voulons que notre code soit compilé sur toutes les plates-formes, avec la prise en charge des liens symboliques sous Unix, nous devons également l'utiliser dans notre programme. Dans ce cas, il est plus facile d'importer sur Unix, tout en définissant notre propre stub sur d'autres systèmes:#[cfg] `symlink`

```
#[cfg(unix)]
use std::os::unix::fs::symlink;

/// Stub implementation of `symlink` for platforms that don't provide it
#[cfg(not(unix))]
fn symlink<P: AsRef<Path>, Q: AsRef<Path>>(<src: P, _dst: Q)
 -> std::io::Result<()>
```

```

 {
 Err(io::Error::new(io::ErrorKind::Other,
 format!("can't copy symbolic link: {}",
 src.as_ref().display())))
 }

```

Il s'avère que c'est un cas particulier. La plupart des fonctionnalités spécifiques à Unix ne sont pas des fonctions autonomes, mais plutôt des caractéristiques d'extension qui ajoutent de nouvelles méthodes aux types de bibliothèque standard. (Nous avons couvert les traits [d'extension dans « Traits et types d'autres personnes »](#).) Il existe un module qui peut être utilisé pour activer toutes ces extensions à la fois : `symlink prelude`

```
use std::os::unix::prelude::*;


```

Par exemple, sous Unix, cela ajoute une méthode à `stat`, fournissant l'accès à la valeur sous-jacente qui représente les autorisations sous Unix. De même, il s'étend avec des accesseurs pour les champs de la valeur sous-jacente, tels que `.uid()`, l'ID utilisateur du propriétaire du fichier.

```
.mode() std::fs::Permissions u32 std::fs::Metadata struct stat .uid()
```

Tout compte fait, ce qu'il y a dedans est assez basique. Beaucoup plus de fonctionnalités spécifiques à la plate-forme sont disponibles via des caisses tierces, comme [winreg](#) pour accéder au registre Windows. `std::os`

## Réseautage

Un tutoriel sur le réseautage dépasse largement le cadre de ce livre. Cependant, si vous en savez déjà un peu plus sur la programmation réseau, cette section vous aidera à démarrer avec la mise en réseau dans Rust.

Pour le code réseau de bas niveau, commencez par le module, qui fournit une prise en charge multiplateforme pour la mise en réseau TCP et UDP. Utilisez la caisse pour la prise en charge SSL/TLS.

```
std::net native_tls
```

Ces modules fournissent les blocs de construction pour une entrée et une sortie simples et bloquantes sur le réseau. Vous pouvez écrire un serveur simple en quelques lignes de code, en utilisant et en générant un thread pour chaque connexion. Par exemple, voici un serveur « echo »

```
: std::net
```

```

use std::net::TcpListener;
use std::io;
use std::thread::spawn;

/// Accept connections forever, spawning a thread for each one.
fn echo_main(addr: &str) -> io::Result<()> {
 let listener = TcpListener::bind(addr)?;
 println!("listening on {}", addr);
 loop {
 // Wait for a client to connect.
 let (mut stream, addr) = listener.accept()?;
 println!("connection received from {}", addr);

 // Spawn a thread to handle this client.
 let mut write_stream = stream.try_clone()?;
 spawn(move || {
 // Echo everything we receive from `stream` back to it.
 io::copy(&mut stream, &mut write_stream)
 .expect("error in client thread: ");
 println!("connection closed");
 });
 }
}

fn main() {
 echo_main("127.0.0.1:17007").expect("error: ");
}

```

Un serveur d'écho répète simplement tout ce que vous lui envoyez. Ce type de code n'est pas si différent de ce que vous écririez en Java ou en Python. (Nous couvrirons dans [le chapitre suivant](#)) `std::thread::spawn()`

Toutefois, pour les serveurs hautes performances, vous devrez utiliser des entrées et des sorties asynchrones. [Le chapitre 20](#) couvre la prise en charge de rust pour la programmation asynchrone et montre le code complet d'un client et d'un serveur réseau.

Les protocoles de niveau supérieur sont pris en charge par des caisses tierces. Par exemple, la caisse offre une belle API pour les clients HTTP. Voici un programme complet de ligne de commande qui récupère n'importe quel document avec une URL ou un vichage sur votre terminal. Ce code a été écrit à l'aide de `cargo`, avec sa fonctionnalité activée. fournit également

```

ment une interface asynchrone. reqwest http: https: reqwest =
"0.11" "blocking" reqwest

use std::error::Error;
use std::io;

fn http_get_main(url: &str) -> Result<(), Box

```

Le framework pour les serveurs HTTP offre des touches de haut niveau telles que les et traits, qui vous aident à composer une application à partir de parties enfichables. La caisse implémente le protocole WebSocket. Et ainsi de suite. Rust est un langage jeune avec un écosystème open source occupé. La prise en charge de la mise en réseau se développe rapidement. `actix-web` Service Transform `websocket`

# Chapitre 19. Concurrence

*À long terme, il n'est pas conseillé d'écrire de grands programmes simultanés dans des langages orientés machine qui permettent une utilisation sans restriction des emplacements de magasin et de leurs adresses. Il n'y a tout simplement aucun moyen de rendre de tels programmes fiables (même à l'aide de mécanismes matériels compliqués).*

—Per Brinch Hansen (1977)

*Les modèles de communication sont des modèles de parallélisme.*

—Whit Morris

Si votre attitude à l'égard de la concurrence a changé au cours de votre carrière, vous n'êtes pas seul. C'est une histoire commune.

Au début, écrire du code simultané est facile et amusant. Les outils (threads, verrous, files d'attente, etc.) sont faciles à prendre en main et à utiliser. Il y a beaucoup de pièges, c'est vrai, mais heureusement vous savez ce qu'ils sont tous, et vous faites attention à ne pas faire d'erreurs.

À un moment donné, vous devez déboguer le code multithread de quelqu'un d'autre, et vous êtes obligé de conclure que *certaines personnes* ne devraient vraiment pas utiliser ces outils.

Ensuite, à un moment donné, vous devez déboguer votre propre code multithread.

L'expérience inculque un scepticisme sain, sinon un cynisme pur et simple, envers tout le code multithread. Ceci est aidé par l'article occasionnel expliquant en détail pourquoi un idiome multithreading évidemment correct ne fonctionne pas du tout. (Cela a à voir avec « le modèle de mémoire. ») Mais vous finissez par trouver une approche de la concurrence que vous pensez pouvoir utiliser de manière réaliste sans faire constamment d'erreurs. Vous pouvez mettre à peu près tout dans cet idiome, et (si vous êtes vraiment bon) vous apprenez à dire « non » à la complexité supplémentaire.

Bien sûr, il y a plutôt beaucoup d'idiomes. Les approches que les programmeurs de systèmes utilisent couramment sont les suivantes :

- Un *thread d'arrière-plan* qui a un seul travail et se réveille périodiquement pour le faire.

- Pools de travail à usage général qui communiquent avec *les clients* via *des files d'attente de tâches*.
- *Pipelines* où les données circulent d'un thread à l'autre, chaque thread faisant un peu de travail.
- *Parallélisme des données*, où l'on suppose (à tort ou à raison) que l'ensemble de l'ordinateur ne fera principalement qu'un seul grand calcul, qui est donc divisé en  $n$  morceaux et exécuté sur  $n$  threads dans l'espoir de faire fonctionner tous les  $n$  cœurs de la machine en même temps.
- *Une mer d'objets synchronisés*, où plusieurs threads ont accès aux mêmes données, et les races sont évitées en utilisant des schémas de *verrouillage ad hoc* basés sur des primitives de bas niveau comme les mutex. (Java inclut un support intégré pour ce modèle, qui était très populaire dans les années 1990 et 2000.)
- *Les opérations d'entier atomique* permettent à plusieurs cœurs de communiquer en transmettant des informations à travers des champs de la taille d'un mot machine. (C'est encore plus difficile à obtenir que tous les autres, à moins que les données échangées ne soient littéralement que des valeurs entières. En pratique, il s'agit généralement de pointeurs.)

Avec le temps, vous pourrez peut-être utiliser plusieurs de ces approches et les combiner en toute sécurité. Vous êtes un maître de l'art. Et les choses seraient géniales, si seulement personne d'autre n'était jamais autorisé à modifier le système de quelque manière que ce soit. Les programmes qui utilisent bien les threads sont pleins de règles non écrites.

Rust offre une meilleure façon d'utiliser la concurrence, non pas en forçant tous les programmes à adopter un seul style (ce qui, pour les programmeurs de systèmes, ne serait pas du tout une solution), mais en prenant en charge plusieurs styles en toute sécurité. Les règles non écrites sont écrites (dans du code) et appliquées par le compilateur.

Vous avez entendu dire que Rust vous permet d'écrire des programmes sûrs, rapides et simultanés. C'est le chapitre où nous vous montrons comment c'est fait. Nous couvrirons trois façons d'utiliser les fils Rust :

- Parallélisme fourche-jointure
- Canaux
- État mutable partagé

En cours de route, vous allez utiliser tout ce que vous avez appris jusqu'à présent sur le langage Rust. Le soin que Rust prend avec les références, la mutabilité et les durées de vie est suffisamment précieux dans les programmes à thread unique, mais c'est dans la programmation simultanée

que la véritable signification de ces règles devient évidente. Ils permettent d'élargir votre boîte à outils, de pirater plusieurs styles de code multithread rapidement et correctement, sans scepticisme, sans cynisme, sans peur.

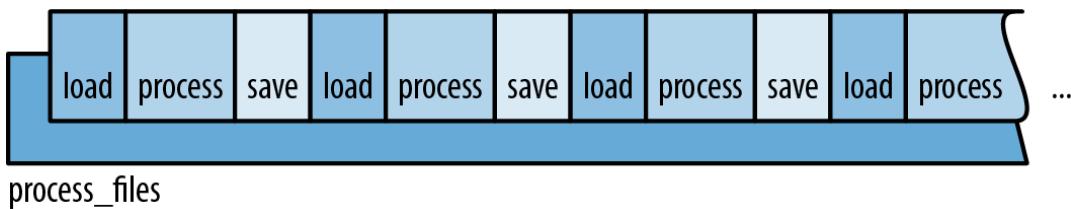
## Parallélisme fourche-jointure

Les cas d'utilisation les plus simples pour les threads surviennent lorsque nous avons plusieurs tâches complètement indépendantes que nous aimerais effectuer à la fois.

Par exemple, supposons que nous fassions du traitement du langage naturel sur un grand corpus de documents. Nous pourrions écrire une boucle :

```
fn process_files(filenames: Vec<String>) -> io::Result<()> {
 for document in filenames {
 let text = load(&document)?; // read source file
 let results = process(text); // compute statistics
 save(&document, results)?; // write output file
 }
 Ok(())
}
```

Le programme s'exécuterait comme illustré à [la figure 19-1](#).



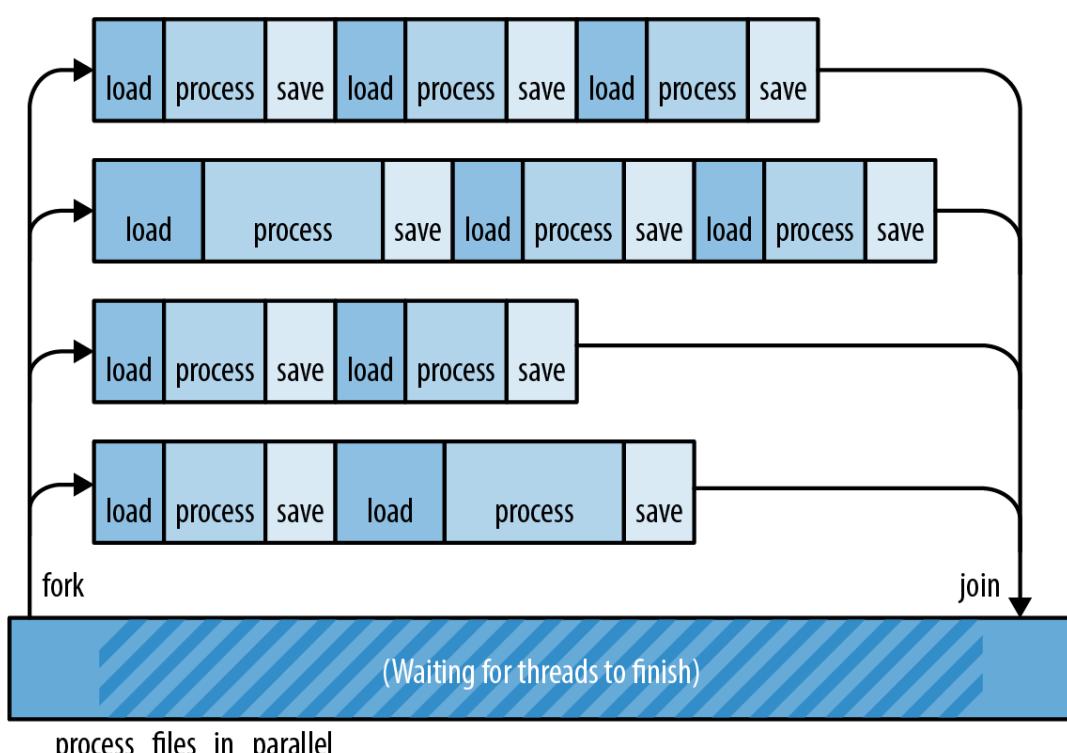
Graphique 19-1. Exécution monothread de `process_files()`

Étant donné que chaque document est traité séparément, il est relativement facile d'accélérer cette tâche en divisant le corpus en morceaux et en traitant chaque morceau sur un fil distinct, comme le montre [la figure 19-2](#).

Ce modèle est appelé *parallélisme fourche-jointure*. *Forker*, c'est commencer un nouveau thread, et *joindre* un thread, c'est attendre qu'il se termine. Nous avons déjà vu cette technique : nous l'avons utilisée pour accélérer le programme Mandelbrot au [chapitre 2](#).

Le parallélisme fourche-jointure est attrayant pour plusieurs raisons :

- C'est très simple. Fork-join est facile à mettre en œuvre, et Rust le rend facile à obtenir.
- Il évite les goulots d'étranglement. Il n'y a pas de verrouillage des ressources partagées dans fork-join. Le seul moment où un fil doit attendre un autre est à la fin. En attendant, chaque thread peut fonctionner librement. Cela permet de maintenir les frais généraux de changement de tâche bas.
- Le calcul de la performance est simple. Dans le meilleur des cas, en démarquant quatre fils, nous pouvons terminer notre travail en un quart du temps. [La figure 19-2](#) montre une raison pour laquelle nous ne devrions pas nous attendre à cette accélération idéale : nous pourrions ne pas être en mesure de répartir le travail uniformément sur tous les fils. Une autre raison de prudence est que parfois les programmes de jointure de fourche doivent passer un certain temps après la jonction des threads *à combiner* les résultats calculés par les threads. Autrement dit, isoler complètement les tâches peut faire un peu de travail supplémentaire. Pourtant, en dehors de ces deux choses, tout programme lié au processeur avec des unités de travail isolées peut s'attendre à un coup de pouce significatif.
- Il est facile de raisonner sur l'exactitude du programme. Un programme de jointure de fourche est *déterministe* tant que les threads sont vraiment isolés, comme les threads de calcul dans le programme Mandelbrot. Le programme produit toujours le même résultat, quelles que soient les variations de vitesse de thread. C'est un modèle de concurrence sans conditions de concurrence.



Graphique 19-2. Traitement de fichiers multithread à l'aide d'une approche de jointure à fourche

Le principal inconvénient de la fourche-jointure est qu'elle nécessite des unités de travail isolées. Plus loin dans ce chapitre, nous examinerons certains problèmes qui ne se divisent pas aussi proprement.

Pour l'instant, restons avec l'exemple du traitement du langage naturel. Nous allons montrer quelques façons d'appliquer le modèle de jointure de fourche à la fonction `process_files`

## spawn et rejoindre

La fonction démarre un nouveau thread : `std::thread::spawn`

```
use std::thread;

thread::spawn(|| {
 println!("hello from a child thread");
});
```

Il faut un argument, une fermeture ou une fonction. Rust démarre un nouveau thread pour exécuter le code de cette fermeture ou fonction. Le nouveau thread est un véritable thread de système d'exploitation avec sa propre pile, tout comme les threads en C++, C# et Java. FnOnce

Voici un exemple plus substantiel, utilisé pour implémenter une version parallèle de la fonction d'avant: `spawn process_files`

```
use std::{thread, io};

fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
 // Divide the work into several chunks.
 const NTHREADS: usize = 8;
 let worklists = split_vec_into_chunks(filenames, NTHREADS);

 // Fork: Spawn a thread to handle each chunk.
 let mut thread_handles = vec![];
 for worklist in worklists {
 thread_handles.push(
 thread::spawn(move || process_files(worklist)))
 }

 // Join: Wait for all threads to finish.
 for handle in thread_handles {
 handle.join().unwrap()?;
 }
}
```

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

Prenons cette fonction ligne par ligne.

```
fn process_files_in_parallel(filenames: Vec<String>) -> io::Result<()> {
```

Notre nouvelle fonction a la même signature de type que l'original, ce qui en fait un remplacement pratique. `process_files`

```
// Divide the work into several chunks.
const NTHREADS: usize = 8;
let worklists = split_vec_into_chunks(filenames, NTHREADS);
```

Nous utilisons une fonction utilitaire, non montrée ici, pour diviser le travail. Le résultat, , est un vecteur de vecteurs. Il contient huit portions de taille égale du vecteur

d'origine. `split_vec_into_chunks worklists filenames`

```
// Fork: Spawn a thread to handle each chunk.
let mut thread_handles = vec![];
for worklist in worklists {
 thread_handles.push(
 thread::spawn(move || process_files(worklist))
);
}
```

Nous créons un fil pour chaque fichier . renvoie une valeur appelée a , que nous utiliserons ultérieurement. Pour l'instant, nous mettons tous les s dans un vecteur. `worklist spawn() JoinHandle JoinHandle`

Notez comment nous obtenons la liste des noms de fichiers dans le thread de travail :

- `worklist` est défini et rempli par la boucle, dans le thread parent. `for`
- Dès que la fermeture est créée, est déplacé dans la fermeture. `move worklist`
- `spawn` puis déplace la fermeture (y compris le vecteur) sur le nouveau thread enfant. `worklist`

Ces déménagements sont bon marché. Comme les mouvements dont nous avons discuté au [chapitre 4](#), les s ne sont pas clonés. En fait, rien n'est alloué ou libéré. La seule donnée déplacée est elle-même : trois mots machine. `Vec<String> String Vec`

La plupart des threads que vous créez ont besoin à la fois de code et de données pour commencer. Les fermetures antirouille contiennent facilement le code que vous voulez et les données que vous voulez.

Passons à autre chose :

```
// Join: Wait for all threads to finish.
for handle in thread_handles {
 handle.join().unwrap()?
}
```

Nous utilisons la méthode `join` que nous avons collectés plus tôt pour attendre que les huit fils se terminent. Joindre des threads est souvent nécessaire pour l'exactitude, car un programme Rust se ferme dès qu'il revient, même si d'autres threads sont toujours en cours d'exécution. Les destructeurs ne sont pas appelés; les fils supplémentaires sont juste tués. Si ce n'est pas ce que vous voulez, assurez-vous de rejoindre tous les fils de discussion qui vous intéressent avant de revenir de

```
..join() JoinHandle main main
```

Si nous parvenons à passer à travers cette boucle, cela signifie que les huit threads enfants se sont terminés avec succès. Notre fonction se termine donc par le retour : `Ok(())`

```
Ok()
}
```

## Gestion des erreurs entre les threads

Le code que nous avons utilisé pour joindre les threads enfants dans notre exemple est plus délicat qu'il n'y paraît, en raison de la gestion des erreurs. Revisitons cette ligne de code :

```
handle.join().unwrap()?
```

La méthode fait deux choses intéressantes pour nous. `.join()`

Tout d'abord, renvoie une erreur *si le thread enfant a paniqué*. Cela rend le threading dans Rust considérablement plus robuste qu'en C++. En C++, un accès au tableau hors limites est un comportement indéfini et le reste du système ne protège pas les conséquences. Dans Rust, la panique est sûre et par fil. Les limites entre les threads servent de pare-feu pour la panique ; la panique ne se propage pas automatiquement d'un thread aux threads qui en dépendent. Au lieu de cela, une panique dans un

thread est signalée comme une erreur dans d'autres threads. Le programme dans son ensemble peut facilement récupérer.

```
handle.join() std::thread::Result Result
```

Dans notre programme, cependant, nous ne tentons pas de gérer la panique de manière sophistiquée. Au lieu de cela, nous utilisons immédiatement sur ce point, affirmant qu'il s'agit d'un résultat et non d'un résultat. Si un thread enfant *paniquait*, cette assertion échouerait, de sorte que le thread parent paniquerait également. Nous propageons explicitement la panique des threads enfants vers le thread parent.

```
.unwrap() Result Ok Err
```

Deuxièmement, transmet la valeur de retour du thread enfant au thread parent. La fermeture à laquelle nous sommes passés a un type de retour de , parce que c'est ce qui retourne. Cette valeur renvoyée n'est pas ignorée. Lorsque le thread enfant a terminé, sa valeur de retour est enregistrée et transfère cette valeur au thread

```
parent.handle.join() spawn io::Result<()> process_files Join
Handle::join()
```

Le type complet renvoyé par dans ce programme est . Le fait partie de l'API / ; le fait partie de notre

```
application.handle.join() std::thread::Result<std::io::Result
<()>> thread::Result spawn join io::Result
```

Dans notre cas, après avoir déballé le , nous utilisons l'opérateur sur le , propageant explicitement les erreurs d'E/S des threads enfants vers le thread parent.

```
thread::Result ? io::Result
```

Tout cela peut sembler assez complexe. Mais considérez qu'il ne s'agit que d'une ligne de code, puis comparez-la avec d'autres langages. Le comportement par défaut en Java et C# est que les exceptions dans les threads enfants soient vidées vers le terminal, puis oubliées. En C++, la valeur par défaut est d'abandonner le processus. Dans Rust, les erreurs sont des valeurs (données) au lieu d'exceptions (flux de contrôle). Ils sont livrés sur des threads comme n'importe quelle autre valeur. Chaque fois que vous utilisez des API de threading de bas niveau, vous finissez par devoir écrire du code de gestion des erreurs avec soin, mais *étant donné que vous devez l'écrire, c'est très agréable à avoir.*

## Partage de données immuables entre threads

Supposons que l'analyse que nous effectuons nécessite une grande base de données de mots et d'expressions anglais :

```

// before
fn process_files(filenames: Vec<String>)

// after
fn process_files(filenames: Vec<String>, glossary: &GigabyteMap)

```

Cela va être gros, alors nous le transmettons par référence. Comment pouvons-nous mettre à jour pour transmettre le glossaire aux threads de travail ? `glossary` `process_files_in_parallel`

Le changement évident ne fonctionne pas :

```

fn process_files_in_parallel(filenames: Vec<String>,
 glossary: &GigabyteMap)
 -> io::Result<()>
{
 ...
 for worklist in worklists {
 thread_handles.push(
 spawn(move || process_files(worklist, glossary)) // error
);
 }
 ...
}

```

Nous avons simplement ajouté un argument à notre fonction et l'avons transmis à . Rust se plaint : `glossary` `process_files`

```

error: explicit lifetime required in the type of `glossary`

 |

38 | spawn(move || process_files(worklist, glossary)) // er

 | ^^^^^^ lifetime `static` required

```

Rust se plaint de la durée de vie de la fermeture à laquelle nous passons, et le message « utile » que le compilateur présente ici n'est en fait d'aucune aide. `spawn`

`spawn` lance des threads indépendants. Rust n'a aucun moyen de savoir combien de temps le thread enfant s'exécutera, il suppose donc le pire : il suppose que le thread enfant peut continuer à fonctionner même après la fin du thread parent et que toutes les valeurs du thread parent ont disparu. De toute évidence, si le fil enfant doit durer aussi longtemps, la fermeture qu'il exécute doit durer aussi longtemps. Mais cette fermeture a une durée de vie limitée : elle dépend de la référence, et les références ne durent pas éternellement. `glossary`

Notez que Rust a raison de rejeter ce code! De la façon dont nous avons écrit cette fonction, il est possible qu'un thread rencontre une erreur d'E/S, ce qui provoque un renflouement avant que les autres threads ne soient terminés. Les fils de discussion enfants pourraient finir par essayer d'utiliser le glossaire une fois que le thread principal l'a libéré. Ce serait une course – avec un comportement indéfini comme prix, si le fil conducteur devait gagner. La rouille ne peut pas permettre cela.

```
process_files_in_parallel
```

Il semble être trop ouvert pour prendre en charge le partage de références entre les threads. En effet, nous avons déjà vu un cas comme celui-ci, dans [« Fermetures qui volent »](#). Là, notre solution consistait à transférer la propriété des données vers le nouveau thread, à l'aide d'une fermeture. Cela ne fonctionnera pas ici, car nous avons de nombreux threads qui doivent tous utiliser les mêmes données. Une alternative sûre est à l'ensemble du glossaire pour chaque fil, mais comme il est grand, nous voulons éviter cela. Heureusement, la bibliothèque standard offre un autre moyen : le comptage de référence atomique. `spawn move clone`

Nous l'avons décrit dans [« Rc et Arc : Propriété partagée »](#). Il est temps de l'utiliser: `Arc`

```
use std::sync::Arc;

fn process_files_in_parallel(filenames: Vec<String>,
 glossary: Arc<GigabyteMap>)
 -> io::Result<()>
{
 ...
 for worklist in worklists {
 // This call to .clone() only clones the Arc and bumps the
 // reference count. It does not clone the GigabyteMap.
 let glossary_for_child = glossary.clone();
 thread_handles.push(
 spawn(move || process_files(worklist, &glossary_for_child)))
 }
 ...
}
```

Nous avons changé le type de : pour exécuter l'analyse en parallèle, l'appelant doit passer dans un , un pointeur intelligent à un qui a été déplacé dans le tas, en utilisant

```
.glossary Arc<GigabyteMap> GigabyteMap Arc::new(giga_map)
```

Lorsque nous appelons , nous faisons une copie du pointeur intelligent, pas l'ensemble . Cela revient à incrémenter un nombre de références. `glossary.clone()` Arc GigabyteMap

Avec cette modification, le programme compile et s'exécute, car il ne dépend plus des durées de vie de référence. Tant qu'un thread possède un , il gardera la carte en vie, même si le thread parent se sauve plus tôt. Il n'y aura pas de course aux données, car les données dans un sont immuables. `Arc<GigabyteMap> Arc`

## Rayonne

La fonction de la bibliothèque standard est une primitive importante, mais elle n'est pas conçue spécifiquement pour le parallélisme fourche-jointure. De meilleures API de jointure de fourche ont été construites dessus. Par exemple, dans [le chapitre 2](#), nous avons utilisé la bibliothèque Crossbeam pour diviser certains travaux sur huit threads. Les filetages à lunette de Crossbeam prennent en charge le *parallelisme* fourche-jointure tout naturellement. `spawn`

La bibliothèque Rayon, de Niko Matsakis et Josh Stone, en est un autre exemple. Il offre deux façons d'exécuter des tâches simultanément :

```
use rayon::prelude::*;

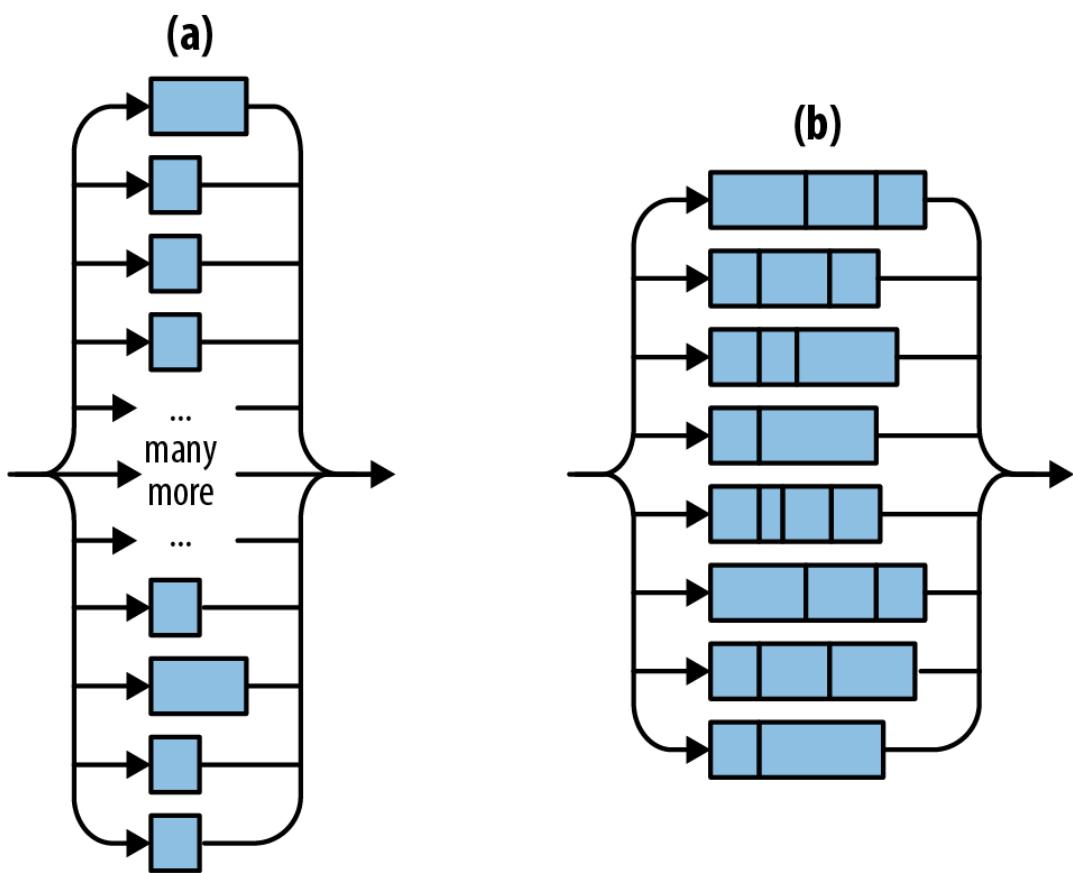
// "do 2 things in parallel"
let (v1, v2) = rayon::join(fn1, fn2);

// "do N things in parallel"
giant_vector.par_iter().for_each(|value| {
 do_thing_with_value(value);
});
```

`rayon::join(fn1, fn2)` appelle simplement les deux fonctions et renvoie les deux résultats. La méthode crée un , une valeur avec , , et d'autres méthodes, un peu comme un Rust . Dans les deux cas, Rayon utilise son propre pool de threads de travail pour répartir le travail lorsque cela est possible. Vous dites simplement à Rayon quelles tâches *peuvent* être effectuées en parallèle; Rayon gère les fils et distribue le travail du mieux qu'il peut. `.par_iter() ParallelIterator map filter Iterator`

Les diagrammes [de la figure 19-3](#) illustrent deux façons de penser l'appel. (a) La rayonne agit comme si elle produisait un fil par élément dans le vecteur. (b) Dans les coulisses, Rayon dispose d'un thread de travail par

œur de processeur, ce qui est plus efficace. Ce pool de threads de travail est partagé par tous les threads de votre programme. Lorsque des milliers de tâches arrivent à la fois, Rayon divise le travail. `giant_vector.par_iter().for_each(...)`



Graphique 19-3. Rayon en théorie et en pratique

Voici une version de l'utilisation de Rayon et une qui prend, plutôt que , juste un

```
:process_files_in_parallel process_file Vec<String> &str
```

```
use rayon::prelude::*;

fn process_files_in_parallel(filenames: Vec<String>, glossary: &Gigabyte)
 -> io::Result<()>
{
 filenames.par_iter()
 .map(|filename| process_file(filename, glossary))
 .reduce_with(|r1, r2| {
 if r1.is_err() { r1 } else { r2 }
 })
 .unwrap_or(Ok(()))
}
```

Ce code est plus court et moins délicat que la version utilisant . Regardons-le ligne par ligne: `std::thread::spawn`

- Tout d'abord, nous utilisons pour créer un itérateur parallèle. `filenames.par_iter()`
- Nous avons l'habitude d'appeler chaque nom de fichier. Cela produit une séquence de valeurs. `.map() process_file ParallelIterator io::Result<()>`
- Nous utilisons pour combiner les résultats. Ici, nous conservons la première erreur, le cas échéant, et jetons le reste. Si nous voulions accumuler toutes les erreurs, ou les imprimer, nous pourrions le faire ici. `.reduce_with()`  
La méthode est également pratique lorsque vous passez une fermeture qui renvoie une valeur utile sur le succès. Ensuite, vous pouvez passer une clôture qui sait comment combiner deux résultats de succès. `.reduce_with() .map() .reduce_with()`
- `reduce_with` renvoie un qui n'est que si était vide. Nous utilisons la méthode de 's pour obtenir le résultat dans ce cas. `Option None filenames Option .unwrap_or() Ok()`

Dans les coulisses, Rayon équilibre dynamiquement les charges de travail entre les threads, à l'aide d'une technique appelée *vol de travail*. Il fera généralement un meilleur travail en gardant tous les processeurs occupés que nous ne pouvons le faire en divisant manuellement le travail à l'avance, comme dans [« spawn and join »](#).

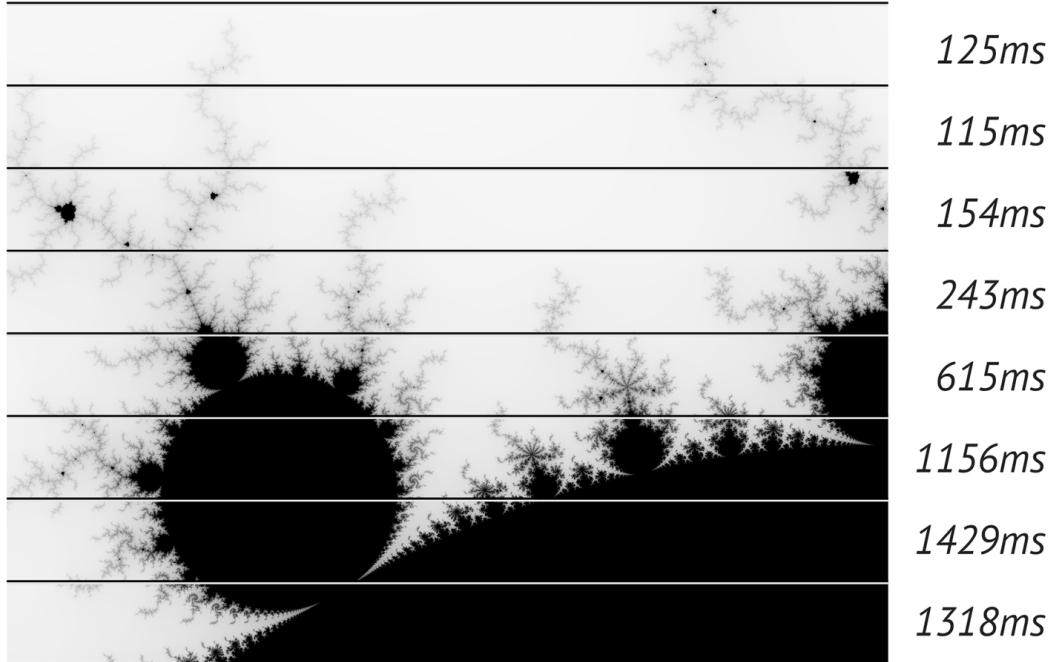
En prime, Rayon prend en charge le partage de références entre les threads. Tout traitement parallèle qui se produit dans les coulisses est garanti d'être terminé au moment du retour. Cela explique pourquoi nous avons pu passer à même si cette fermeture sera appelée sur plusieurs threads. `reduce_with glossary process_file`

(Incidemment, ce n'est pas un hasard si nous avons utilisé une méthode et une méthode. Le modèle de programmation MapReduce, popularisé par Google et Apache Hadoop, a beaucoup en commun avec le fork-join. Cela peut être considéré comme une approche de jointure de fourche pour interroger des données distribuées.) `map reduce`

## Revisiter l'ensemble Mandelbrot

Au [chapitre 2](#), nous avons utilisé la simultanéité de jointure de fourche pour rendre l'ensemble de Mandelbrot. Cela a rendu le rendu quatre fois plus rapide - impressionnant, mais pas aussi impressionnant qu'il pourrait l'être, étant donné que nous avons fait en sorte que le programme génère huit threads de travail et l'exécute sur une machine à huit cœurs!

Le problème est que nous n'avons pas réparti la charge de travail uniformément. Calculer un pixel de l'image revient à exécuter une boucle (voir « [Qu'est réellement l'ensemble de Mandelbrot »\). Il s'avère que les parties gris pâle de l'image, où la boucle se ferme rapidement, sont beaucoup plus rapides à rendre que les parties noires, où la boucle exécute les 255 itérations complètes. Ainsi, bien que nous divisions la zone en bandes horizontales de taille égale, nous créons des charges de travail inégales, comme \[le montre la figure 19-4\]\(#\).](#)



Graphique 19-4. Répartition inégale du travail dans le programme Mandelbrot

Ceci est facile à réparer à l'aide de Rayon. Nous pouvons simplement lancer une tâche parallèle pour chaque ligne de pixels dans la sortie. Cela crée plusieurs centaines de tâches que Rayon peut répartir sur ses threads. Grâce au vol de travail, peu importe que les tâches varient en taille. Rayon équilibrera le travail au fur et à mesure.

Voici le code. La première ligne et la dernière ligne font partie de la fonction que nous avons montrée dans « [A Concurrent Mandelbrot Program](#) », mais nous avons changé le code de rendu, qui est tout ce qui se trouve entre les deux: `main`

```
let mut pixels = vec![0; bounds.0 * bounds.1];

// Scope of slicing up `pixels` into horizontal bands.
{
 let bands: Vec<(usize, &mut [u8])> = pixels
 .chunks_mut(bounds.0)
 .enumerate()
 .collect();
```

```

bands.into_par_iter()
 .for_each(|(i, band)| {
 let top = i;
 let band_bounds = (bounds.0, 1);
 let band_upper_left = pixel_to_point(bounds, (0, top),
 upper_left, lower_right);
 let band_lower_right = pixel_to_point(bounds, (bounds.0, top),
 upper_left, lower_right);
 render(band, band_bounds, band_upper_left, band_lower_right)
 });
}

write_image(&args[1], &pixels, bounds).expect("error writing PNG file");

```

Tout d'abord, nous créons , la collection de tâches que nous allons transmettre à Rayon. Chaque tâche n'est qu'un tuple de type : le numéro de ligne, puisque le calcul l'exige, et la tranche de à remplir. Nous utilisons la méthode pour diviser le tampon d'image en lignes, pour attacher un numéro de ligne à chaque ligne et pour convertir toutes les paires nombre-tranche en un vecteur. (Nous avons besoin d'un vecteur car Rayon crée des itérateurs parallèles uniquement à partir de tableaux et de vecteurs.) bands (usize, &mut [u8]) pixels chunks\_mut enumerate collect

Ensute, nous nous transformons en un itérateur parallèle et utilisons la méthode pour dire à Rayon quel travail nous voulons faire. bands .for\_each()

Puisque nous utilisons Rayon, nous devons ajouter cette ligne à *main.rs* :

```
use rayon::prelude::*;


```

et ceci à *Cargo.toml*:

```
[dependencies]
rayon = "1"
```

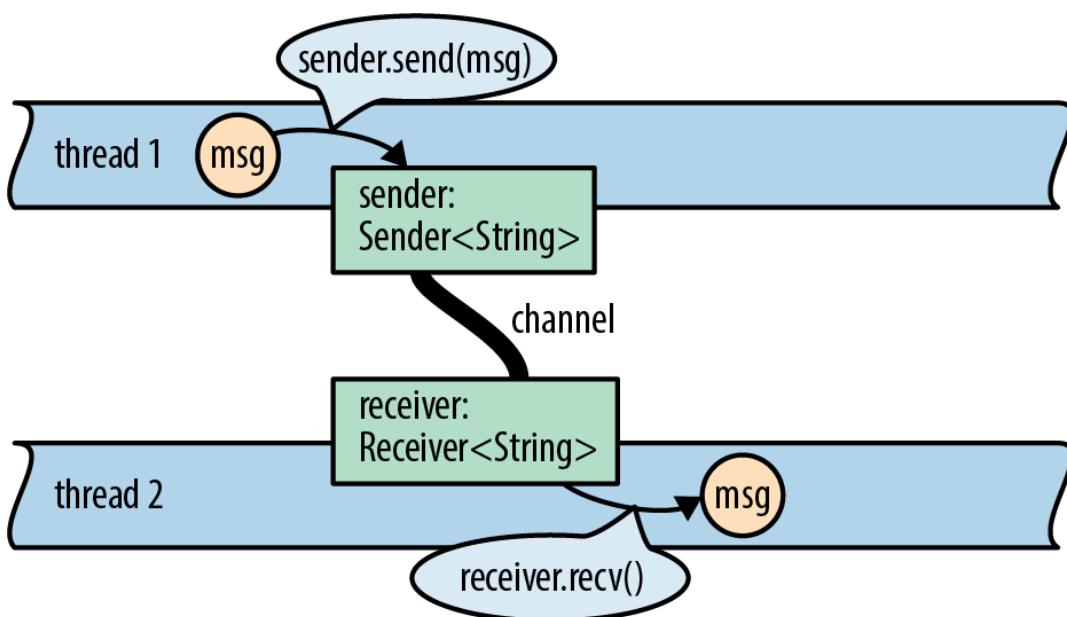
Avec ces changements, le programme utilise maintenant environ 7,75 coeurs sur une machine à 8 coeurs. C'est 75% plus rapide qu'avant, lorsque nous divisions le travail manuellement. Et le code est un peu plus court, reflétant les avantages de laisser une caisse faire un travail (répartition du travail) plutôt que de le faire nous-mêmes.

## Canaux

Un *canal* est un canal unidirectionnel permettant d'envoyer des valeurs d'un thread à un autre. En d'autres termes, il s'agit d'une file d'attente sécurisée.

La figure 19-5 illustre la façon dont les canaux sont utilisés. Ils sont quelque chose comme des tuyaux Unix: une extrémité est pour envoyer des données, et l'autre est pour recevoir. Les deux extrémités appartiennent généralement à deux threads différents. Mais alors que les tuyaux Unix sont destinés à l'envoi d'octets, les canaux sont destinés à l'envoi de valeurs Rust. `msg` une seule valeur dans le canal ; en supprime un. La propriété est transférée du thread d'envoi au thread de réception. Si le canal est vide, bloque jusqu'à ce qu'une valeur soit envoyée.

```
sender.send(item) receiver.recv() receiver.recv()
```



Graphique 19-5. Un canal pour s: la propriété de la chaîne msg est transférée du thread 1 au thread 2. `String`

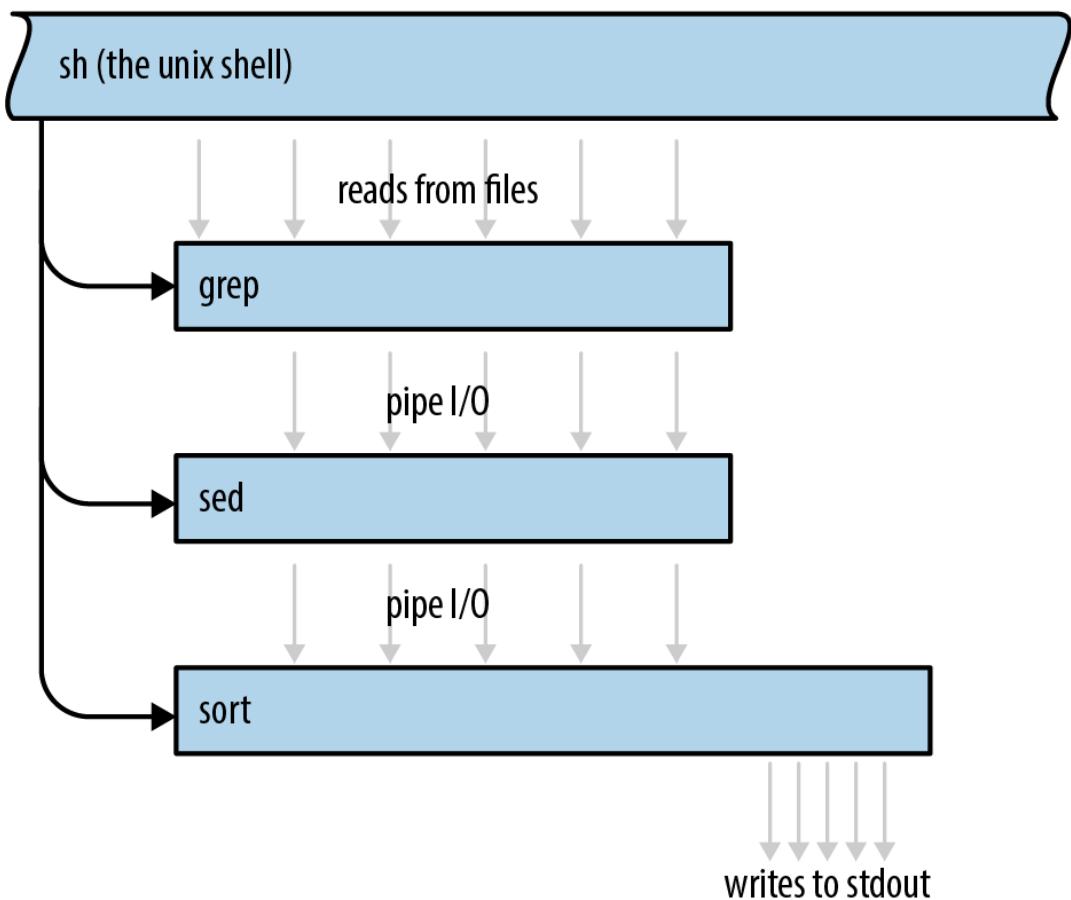
Avec les canaux, les threads peuvent communiquer en se transmettant des valeurs. C'est un moyen très simple pour les threads de travailler ensemble sans utiliser de verrouillage ou de mémoire partagée.

Ce n'est pas une technique nouvelle. Erlang a des processus isolés et des messages qui passent depuis 30 ans maintenant. Les tuyaux Unix existent depuis près de 50 ans. Nous avons tendance à penser que les tuyaux offrent de la flexibilité et de la composabilité, pas de la concurrence, mais en fait, ils font tout ce qui précède. Un exemple de pipeline Unix est illustré à la figure 19-6. Il est certainement possible que les trois programmes fonctionnent en même temps.

Les canaux de rouille sont plus rapides que les tuyaux Unix. L'envoi d'une valeur la déplace plutôt que de la copier, et les déplacements sont rapi-

des, même lorsque vous déplacez des structures de données contenant plusieurs mégaoctets de données.

```
grep -h '^=' *.txt | sed 's/=//g' | sort
```



Graphique 19-6. Exécution d'un pipeline Unix

## Envoi de valeurs

Au cours des prochaines sections, nous utiliserons des canaux pour créer un programme simultané qui crée un *index inversé*, l'un des ingrédients clés d'un moteur de recherche. Chaque moteur de recherche fonctionne sur une collection particulière de documents. L'*index inversé* est la base de données qui indique quels mots apparaissent où.

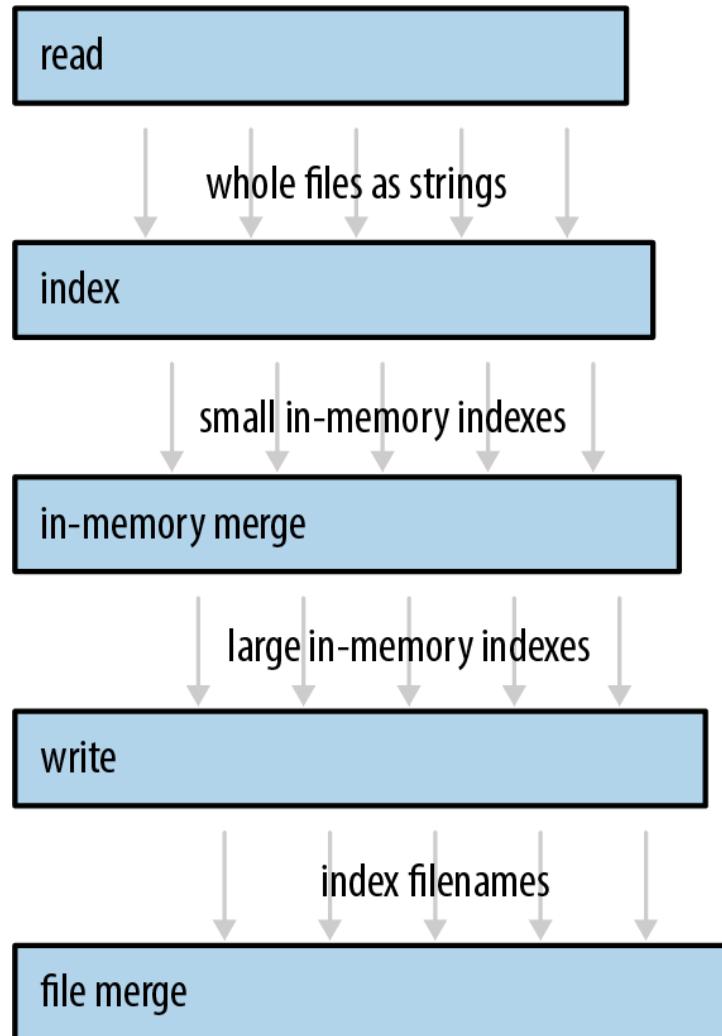
Nous allons montrer les parties du code qui ont à voir avec les threads et les canaux. Le [programme complet](#) est court, environ un millier de lignes de code en tout.

Notre programme est structuré comme un pipeline, comme le montre [la figure 19-7](#). Les pipelines ne sont qu'une des nombreuses façons d'utiliser les canaux (nous discuterons de quelques autres utilisations plus tard), mais ils constituent un moyen simple d'introduire la concurrence d'accès concurrentiel dans un programme monothread existant.

Nous utiliserons un total de cinq threads, chacun effectuant une tâche distincte. Chaque thread produit une sortie en continu pendant toute la durée de vie du programme. Le premier thread, par exemple, lit simple-

ment les documents sources du disque en mémoire, un par un. (Nous voulons un thread pour le faire parce que nous allons écrire le code le plus simple possible ici, en utilisant , qui est une API de blocage. Nous ne voulons pas que le processeur reste inactif chaque fois que le disque fonctionne.) La sortie de cette étape est longue d'un par document, de sorte que ce thread est connecté au thread suivant par un canal de

```
s. fs::read_to_string String String
```



Graphique 19-7. Le pipeline du générateur d'index, où les flèches représentent les valeurs envoyées via un canal d'un thread à un autre (les E/S disque ne sont pas affichées)

Notre programme commencera par générer le thread qui lit les fichiers. Supposons est un , un vecteur de noms de fichiers. Le code pour démarrer notre thread de lecture de fichiers ressemble à ceci

```

: documents Vec<PathBuf>

use std::fs;
use std::sync::mpsc;

let (sender, receiver) = mpsc::channel();

let handle = thread::spawn(move || {
 for filename in documents {
 let text = fs::read_to_string(filename)?;
 }
});
```

```

 if sender.send(text).is_err() {
 break;
 }
 }
 Ok(())
);

```

Les canaux font partie du module. Nous expliquerons ce que ce nom signifie plus tard; Tout d'abord, regardons comment ce code fonctionne.

Nous commençons par créer un canal : `std::sync::mpsc`

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

La fonction renvoie une paire de valeurs : un expéditeur et un récepteur. La structure de données de file d'attente sous-jacente est un détail d'implémentation que la bibliothèque standard n'expose pas. `channel`

Les canaux sont tapés. Nous allons utiliser ce canal pour envoyer le texte de chaque fichier, nous avons donc un de type et un de type . Nous aurions pu demander explicitement un canal de chaînes, en écrivant . Au lieu de cela, nous laissons l'inférence de type Rust le

```
comprendre. sender Sender<String> receiver Receiver<String> m
psc::channel::<String>()
```

```
let handle = thread::spawn(move || {
```

Comme précédemment, nous utilisons pour démarrer un thread. La propriété (mais non) est transférée au nouveau thread via cette fermeture. `std::thread::spawn` `sender` `receiver` `move`

Les quelques lignes de code suivantes lisent simplement les fichiers du disque:

```
for filename in documents {
 let text = fs::read_to_string(filename)?;
```

Après avoir lu avec succès un fichier, nous envoyons son texte dans le canal:

```

if sender.send(text).is_err() {
 break;
}
}
```

```
sender.send(text) déplace la valeur dans le canal. En fin de compte, il sera à nouveau déplacé vers celui qui recevra la valeur. Qu'elle contienne 10 lignes de texte ou 10 mégaoctets, cette opération copie trois mots machine (la taille d'une struct), et l'appel correspondant copiera également trois mots machine. text text String receiver.recv()
```

Les méthodes et renvoient toutes deux des s, mais ces méthodes échouent uniquement si l'autre extrémité du canal a été supprimée. Un appel échoue si le a été supprimé, car sinon la valeur resterait dans le canal pour toujours: sans un , il n'y a aucun moyen pour un thread de le recevoir. De même, un appel échoue s'il n'y a pas de valeurs en attente dans le canal et que le a été supprimé, sinon attendrait éternellement: sans un , il n'y a aucun moyen pour un thread d'envoyer la valeur suivante. Laisser tomber votre extrémité d'un canal est la façon normale de « raccrocher », de fermer la connexion lorsque vous avez

```
terminé. send recv Result send Receiver Receiver recv Sender r
ecv Sender
```

Dans notre code, n'échouera que si le thread du récepteur s'est arrêté prématurément. Ceci est typique pour le code qui utilise des canaux. Que cela se soit produit délibérément ou en raison d'une erreur, il est normal que notre fil de discussion pour le lecteur s'éteigne tranquillement. `sender.send(text)`

Lorsque cela se produit, ou que le fil de discussion termine la lecture de tous les documents, il renvoie : `Ok(())`

```
Ok(())
});
```

Notez que cette fermeture renvoie un fichier . Si le thread rencontre une erreur d'E/S, il se ferme immédiatement et l'erreur est stockée dans le fichier . `Result JoinHandle`

Bien sûr, comme tout autre langage de programmation, Rust admet de nombreuses autres possibilités en matière de gestion des erreurs.

Lorsqu'une erreur se produit, nous pouvons simplement l'imprimer en utilisant et passer au fichier suivant. Nous pourrions transmettre les erreurs via le même canal que celui que nous utilisons pour les données, ce qui en fait un canal de s, ou créer un deuxième canal uniquement pour les erreurs. L'approche que nous avons choisie ici est à la fois légère et responsable : nous utilisons l'opérateur, il n'y a donc pas un tas de code standard, ni même un explicite comme vous pourriez le voir en Java, et

```
pourtant les erreurs ne passeront pas
silencieusement. println! Result ? try/catch
```

Pour plus de commodité, notre programme enveloppe tout ce code dans une fonction qui renvoie à la fois le (que nous n'avons pas encore utilisé) et le nouveau thread : `receiver JoinHandle`

```
fn start_file_reader_thread(documents: Vec<PathBuf>)
 -> (mpsc::Receiver<String>, thread::JoinHandle<io::Result<()>>)
{
 let (sender, receiver) = mpsc::channel();

 let handle = thread::spawn(move || {
 ...
 });

 (receiver, handle)
}
```

Notez que cette fonction lance le nouveau thread et le renvoie immédiatement. Nous allons écrire une fonction comme celle-ci pour chaque étape de notre pipeline.

## Réception des valeurs

Maintenant, nous avons un thread exécutant une boucle qui envoie des valeurs. On peut engendrer un second thread exécutant une boucle qui appelle : `receiver.recv()`

```
while let Ok(text) = receiver.recv() {
 do_something_with(text);
}
```

Mais si s sont itérables, il y a donc une meilleure façon d'écrire ceci: Receiver

```
for text in receiver {
 do_something_with(text);
}
```

Ces deux boucles sont équivalentes. Quoi qu'il en soit, nous l'écrivons, si le canal se trouve vide lorsque le contrôle atteint le haut de la boucle, le thread récepteur se bloquera jusqu'à ce qu'un autre thread envoie une valeur. La boucle se ferme normalement lorsque le canal est vide et qu'il a été abandonné. Dans notre programme, cela se produit naturellement

lorsque le fil du lecteur se ferme. Ce thread exécute une fermeture qui possède la variable ; lorsque la fermeture sort, est abandonné. Sender sender sender

Nous pouvons maintenant écrire du code pour la deuxième étape du pipeline :

```
fn start_file_indexing_thread(texts: mpsc::Receiver<String>)
 -> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
{
 let (sender, receiver) = mpsc::channel();

 let handle = thread::spawn(move || {
 for (doc_id, text) in texts.into_iter().enumerate() {
 let index = InMemoryIndex::from_single_document(doc_id, text);
 if sender.send(index).is_err() {
 break;
 }
 }
 });
}

(receiver, handle)
```

Cette fonction génère un thread qui reçoit des valeurs d'un canal () et envoie des valeurs à un autre canal (/). Le travail de ce thread consiste à prendre chacun des fichiers chargés dans la première étape et à transformer chaque document en un petit index inversé en mémoire. String texts InMemoryIndex sender receiver

La boucle principale de ce thread est simple. Tout le travail d'indexation d'un document se fait par la fonction . Nous n'afficherons pas son code source ici, mais il divise la chaîne d'entrée aux limites des mots, puis produit une carte des mots aux listes de positions. InMemoryIndex:: from\_single\_document

Cette étape n'effectue pas d'E/S, elle n'a donc pas à traiter avec s. Au lieu d'un , il renvoie . io::Error io::Result<()> ()

## Exécution du pipeline

Les trois étapes restantes sont de conception similaire. Chacun consomme un créé par l'étape précédente. Notre objectif pour le reste du pipeline est de fusionner tous les petits index en un seul grand fichier d'index sur disque. Le moyen le plus rapide que nous avons trouvé pour le faire est

en trois étapes. Nous n'afficherons pas le code ici, juste les signatures de type de ces trois fonctions. La source complète est en ligne. [Receiver](#)

Tout d'abord, nous fusionnons les index en mémoire jusqu'à ce qu'ils deviennent encombrants (étape 3) :

```
fn start_in_memory_merge_thread(file_indexes: mpsc::Receiver<InMemoryIndex>
-> (mpsc::Receiver<InMemoryIndex>, thread::JoinHandle<()>)
```

Nous écrivons ces grands index sur disque (étape 4) :

```
fn start_index_writer_thread(big_indexes: mpsc::Receiver<InMemoryIndex>,
 output_dir: &Path)
-> (mpsc::Receiver<PathBuf>, thread::JoinHandle<io::Result<()>>)
```

Enfin, si nous avons plusieurs fichiers volumineux, nous les fusionnons à l'aide d'un algorithme de fusion basé sur des fichiers (étape 5):

```
fn merge_index_files(files: mpsc::Receiver<PathBuf>, output_dir: &Path)
-> io::Result<()>
```

Cette dernière étape ne renvoie pas un , car c'est la fin de la ligne. Il produit un seul fichier de sortie sur le disque. Il ne renvoie pas un , car nous ne prenons pas la peine de générer un fil pour cette étape. Le travail se fait sur le fil de l'appelant. [Receiver JoinHandle](#)

Nous arrivons maintenant au code qui lance les threads et vérifie les erreurs:

```
fn run_pipeline(documents: Vec<PathBuf>, output_dir: PathBuf)
-> io::Result<()>
{
 // Launch all five stages of the pipeline.
 let (texts, h1) = start_file_reader_thread(documents);
 let (pints, h2) = start_file_indexing_thread(texts);
 let (gallons, h3) = start_in_memory_merge_thread(pints);
 let (files, h4) = start_index_writer_thread(gallons, &output_dir);
 let result = merge_index_files(files, &output_dir);

 // Wait for threads to finish, holding on to any errors that they encounter.
 let r1 = h1.join().unwrap();
 h2.join().unwrap();
 h3.join().unwrap();
 let r4 = h4.join().unwrap();

 // Return the first error encountered, if any.
```

```

 // (As it happens, h2 and h3 can't fail: those threads
 // are pure in-memory data processing.)
 r1?;
 r4?;
 result
}

```

Comme précédemment, nous avons l'habitude de propager explicitement les paniques des threads enfants vers le thread principal. La seule autre chose inhabituelle ici est qu'au lieu d'utiliser tout de suite, nous mettons de côté les valeurs jusqu'à ce que nous ayons rejoint les quatre threads.

```
.join().unwrap() ? io::Result
```

Ce pipeline est 40 % plus rapide que l'équivalent monothread. Ce n'est pas mal pour un après-midi de travail, mais dérisoire à côté du coup de pouce de 675% que nous avons obtenu pour le programme Mandelbrot. Nous n'avons clairement pas saturé la capacité d'E/S du système ni tous les cœurs de processeur. Que se passe-t-il?

Les pipelines sont comme des lignes d'assemblage dans une usine de fabrication : les performances sont limitées par le débit de l'étape la plus lente. Une toute nouvelle chaîne de montage non réglée peut être aussi lente que la production unitaire, mais les chaînes de montage récompensent le réglage ciblé. Dans notre cas, la mesure montre que la deuxième étape est le goulet d'étranglement. Notre thread d'indexation utilise et , il passe donc beaucoup de temps à fouiller dans les tables Unicode. Les autres étapes en aval de l'indexation passent la plupart de leur temps à dormir dans , en attendant

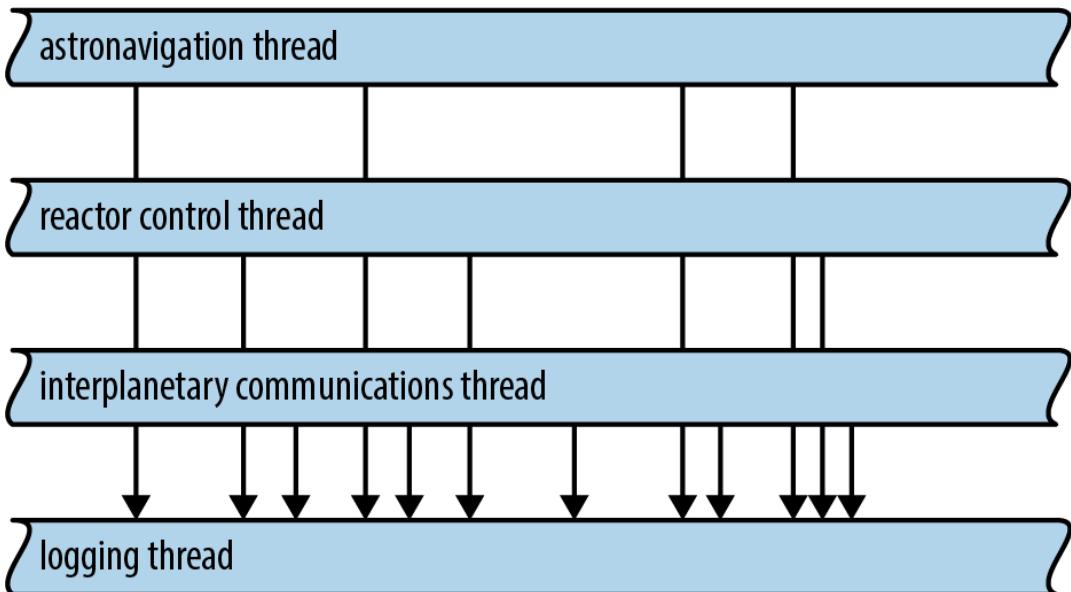
```
l'entrée. .to_lowercase() .is_alphanumeric() Receiver::recv
```

Cela signifie que nous devrions être en mesure d'aller plus vite. Au fur et à mesure que nous nous attaquerons aux goulets d'étranglement, le degré de parallélisme augmentera. Maintenant que vous savez comment utiliser les canaux et que notre programme est composé de morceaux de code isolés, il est facile de voir des moyens de résoudre ce premier goulet d'étranglement. Nous pourrions optimiser manuellement le code pour la deuxième étape, comme n'importe quel autre code; diviser l'œuvre en deux étapes ou plus; ou exécutez plusieurs threads d'indexation de fichiers à la fois.

## Caractéristiques et performances du canal

La partie de signifie *multiproducteur, mono-consommateur*, une description laconique du type de communication fourni par les canaux de

Les canaux de notre exemple de programme transportent des valeurs d'un seul expéditeur à un seul récepteur. C'est un cas assez courant. Mais les canaux Rust prennent également en charge plusieurs expéditeurs, au cas où vous auriez besoin, par exemple, d'un seul thread qui gère les demandes de nombreux threads clients, comme illustré à [la figure 19-8](#).



Graphique 19-8. Un canal unique recevant les demandes de nombreux expéditeurs

`Sender<T>` implémente le trait. Pour obtenir un canal avec plusieurs expéditeurs, créez simplement un canal normal et clonez l'expéditeur autant de fois que vous le souhaitez. Vous pouvez déplacer chaque valeur vers un thread différent. `Clone Sender`

A ne peut pas être cloné, donc si vous avez besoin de plusieurs threads recevant des valeurs du même canal, vous avez besoin d'un fichier . Nous montrerons comment le faire plus loin dans ce chapitre. `Receiver<T> Mutex`

Les canaux de rouille sont soigneusement optimisés. Lorsqu'un canal est créé pour la première fois, Rust utilise une implémentation de file d'attente spéciale « one-shot ». Si vous n'envoyez qu'un seul objet via le canal, la surcharge est minime. Si vous envoyez une deuxième valeur, Rust bascule vers une autre implémentation de file d'attente. Il s'installe à long terme, vraiment, préparant le canal à transférer de nombreuses valeurs tout en minimisant les frais généraux d'allocation. Et si vous clonez le , Rust doit se rabattre sur une autre implémentation, qui est sûre lorsque plusieurs threads essaient d'envoyer des valeurs à la fois. Mais même la plus lente de ces trois implémentations est une file d'attente sans verrouillage, de sorte que l'envoi ou la réception d'une valeur est tout au plus quelques opérations atomiques et une allocation de tas, plus le dé-

placement lui-même. Les appels système ne sont nécessaires que lorsque la file d'attente est vide et que le thread récepteur doit donc se mettre en veille. Dans ce cas, bien sûr, le trafic via votre canal n'est pas maximisé de toute façon. `Sender`

Malgré tout ce travail d'optimisation, il y a une erreur que les applications peuvent facilement commettre en ce qui concerne les performances des canaux : envoyer des valeurs plus rapidement qu'elles ne peuvent être reçues et traitées. Cela entraîne un arriéré de valeurs sans cesse croissant à accumuler dans le canal. Par exemple, dans notre programme, nous avons constaté que le thread du lecteur de fichiers (étape 1) pouvait charger les fichiers beaucoup plus rapidement que le thread d'indexation de fichiers (étape 2) pouvait les indexer. Le résultat est que des centaines de mégaoctets de données brutes seraient lus à partir du disque et stockés dans la file d'attente à la fois.

Ce genre de mauvaise conduite coûte de la mémoire et nuit à la localité.pire encore, le thread d'envoi continue de fonctionner, utilisant le processeur et d'autres ressources système pour envoyer toujours plus de valeurs au moment où ces ressources sont les plus nécessaires à la réception.

Ici, Rust reprend une page des tuyaux Unix. Unix utilise une astuce élégante pour fournir une *certaine contre-pression* afin que les expéditeurs rapides soient obligés de ralentir: chaque tuyau d'un système Unix a une taille fixe, et si un processus essaie d'écrire sur un tuyau qui est momentanément plein, le système bloque simplement ce processus jusqu'à ce qu'il y ait de la place dans le tuyau. L'équivalent Rust est appelé *canal synchrone* :

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

let (sender, receiver) = mpsc::sync_channel(1000);
```

Un canal synchrone est exactement comme un canal normal, sauf que lorsque vous le créez, vous spécifiez le nombre de valeurs qu'il peut contenir. Pour un canal synchrone, il s'agit potentiellement d'une opération de blocage. Après tout, l'idée est que le blocage n'est pas toujours mauvais. Dans notre exemple de programme, le remplacement de l'entrée par un avec de la place pour 32 valeurs réduit l'utilisation de la mémoire des deux tiers sur notre ensemble de données de référence, sans diminuer le débit. `sender.send(value)` channel `start_file_reader_thread` `sync_channel`

# Sécurité des threads : envoi et synchronisation

Jusqu'à présent, nous avons agi comme si toutes les valeurs pouvaient être librement déplacées et partagées à travers les fils. C'est en grande partie vrai, mais l'histoire complète de la sécurité du fil de Rust repose sur deux traits intégrés, et . `std::marker::Send` `std::marker::Sync`

- Les types qui implémentent peuvent être transmis en toute sécurité par valeur à un autre thread. Ils peuvent être déplacés d'un thread à l'autre. `Send`
- Les types qui implémentent peuvent passer en toute sécurité par non-référence à un autre thread. Ils peuvent être partagés entre les threads. `Sync` `mut`

Par *sûr* ici, nous entendons la même chose que nous voulons toujours dire: exempt de courses de données et d'autres comportements indéfinis.

Par exemple, dans l'exemple , nous avons utilisé une fermeture pour passer un du thread parent à chaque thread enfant. Nous ne l'avons pas signalé à l'époque, mais cela signifie que le vecteur et ses chaînes sont alloués dans le thread parent, mais libérés dans le thread enfant. Le fait que l'implémentation soit une promesse d'API que c'est OK: l'allocateur utilisé en interne par et est thread-

```
safe. process_files_in_parallel Vec<String> Vec<String> Send v
ec String
```

(Si vous deviez écrire vos propres types avec des allocateurs rapides mais non thread-safe, vous devrez les implémenter en utilisant des types qui ne le sont pas, tels que des pointeurs dangereux. Rust en déduirait alors que votre et les types ne le sont pas et les limiterait à une utilisation à filetage unique. Mais c'est un cas

```
rare.) vec String Send NonThreadSafeVec NonThreadSafeString s
end
```

Comme [l'illustre la figure 19-9](#), la plupart des types sont à la fois et . Vous n'avez même pas besoin d'utiliser pour obtenir ces traits sur les structs et les enums dans votre programme. Rust le fait pour vous. Une struct ou enum est si ses champs sont , et si ses champs sont . `Send Sync` #  
[derive] `Send Send Sync Sync`

Certains types sont , mais pas . Ceci est généralement intentionnel, comme dans le cas de , où il garantit que l'extrémité réceptrice d'un canal est utilisée par un seul thread à la fois. `Send Sync mpsc::Receiver mpsc`

Les quelques types qui ne sont ni ne sont principalement ceux qui utilisent la mutabilité d'une manière qui n'est pas thread-safe. Par exemple, considérez , le type de pointeurs intelligents de comptage de références. Send Sync std::rc::Rc<T>

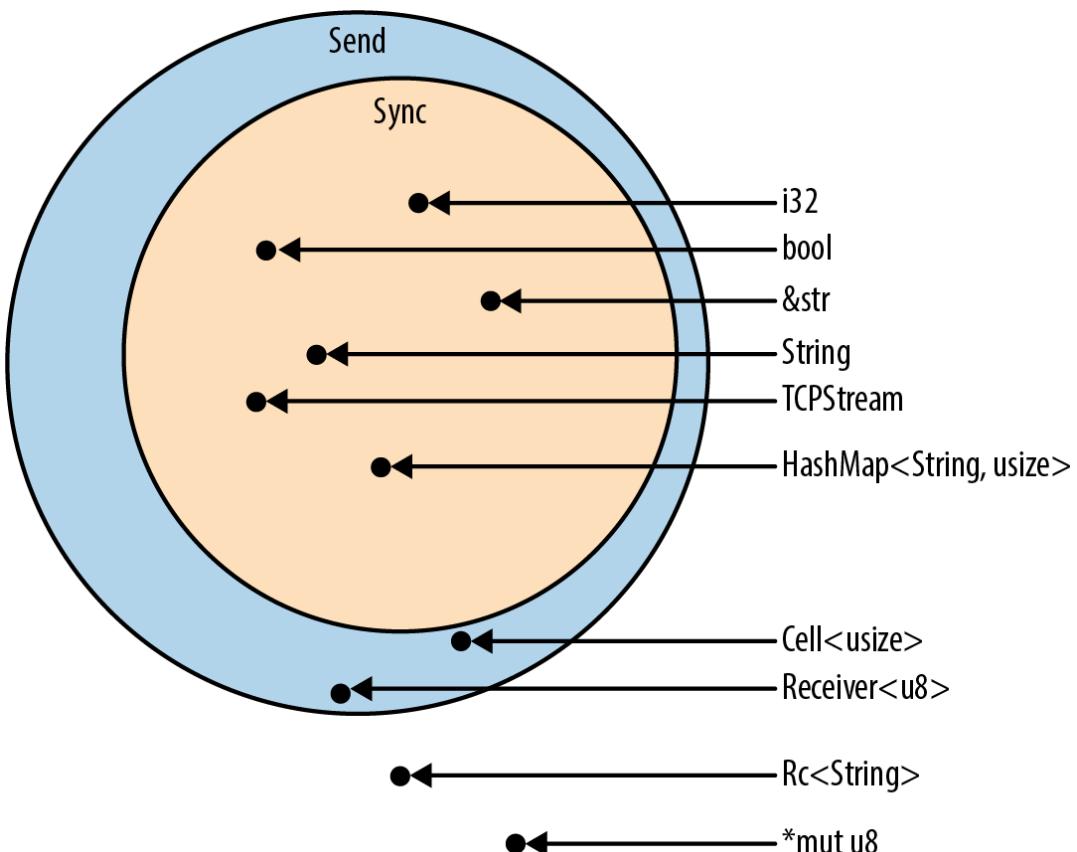
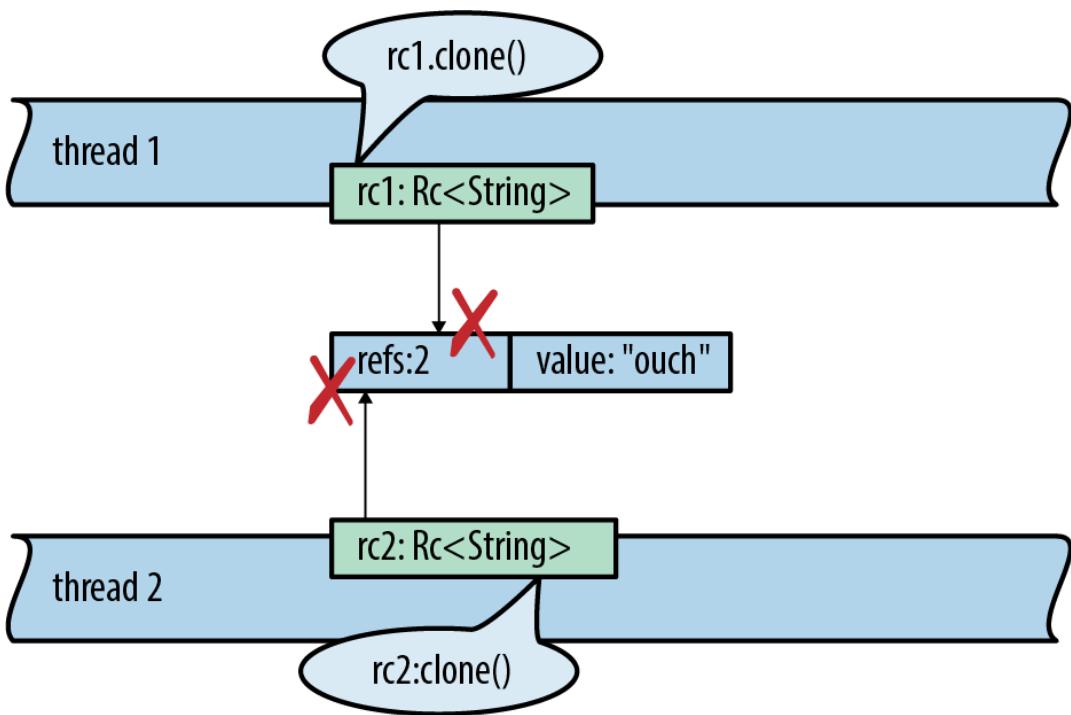


Figure 19-9. et types Send Sync

Que se passerait-il si , permettant aux threads de partager un seul via des références partagées ? Si les deux threads essaient de cloner le en même temps, comme illustré à la [figure 19-10](#), nous avons une course de données car les deux threads incrémentent le nombre de références partagées. Le nombre de références peut devenir inexact, ce qui conduit à un comportement non défini d'utilisation après ou de double libre plus tard. `Rc<String> Sync Rc Rc`



Graphique 19-10. Pourquoi ni l'un ni l'autre `Rc<String>` Sync Send

Bien sûr, Rust empêche cela. Voici le code pour configurer cette course aux données :

```
use std::thread;
use std::rc::Rc;

fn main() {
 let rc1 = Rc::new("ouch".to_string());
 let rc2 = rc1.clone();
 thread::spawn(move || { // error
 rc2.clone();
 });
 rc1.clone();
}
```

Rust refuse de le compiler, donnant un message d'erreur détaillé:

```
error: `Rc<String>` cannot be sent between threads safely
|
10 | thread::spawn(move || { // error
| ^^^^^ `Rc<String>` cannot be sent between threads safely
|
= help: the trait `std::marker::Send` is not implemented for `Rc<String>`
= note: required because it appears within the type `[closure@...,...]`
= note: required by `std::thread::spawn`
```

Vous pouvez maintenant voir comment et aider Rust à appliquer la sécurité des fils. Ils apparaissent sous forme de limites dans la signature de type des fonctions qui transfèrent des données au-delà des limites de

thread. Lorsque vous êtes un thread, la fermeture que vous passez doit être , ce qui signifie que toutes les valeurs qu'il contient doivent être . De même, si vous souhaitez envoyer des valeurs via un canal à un autre thread, les valeurs doivent être . Send Sync spawn Send Send Send

## Tuyaute de presque n'importe quel itérateur vers un canal

Notre générateur d'index inversé est construit comme un pipeline. Le code est assez clair, mais il nous oblige à configurer manuellement des canaux et à lancer des threads. En revanche, les pipelines d'itérateurs que nous avons construits dans [le chapitre 15](#) semblaient contenir beaucoup plus de travail en quelques lignes de code. Pouvons-nous construire quelque chose comme ça pour les pipelines de thread ?

En fait, ce serait bien si nous pouvions unifier les pipelines d'itérateurs et les pipelines de thread. Ensuite, notre générateur d'index pourrait être écrit en tant que pipeline d'itérateur. Cela pourrait commencer comme ceci:

```
documents.into_iter()
 .map(read_whole_file)
 .errors_to(error_sender) // filter out error results
 .off_thread() // spawn a thread for the above work
 .map(make_single_file_index)
 .off_thread() // spawn another thread for stage 2
 ...
 ...
```

Les traits nous permettent d'ajouter des méthodes aux types de bibliothèque standard, de sorte que nous pouvons réellement le faire. Nous commençons par écrire un trait qui déclare la méthode que nous voulons:

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

pub trait OffThreadExt: Iterator {
 /// Transform this iterator into an off-thread iterator: the
 /// `next()` calls happen on a separate worker thread, so the
 /// iterator and the body of your loop run concurrently.
 fn off_thread(self) -> mpsc::IntoIter<Self::Item>;
}
```

Ensuite, nous implémentons ce trait pour les types d'itérateurs. Cela aide qui est déjà itérable: `mpsc::Receiver`

```

use std::thread;

impl<T> OffThreadExt for T
 where T: Iterator + Send + 'static,
 T::Item: Send + 'static
{
 fn off_thread(self) -> mpsc::IntoIter<Self::Item> {
 // Create a channel to transfer items from the worker thread.
 let (sender, receiver) = mpsc::sync_channel(1024);

 // Move this iterator to a new worker thread and run it there.
 thread::spawn(move || {
 for item in self {
 if sender.send(item).is_err() {
 break;
 }
 }
 });
 }

 // Return an iterator that pulls values from the channel.
 receiver.into_iter()
}
}

```

La clause de ce code a été déterminée via un processus similaire à celui décrit dans [« Reverse-Engineering Bounds »](#). Au début, nous avions juste ceci: where

```
impl<T> OffThreadExt for T
```

C'est-à-dire que nous voulions que la mise en œuvre fonctionne pour tous les itérateurs. Rust n'avait rien de tout cela. Étant donné que nous utilisons pour déplacer un itérateur de type vers un nouveau thread, nous devons spécifier . Étant donné que nous renvoyons les éléments via un canal, nous devons spécifier . Avec ces changements, Rust était satisfait.

```
spawn T T: Iterator + Send + 'static T::Item: Send + 'static
```

C'est le caractère de Rust en un mot : nous sommes libres d'ajouter un outil d'alimentation de concurrence à presque tous les itérateurs de la langue, mais non sans d'abord comprendre et documenter les restrictions qui le rendent sûr à utiliser.

## Au-delà des pipelines

Dans cette section, nous avons utilisé les pipelines comme exemples, car les pipelines sont un moyen simple et évident d'utiliser les canaux. Tout le monde les comprend. Ils sont concrets, pratiques et déterministes. Cependant, les canaux sont utiles pour plus que de simples pipelines. Ils constituent également un moyen rapide et facile d'offrir n'importe quel service asynchrone à d'autres threads dans le même processus.

Par exemple, supposons que vous souhaitiez effectuer la journalisation sur son propre thread, comme dans [la figure 19-8](#). D'autres threads peuvent envoyer des messages de journalisation au thread de journalisation via un canal ; puisque vous pouvez cloner le canal , de nombreux threads clients peuvent avoir des expéditeurs qui expédient les messages de journalisation au même thread de journalisation. Sender

L'exécution d'un service comme la journalisation sur son propre thread présente des avantages. Le thread de journalisation peut faire pivoter les fichiers journaux chaque fois qu'il en a besoin. Il n'a pas besoin de faire de coordination sophistiquée avec les autres fils. Ces fils ne seront pas bloqués. Les messages s'accumuleront sans danger dans le canal pendant un moment jusqu'à ce que le fil de journalisation se remette au travail.

Les canaux peuvent également être utilisés dans les cas où un thread envoie une demande à un autre thread et a besoin d'obtenir une sorte de réponse. La requête du premier thread peut être une struct ou un tuple qui inclut un , une sorte d'enveloppe auto-adressée que le second thread utilise pour envoyer sa réponse. Cela ne signifie pas que l'interaction doit être synchrone. Le premier thread décide s'il faut bloquer et attendre la réponse ou utiliser la méthode pour l'interroger. Sender `.try_recv()`

Les outils que nous avons présentés jusqu'à présent – fork-join pour le calcul hautement parallèle, canaux pour connecter des composants lâches – sont suffisants pour un large éventail d'applications. Mais nous n'avons pas fini.

## État mutable partagé

Dans les mois qui ont suivi la publication de la caisse dans le [chapitre 8](#), votre logiciel de simulation de fougère a vraiment décollé. Maintenant, vous créez un jeu de stratégie multijoueur en temps réel dans lequel huit joueurs s'affrontent pour faire pousser principalement des fougères d'époque authentiques dans un paysage jurassique simulé. Le serveur de ce jeu est une application massivement parallèle, avec des requêtes affluant sur de nombreux threads. Comment ces fils peuvent-ils se coordonner

pour commencer une partie dès que huit joueurs sont disponibles ?

`fern_sim`

Le problème à résoudre ici est que de nombreux threads ont besoin d'accéder à une liste partagée de joueurs qui attendent de rejoindre un jeu. Ces données sont nécessairement à la fois mutables et partagées sur tous les threads. Si Rust n'a pas d'état mutable partagé, où cela nous laisse-t-il ?

Vous pouvez résoudre ce problème en créant un nouveau thread dont tout le travail consiste à gérer cette liste. D'autres fils communiqueraient avec elle via des canaux. Bien sûr, cela coûte un thread, qui a une certaine surcharge du système d'exploitation.

Une autre option consiste à utiliser les outils fournis par Rust pour partager en toute sécurité des données mutables. De telles choses existent. Ce sont des primitives de bas niveau qui seront familières à tout programmeur système qui a travaillé avec des threads. Dans cette section, nous couvrirons les mutex, les verrous en lecture/écriture, les variables de condition et les entiers atomiques. Enfin, nous montrerons comment implémenter des variables globales mutables dans Rust.

## Qu'est-ce qu'un Mutex?

Un *mutex* (ou *verrou*) est utilisé pour forcer plusieurs threads à se relayer lors de l'accès à certaines données. Nous présenterons les mutex de Rust dans la section suivante. Tout d'abord, il est logique de se rappeler à quoi ressemblent les mutex dans d'autres langues. Une simple utilisation d'un mutex en C++ peut ressembler à ceci :

```
// C++ code, not Rust
void FernEmpireApp::JoinWaitingList(PlayerId player) {
 mutex.Acquire();

 waitingList.push_back(player);

 // Start a game if we have enough players waiting.
 if (waitingList.size() >= GAME_SIZE) {
 vector<PlayerId> players;
 waitingList.swap(players);
 StartGame(players);
 }

 mutex.Release();
}
```

Les appels et marquent le début et la fin d'une *section critique* dans ce code. Pour chacun d'un programme, un seul thread peut être exécuté dans une section critique à la fois. Si un thread se trouve dans une section critique, tous les autres threads qui appellent seront bloqués jusqu'à ce que le premier thread atteigne

```
.mutex.Acquire() mutex.Release() mutex mutex.Acquire() mutex
x.Release()
```

Nous disons que le mutex *protège* les données: dans ce cas, protège . Il est de la responsabilité du programmeur, cependant, de s'assurer que chaque thread acquiert toujours le mutex avant d'accéder aux données, et le libère après. `mutex waitingList`

Les mutex sont utiles pour plusieurs raisons :

- Ils empêchent les *courses de données*, des situations où les threads de course lisent et écrivent simultanément la même mémoire. Les courses de données sont un comportement indéfini en C++ et Go. Les langages managés comme Java et C# promettent de ne pas planter, mais les résultats des courses de données sont toujours (pour résumer) absurdes.
- Même si les courses de données n'existaient pas, même si toutes les lectures et écritures se produisaient une par une dans l'ordre du programme, sans mutex, les actions des différents threads pouvaient s'entrelier de manière arbitraire. Imaginez essayer d'écrire du code qui fonctionne même si d'autres threads modifient ses données pendant son exécution. Imaginez que vous essayez de le déboguer. Ce serait comme si votre programme était hanté.
- Mutexes prend en charge la programmation avec des *invariants*, des règles sur les données protégées qui sont vraies par construction lorsque vous les configurez et maintenues par chaque section critique.

Bien sûr, tout cela est vraiment la même raison: des conditions de course incontrôlées rendent la programmation insoluble. Les mutex apportent un peu d'ordre au chaos (mais pas autant d'ordre que les canaux ou le fork-join).

Cependant, dans la plupart des langues, les mutex sont très faciles à gâcher. En C++, comme dans la plupart des langages, les données et le verrou sont des objets distincts. Idéalement, les commentaires expliquent que chaque thread doit acquérir le mutex avant de toucher les données :

```
class FernEmpireApp {
 ...
```

```

private:
 // List of players waiting to join a game. Protected by `mutex`.
 vector<PlayerId> waitingList;

 // Lock to acquire before reading or writing `waitingList`.
 Mutex mutex;
 ...
};

}

```

Mais même avec de si beaux commentaires, le compilateur ne peut pas imposer un accès sécurisé ici. Lorsqu'un morceau de code néglige d'acquérir le mutex, nous obtenons un comportement indéfini. En pratique, cela signifie des bogues extrêmement difficiles à reproduire et à corriger.

Même en Java, où il existe une association notionnelle entre les objets et les mutex, la relation n'est pas très profonde. Le compilateur ne tente pas de l'appliquer et, dans la pratique, les données protégées par un verrou sont rarement exactement les champs de l'objet associé. Il inclut souvent des données dans plusieurs objets. Les schémas de verrouillage sont encore délicats. Les commentaires restent le principal outil pour les appliquer.

## Mutex<T>

Nous allons maintenant montrer une implémentation de la liste d'attente dans Rust. Dans notre serveur de jeu Fern Empire, chaque joueur dispose d'un identifiant unique :

```
type PlayerId = u32;
```

La liste d'attente n'est qu'une collection de joueurs:

```
const GAME_SIZE: usize = 8;

/// A waiting list never grows to more than GAME_SIZE players.
type WaitingList = Vec<PlayerId>;
```

La liste d'attente est stockée sous la forme d'un champ de , un singleton qui est configuré dans un lors du démarrage du serveur. Chaque fil a un pointage vers lui. Il contient toute la configuration partagée et d'autres flotsam dont notre programme a besoin. La plupart de ces éléments sont en lecture seule. La liste d'attente étant à la fois partagée et modifiable, elle doit être protégée par un : FernEmpireApp Arc Arc Mutex

```

use std::sync::Mutex;

/// All threads have shared access to this big context struct.
struct FernEmpireApp {
 ...
 waiting_list: Mutex<WaitingList>,
 ...
}

```

Contrairement à C++, dans Rust, les données protégées sont stockées à l'intérieur du fichier . La configuration du ressemble à ceci: Mutex Mutex

```

use std::sync::Arc;

let app = Arc::new(FernEmpireApp {
 ...
 waiting_list: Mutex::new(vec![]),
 ...
});

```

La création d'un nouveau ressemble à la création d'un nouveau ou , mais tout en signifiant l'allocation de tas, il s'agit uniquement de verrouiller. Si vous voulez que votre allocation soit allouée dans le tas, vous devez le dire, comme nous l'avons fait ici en utilisant pour l'ensemble de l'application et juste pour les données protégées. Ces types sont couramment utilisés ensemble : ils sont pratiques pour partager des éléments entre les threads et sont pratiques pour les données modifiables partagées entre les

threads. Mutex Box Arc Box Arc Mutex Mutex Arc::new Mutex::new Arc Mutex

Maintenant, nous pouvons implémenter la méthode qui utilise le mutex: join\_waiting\_list

```

impl FernEmpireApp {
 /// Add a player to the waiting list for the next game.
 /// Start a new game immediately if enough players are waiting.
 fn join_waiting_list(&self, player: PlayerId) {
 // Lock the mutex and gain access to the data inside.
 // The scope of `guard` is a critical section.
 let mut guard = self.waiting_list.lock().unwrap();

 // Now do the game logic.
 guard.push(player);
 if guard.len() == GAME_SIZE {

```

```

 let players = guard.split_off(0);
 self.start_game(players);
 }
}

```

La seule façon d'accéder aux données est d'appeler la méthode : `.lock()`

```

let mut guard = self.waiting_list.lock().unwrap();

self.waiting_list.lock() bloque jusqu'à ce que le mutex puisse
être obtenu. La valeur renvoyée par cet appel de méthode est un wrapper
mince autour d'un . Grâce aux coercitions deref, discutées, nous pouvons
appeler des méthodes directement sur la
garde: MutexGuard<WaitingList> &mut WaitingList WaitingList

guard.push(player);

```

Le gardien nous permet même d'emprunter des références directes aux données sous-jacentes. Le système de durée de vie de Rust garantit que ces références ne peuvent pas survivre au gardien lui-même. Il n'y a aucun moyen d'accéder aux données dans un sans maintenir le verrou. `Mutex`

Lorsqu'il est déposé, le verrou est relâché. Normalement, cela se produit à la fin du bloc, mais vous pouvez également le déposer manuellement: `guard`

```

if guard.len() == GAME_SIZE {
 let players = guard.split_off(0);
 drop(guard); // don't keep the list locked while starting a game
 self.start_game(players);
}

```

## mut et Mutex

Il peut sembler étrange – certainement cela nous a semblé étrange au début – que notre méthode ne prenne pas par référence. Sa signature type est : `join_waiting_list self mut`

```
fn join_waiting_list(&self, player: PlayerId)
```

La collection sous-jacente, , nécessite une référence lorsque vous appelez sa méthode. Sa signature type est : `Vec<PlayerId> mut push`

```
pub fn push(&mut self, item: T)
```

Et pourtant, ce code se compile et s'exécute correctement. Qu'est-ce qui se passe?

Dans Rust, signifie *un accès exclusif*. Plain signifie *accès partagé*. `&mut` &

Nous avons l'habitude de transmettre l'accès du parent à l'enfant, du conteneur au contenu. Vous ne vous attendez à pouvoir faire appel à des méthodes que si vous avez une référence pour commencer (ou si vous possédez, auquel cas félicitations pour être Elon Musk). C'est la valeur par défaut, car si vous n'avez pas un accès exclusif au parent, Rust n'a généralement aucun moyen de s'assurer que vous avez un accès exclusif à l'enfant.

```
self.starships[id].engine &mut starships starships
```

Mais a un moyen: la serrure. En fait, un mutex n'est guère plus qu'un moyen de faire exactement cela, de fournir un accès *exclusif()* aux données à l'intérieur, même si de nombreux threads peuvent avoir *partagé* (non-) un accès à lui-même.

Le système de type Rust nous dit ce qui se passe. Il applique dynamiquement l'accès exclusif, ce qui est généralement fait statiquement, au moment de la compilation, par le compilateur Rust.

(Vous vous souviendrez peut-être que cela fait la même chose, sauf sans essayer de prendre en charge plusieurs threads. et sont les deux saveurs de mutabilité intérieure, que nous avons couvertes.)

## Pourquoi les mutex ne sont pas toujours une bonne idée

Avant de commencer sur les mutex, nous avons présenté quelques approches de la concurrence d'accès qui auraient pu sembler étrangement faciles à utiliser correctement si vous venez de C++. Ce n'est pas un hasard : ces approches sont conçues pour fournir des garanties solides contre les aspects les plus déroutants de la programmation simultanée. Les programmes qui utilisent exclusivement le parallélisme fork-join sont déterministes et ne peuvent pas bloquer. Les programmes qui utilisent des canaux se comportent presque aussi bien. Ceux qui utilisent des canaux exclusivement pour le pipelining, comme notre générateur d'index, sont déterministes : le moment de la remise des messages peut vari-

er, mais cela n'affectera pas la sortie. Et ainsi de suite. Les garanties sur les programmes multithread sont agréables!

La conception de Rust vous fera presque certainement utiliser des mutex plus systématiquement et plus judicieusement que jamais auparavant. Mais il vaut la peine de faire une pause et de réfléchir à ce que les garanties de sécurité de Rust peuvent et ne peuvent pas aider. Mutex

Le code Safe Rust ne peut pas déclencher une *course aux données*, un type spécifique de bogue où plusieurs threads lisent et écrivent la même mémoire simultanément, produisant des résultats dénués de sens. C'est génial : les courses de données sont toujours des bugs, et ils ne sont pas rares dans les vrais programmes multithread.

Cependant, les threads qui utilisent des mutex sont sujets à d'autres problèmes que Rust ne résout pas pour vous :

- Les programmes Rust valides ne peuvent pas avoir de courses de données, mais ils peuvent toujours avoir d'autres *conditions de course*, des situations où le comportement d'un programme dépend du timing entre les threads et peut donc varier d'une exécution à l'autre. Certaines conditions de course sont bénignes. Certains se manifestent par des flocons généraux et des bogues incroyablement difficiles à corriger. L'utilisation de mutex de manière non structurée invite à des conditions de course. C'est à vous de vous assurer qu'ils sont bénins.
- L'état mutable partagé affecte également la conception du programme. Là où les canaux servent de limite d'abstraction dans votre code, ce qui facilite la séparation des composants isolés à des fins de test, les mutex encouragent une méthode de travail « juste-ajouter-une-méthode » qui peut conduire à un blob monolithique de code interdépendant.
- Enfin, les mutex ne sont tout simplement pas aussi simples qu'ils le semblent au premier abord, comme le montreront les deux sections suivantes.

Tous ces problèmes sont inhérents aux outils. Utilisez une approche plus structurée lorsque vous le pouvez; utilisez un quand vous devez. Mutex

## Impasse

Un thread peut se bloquer en essayant d'acquérir un verrou qu'il détient déjà :

```
let mut guard1 = self.waiting_list.lock().unwrap();
let mut guard2 = self.waiting_list.lock().unwrap(); // deadlock
```

Supposons que le premier appel réussisse, en prenant le verrou. Le deuxième appel voit que le verrou est maintenu, il se bloque, attendant qu'il soit libéré. Il attendra pour toujours. Le fil d'attente est celui qui maintient la serrure. `self.waiting_list.lock()`

En d'autres termes, le verrou dans un n'est pas un verrou récursif. `Mutex`

Ici, le bug est évident. Dans un programme réel, les deux appels peuvent être dans deux méthodes différentes, dont l'une appelle l'autre. Le code de chaque méthode, pris séparément, aurait l'air bien. Il existe également d'autres moyens d'obtenir un blocage, impliquant plusieurs threads qui acquièrent chacun plusieurs mutex à la fois. Le système d'emprunt de Rust ne peut pas vous protéger contre les blocages. La meilleure protection est de garder les sections critiques petites: entrez, faites votre travail et sortez. `lock()`

Il est également possible d'obtenir une impasse avec les canaux. Par exemple, deux threads peuvent se bloquer, chacun attendant de recevoir un message de l'autre. Cependant, encore une fois, une bonne conception de programme peut vous donner une grande confiance que cela ne se produira pas dans la pratique. Dans un pipeline, comme notre générateur d'index inversé, le flux de données est acyclique. L'impasse est aussi improbable dans un tel programme que dans un pipeline shell Unix.

## Mutex empoisonnés

`Mutex::lock()` renvoie un pour la même raison que : échouer gracieusement si un autre thread a paniqué. Lorsque nous écrivons, nous disons à Rust de propager la panique d'un fil à l'autre. L'idiome est similaire. `Result JoinHandle::join() handle.join().unwrap() mutex.lock().unwrap()`

Si un fil panique en tenant un , Rust marque le comme *empoisonné*. Toute tentative ultérieure à l'empoisonné obtiendra un résultat d'erreur. Notre appel dit à Rust de paniquer si cela se produit, propageant la panique de l'autre fil à celui-ci. `Mutex Mutex lock Mutex .unwrap()`

À quel point est-ce mauvais d'avoir un mutex empoisonné? Le poison semble mortel, mais ce scénario n'est pas nécessairement fatal. Comme nous l'avons dit au [chapitre 7](#), la panique est sans danger. Un fil de panique laisse le reste du programme dans un état sûr.

La raison pour laquelle les mutex sont empoisonnés par la panique n'est donc pas par peur d'un comportement indéfini. Le problème est plutôt

que vous avez probablement programmé avec des invariants. Puisque votre programme a paniqué et renfloué une section critique sans avoir terminé ce qu'il faisait, peut-être après avoir mis à jour certains champs des données protégées mais pas d'autres, il est possible que les invariants soient maintenant cassés. La rouille empoisonne le mutex pour empêcher d'autres fils de gaffer involontairement dans cette situation brisée et de l'aggraver. Vous pouvez toujours verrouiller un mutex empoisonné et accéder aux données à l'intérieur, avec une exclusion mutuelle pleinement appliquée; reportez-vous à la documentation pour `.PoisonError::into_inner()`.

## Canaux multiconsommateurs utilisant Mutexes

Nous avons mentionné plus tôt que les chaînes de Rust sont plusieurs producteurs, un seul consommateur. Ou pour le dire plus concrètement, un canal n'en a qu'un. Nous ne pouvons pas avoir un pool de threads où de nombreux threads utilisent un seul canal comme liste de travail partagée. `Receiver mpsc`

Cependant, il s'avère qu'il existe une solution de contournement très simple, en utilisant uniquement des éléments de bibliothèque standard. Nous pouvons ajouter un围绕 de la et le partager de toute façon. Voici un module qui le fait : `Mutex Receiver`

```
pub mod shared_channel {
 use std::sync::{Arc, Mutex};
 use std::sync::mpsc::{channel, Sender, Receiver};

 /// A thread-safe wrapper around a `Receiver`.
 #[derive(Clone)]
 pub struct SharedReceiver<T>(Arc<Mutex<Receiver<T>>>);

 impl<T> Iterator for SharedReceiver<T> {
 type Item = T;

 /// Get the next item from the wrapped receiver.
 fn next(&mut self) -> Option<T> {
 let guard = self.0.lock().unwrap();
 guard.recv().ok()
 }
 }

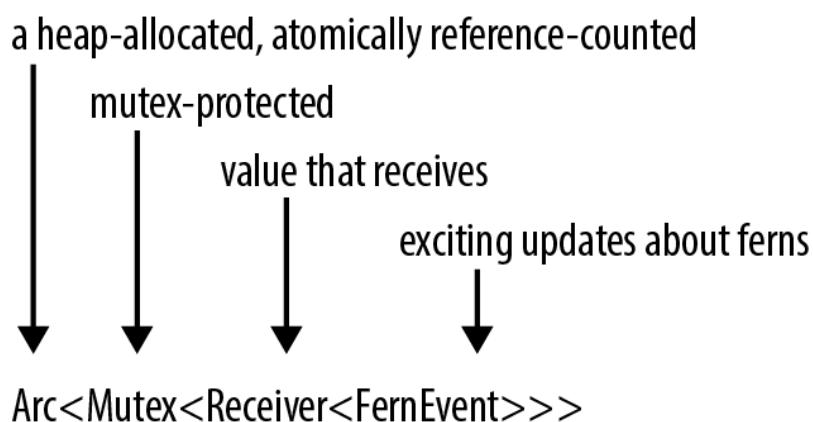
 /// Create a new channel whose receiver can be shared across threads
 /// This returns a sender and a receiver, just like the stdlib's
 /// `channel()`, and sometimes works as a drop-in replacement.
```

```

pub fn shared_channel<T>() -> (Sender<T>, SharedReceiver<T>) {
 let (sender, receiver) = channel();
 (sender, SharedReceiver(Arc::new(Mutex::new(receiver))))
}

```

Nous utilisons un fichier . Les génériques se sont vraiment empilés. Cela se produit plus souvent dans Rust qu'en C++. Il peut sembler que cela deviendrait déroutant, mais souvent, comme dans ce cas, la simple lecture des noms peut aider à expliquer ce qui se passe, comme le montre [la figure 19-11](#). Arc<Mutex<Receiver<T>>>



Graphique 19-11. Comment lire un type complexe

## Verrous en lecture/écriture (RwLock<T>)

Passons maintenant des mutex aux autres outils fournis dans , la boîte à outils de synchronisation de thread de bibliothèque standard de Rust.

Nous allons avancer rapidement, car une discussion complète de ces outils dépasse le cadre de ce livre. std::sync

Les programmes serveur ont souvent des informations de configuration qui sont chargées une fois et qui changent rarement. La plupart des threads interrogent uniquement la configuration, mais comme la configuration *peut* changer (il peut être possible de demander au serveur de recharger sa configuration à partir du disque, par exemple), elle doit de toute façon être protégée par un verrou. Dans des cas comme celui-ci, un mutex peut fonctionner, mais c'est un goulot d'étranglement inutile. Les threads ne devraient pas avoir à interroger la configuration à tour de rôle si elle ne change pas. C'est le cas d'un *verrou en lecture/écriture*, ou . RwLock

Alors qu'un mutex a une seule méthode, un verrou en lecture/écriture a deux méthodes de verrouillage et . La méthode est comme . Il attend un accès exclusif aux données protégées. La méthode fournit un non-accès, avec l'avantage qu'il est moins susceptible d'avoir à attendre, car de nom-

breux threads peuvent lire en toute sécurité à la fois. Avec un mutex, à tout moment, les données protégées n'ont qu'un seul lecteur ou écrivain (ou aucun). Avec un verrou de lecture / écriture, il peut avoir un écrivain ou plusieurs lecteurs, un peu comme les références Rust en général.

```
lock read write
```

```
RwLock::write Mutex::lock mut RwLock
::read mut
```

FernEmpireApp peut avoir une structure pour la configuration, protégée par un : RwLock

```
use std::sync::RwLock;

struct FernEmpireApp {
 ...
 config: RwLock<AppConfig>,
 ...
}
```

Les méthodes qui lisent la configuration utiliseraient : RwLock::read()

```
/// True if experimental fungus code should be used.
fn mushrooms_enabled(&self) -> bool {
 let config_guard = self.config.read().unwrap();
 config_guard.mushrooms_enabled
}
```

La méthode de recharge de la configuration utiliserait : RwLock::write()

```
fn reload_config(&self) -> io::Result<()> {
 let new_config = AppConfig::load()?;
 let mut config_guard = self.config.write().unwrap();
 *config_guard = new_config;
 Ok(())
}
```

Rust, bien sûr, est particulièrement bien adapté pour appliquer les règles de sécurité sur les données. Le concept d'écrivain unique ou de lecteur multiple est au cœur du système d'emprunt de Rust. renvoie un garde qui fournit un accès non (partagé) au ; renvoie un autre type de garde qui fournit un accès exclusif).

```
RwLock self.config.read() mut AppConfig self.config
.write() mut
```

# Variables de condition (Condvar)

Souvent, un thread doit attendre qu'une certaine condition devienne vraie:

- Pendant l'arrêt du serveur, le thread principal peut avoir besoin d'attendre que tous les autres threads aient fini de se séparer.
- Lorsqu'un thread de travail n'a rien à faire, il doit attendre qu'il y ait des données à traiter.
- Un thread implémentant un protocole de consensus distribué peut devoir attendre qu'un quorum de pairs ait répondu.

Parfois, il existe une API de blocage pratique pour la condition exacte que nous voulons attendre, comme pour l'exemple d'arrêt du serveur. Dans d'autres cas, il n'y a pas d'API de blocage intégrée. Les programmes peuvent utiliser des *variables de condition* pour créer les leurs. Dans Rust, le type implémente des variables de condition. A a des méthodes et ; bloque jusqu'à ce qu'un autre thread appelle

```
.JoinHandle::join std::sync::Condvar Condvar .wait() .notify_all() .wait() .notify_all()
```

Il y a un peu plus que cela, car une variable de condition concerne toujours une condition vraie ou fausse particulière à propos de certaines données protégées par un . Ceci et le sont donc liés. Une explication complète est plus que ce que nous avons de la place ici, mais pour le bénéfice des programmeurs qui ont déjà utilisé des variables de condition, nous allons montrer les deux bits clés du code. Mutex Mutex Condvar

Lorsque la condition souhaitée devient vraie, nous appelons (ou ) pour réveiller les fils d'attente : Condvar::notify\_all notify\_one

```
self.has_data_condvar.notify_all();
```

Pour s'endormir et attendre qu'une condition devienne vraie, nous utilisons : Condvar::wait()

```
while !guard.has_data() {
 guard = self.has_data_condvar.wait(guard).unwrap();
}
```

Cette boucle est un idiome standard pour les variables de condition. Cependant, la signature de est inhabituelle. Il prend un objet par valeur, le consomme et renvoie un nouveau sur le succès. Cela capture l'intuition que la méthode libère le mutex, puis le réacquiert avant de revenir. Pass-

er la valeur par est une façon de dire : « Je vous accorde, méthode, mon autorité exclusive pour libérer le mutex.

```
» while Condvar::wait MutexGuard MutexGuard wait MutexGuard .
wait()
```

## Atomes

Le module contient des types atomiques pour une programmation simultanée sans verrouillage. Ces types sont fondamentalement les mêmes que les atomes C ++ standard, avec quelques extras: `std::sync::atomic`

- `AtomicIsize` et sont des types entiers partagés correspondant au thread unique et aux types. `AtomicUsize` `usize` `usize`
- `AtomicI8` , , , et leurs variantes non signées comme sont des types entiers partagés qui correspondent aux types monothread , , etc. `AtomicI16` `AtomicI32` `AtomicI64` `AtomicU8` `i8` `i16`
- An est une valeur partagée. `AtomicBool` `bool`
- An est une valeur partagée du type de pointeur non sécurisé  
. `AtomicPtr<T>` `*mut T`

L'utilisation correcte des données atomiques dépasse le cadre de ce livre. Il suffit de dire que plusieurs threads peuvent lire et écrire une valeur atomique à la fois sans provoquer de courses de données.

Au lieu des opérateurs arithmétiques et logiques habituels, les types atomiques exposent des méthodes qui effectuent des *opérations atomiques*, des charges individuelles, des magasins, des échanges et des opérations arithmétiques qui se produisent en toute sécurité, en tant qu'unité, même si d'autres threads effectuent également des opérations atomiques qui touchent le même emplacement de mémoire. L'incrémentation d'un nom ressemble à ceci : `AtomicIsize atom`

```
use std::sync::atomic::{AtomicIsize, Ordering};

let atom = AtomicIsize::new(0);
atom.fetch_add(1, Ordering::SeqCst);
```

Ces méthodes peuvent être compilées en instructions spécialisées en langage machine. Sur l'architecture x86-64, cet appel se compile en une instruction, où un ordinaire peut compiler en une instruction simple ou un certain nombre de variations sur ce thème. Le compilateur Rust doit également renoncer à certaines optimisations autour de l'opération atomique, car, contrairement à une charge ou un stockage normal, il peut

```
légitimement affecter ou être affecté par d'autres threads immédiatement. .fetch_add() lock incq n += 1 incq
```

L'argument est un *ordre de mémoire*. Les ordres de mémoire sont quelque chose comme les niveaux d'isolement des transactions dans une base de données. Ils disent au système à quel point vous vous souciez de notions philosophiques telles que les causes des effets précédents et le temps n'ayant pas de boucles, par opposition à la performance. Les ordres de mémoire sont cruciaux pour l'exactitude du programme, et ils sont difficiles à comprendre et à raisonner. Heureusement, la pénalité de performance pour le choix de la cohérence séquentielle, l'ordre de mémoire le plus strict, est souvent assez faible, contrairement à la pénalité de performance pour la mise en mode d'une base de données SQL. Donc, en cas de doute, utilisez . Rust hérite de plusieurs autres ordres de mémoire des atomes C++ standard, avec diverses garanties plus faibles sur la nature de l'existence et la causalité. Nous n'en discuterons pas ici.

```
Ordering::SeqCst SERIALIZABLE Ordering::SeqCst
```

Une utilisation simple des atomes est pour l'annulation. Supposons que nous ayons un thread qui effectue des calculs de longue durée, tels que le rendu d'une vidéo, et que nous aimerais pouvoir l'annuler de manière asynchrone. Le problème est de communiquer au thread que nous voulons qu'il s'arrête. Nous pouvons le faire via un partage : `AtomicBool`

```
use std::sync::Arc;
use std::sync::atomic::AtomicBool;

let cancel_flag = Arc::new(AtomicBool::new(false));
let worker_cancel_flag = cancel_flag.clone();
```

Ce code crée deux pointeurs intelligents qui pointent vers le même tas alloué , dont la valeur initiale est . Le premier, nommé , restera dans le fil principal. Le second, , sera déplacé vers le thread de travail.

```
Arc<AtomicBool> AtomicBool false cancel_flag worker_cancel_flag
```

Voici le code du travailleur :

```
use std::thread;
use std::sync::atomic::Ordering;

let worker_handle = thread::spawn(move || {
 for pixel in animation.pixels_mut() {
 render(pixel); // ray-tracing - this takes a few microseconds
```

```

 if worker_cancel_flag.load(Ordering::SeqCst) {
 return None;
 }
 }
 Some(animation)
);

```

Après avoir rendu chaque pixel, le thread vérifie la valeur de l'indicateur en appelant sa méthode `: .load()`

```
worker_cancel_flag.load(Ordering::SeqCst)
```

Si, dans le thread principal, nous décidons d'annuler le thread de travail, nous stockons dans le, puis attendons que le thread se ferme

```
: true AtomicBool
```

```

// Cancel rendering.
cancel_flag.store(true, Ordering::SeqCst);

// Discard the result, which is probably `None`.
worker_handle.join().unwrap();

```

Bien sûr, il existe d'autres moyens de mettre cela en œuvre. L'ici pourrait être remplacé par un ou un canal. La principale différence est que les atomes ont une surcharge minimale. Les opérations atomiques n'utilisent jamais d'appels système. Une charge ou un stockage se compile souvent en une seule instruction CPU. `AtomicBool Mutex<bool>`

Les atomes sont une forme de mutabilité intérieure, comme ou , de sorte que leurs méthodes prennent par référence partagée (non-). Cela les rend utiles en tant que variables globales simples. `Mutex RwLock self mut`

## Variables globales

Supposons que nous écrivions du code réseau. Nous aimerais avoir une variable globale, un compteur que nous incrémentons chaque fois que nous servons un paquet :

```

/// Number of packets the server has successfully handled.
static PACKETS_SERVED: usize = 0;

```

Cela compile bien. Il n'y a qu'un seul problème. n'est pas mutable, donc nous ne pouvons jamais le changer. `PACKETS_SERVED`

Rust fait tout ce qu'elle peut raisonnablement pour décourager l'état mutable mondial. Les constantes déclarées avec sont, bien sûr, immuables. Les variables statiques sont également immuables par défaut, il n'y a donc aucun moyen d'obtenir une référence à une. A peut être déclaré , mais alors y accéder n'est pas sûr. L'insistance de Rust sur la sécurité des fils est une raison majeure de toutes ces règles. `const mut static mut`

L'état global mutable a également des conséquences malheureuses en génie logiciel: il a tendance à rendre les différentes parties d'un programme plus étroitement couplées, plus difficiles à tester et plus difficiles à modifier plus tard. Pourtant, dans certains cas, il n'y a tout simplement pas d'alternative raisonnable, nous ferions donc mieux de trouver un moyen sûr de déclarer des variables statiques mutables.

Le moyen le plus simple de prendre en charge l'incrémentation, tout en le gardant thread-safe, est d'en faire un entier atomique : `PACKETS_SERVED`

```
use std::sync::atomic::AtomicUsize;

static PACKETS_SERVED: AtomicUsize = AtomicUsize::new(0);
```

Une fois cette statique déclarée, l'incrémentation du nombre de paquets est simple :

```
use std::sync::atomic::Ordering;

PACKETS_SERVED.fetch_add(1, Ordering::SeqCst);
```

Les globaux atomiques sont limités aux entiers simples et aux booléens. Néanmoins, créer une variable globale de tout autre type revient à résoudre deux problèmes.

Tout d'abord, la variable doit être rendue thread-safe d'une manière ou d'une autre, car sinon elle ne peut pas être globale : pour la sécurité, les variables statiques doivent être à la fois et non-. Heureusement, nous avons déjà vu la solution à ce problème. Rust a des types pour partager en toute sécurité des valeurs qui changent: , et les types atomiques. Ces types peuvent être modifiés même lorsqu'ils sont déclarés comme non-. C'est ce qu'ils font. (Voir [« mut et Mutex »](#).) `Sync mut Mutex RwLock mut`

Deuxièmement, les initialiseurs statiques ne peuvent appeler que des fonctions spécifiquement marquées comme , que le compilateur peut évaluer pendant la compilation. En d'autres termes, leur production est déterministe; cela ne dépend que de leurs arguments, pas d'un autre état

ou d'E/S. De cette façon, le compilateur peut intégrer les résultats de ce calcul en tant que constante de temps de compilation. Ceci est similaire à C++ .const constexpr

Les constructeurs pour les types (, , et ainsi de suite) sont tous des fonctions, ce qui nous a permis de créer un précédent. Quelques autres types, comme , et , ont des constructeurs simples qui le sont

```
atomic AtomicUsize atomicBool const static AtomicUsize S
string Ipv4Addr Ipv6Addr const
```

Vous pouvez également définir vos propres fonctions en préfixant simplement la signature de la fonction par . Rust limite ce que les fonctions peuvent faire à un petit ensemble d'opérations, qui sont suffisantes pour être utiles tout en ne permettant aucun résultat non déterministe. les fonctions ne peuvent pas prendre les types comme arguments génériques, seulement les durées de vie, et il n'est pas possible d'allouer de la mémoire ou de fonctionner sur des pointeurs bruts, même en blocs. Nous pouvons cependant utiliser des opérations arithmétiques (y compris l'encapsulation et l'arithmétique saturée), des opérations logiques qui ne court-circuitent pas et d'autres fonctions. Par exemple, nous pouvons créer des fonctions pratiques pour faciliter la définition des s et s et réduire la duplication du code

```
:const const const unsafe const static const
```

```
const fn mono_to_rgba(level: u8) -> Color {
 Color {
 red: level,
 green: level,
 blue: level,
 alpha: 0xFF
 }
}

const WHITE: Color = mono_to_rgba(255);
const BLACK: Color = mono_to_rgba(000);
```

En combinant ces techniques, nous pourrions être tentés d'écrire:

```
static HOSTNAME: Mutex<String> =
 Mutex::new(String::new()); // error: calls in statics are limited to
 // constant functions, tuple structs, and
 // tuple variants
```

Malheureusement, alors que et sont , n'est pas. Afin de contourner ces limitations, nous devons utiliser la caisse.

```
AtomicUsize::new() String::new() const
fn Mutex::new() lazy_static
```

Nous avons introduit la caisse dans [« Building Regex Values Lazily »](#). La définition d'une variable avec la macro vous permet d'utiliser n'importe quelle expression de votre choix pour l'initialiser ; il s'exécute la première fois que la variable est déréférencée et la valeur est enregistrée pour toutes les utilisations ultérieures. `lazy_static! lazy_static!`

Nous pouvons déclarer un global -contrôlé avec comme ceci:

```
Mutex HashMap lazy_static
```

```
use lazy_static::lazy_static;

use std::sync::Mutex;

lazy_static! {
 static ref HOSTNAME: Mutex<String> = Mutex::new(String::new());
}
```

La même technique fonctionne pour d'autres structures de données complexes comme `s` et `s`. C'est également très pratique pour les statiques qui ne sont pas du tout mutables, mais qui nécessitent simplement une initialisation non triviale. `HashMap Deque`

L'utilisation impose un coût de performance minime à chaque accès aux données statiques. L'implémentation utilise , une primitive de synchronisation de bas niveau conçue pour une initialisation unique. Dans les coulisses, chaque fois qu'une statique paresseuse est consultée, le programme exécute une instruction de charge atomique pour vérifier que l'initialisation a déjà eu lieu. ( est un but assez spécial, nous ne le couvrirons donc pas en détail ici. Il est généralement plus pratique à utiliser à la place. Cependant, il est pratique pour initialiser des bibliothèques non-Rust; pour obtenir un exemple, voir [« Une interface sécurisée vers libgit2 »](#).) `lazy_static! std::sync::Once Once lazy_static!`

## À quoi ressemble le piratage de code simultané dans Rust

Nous avons montré trois techniques d'utilisation des threads dans Rust : le parallélisme fourche-jointure, les canaux et l'état mutable partagé avec

des verrous. Notre objectif a été de fournir une bonne introduction aux pièces fournies par Rust, en mettant l'accent sur la façon dont elles peuvent s'intégrer dans de vrais programmes.

Rust insiste sur la sécurité, donc à partir du moment où vous décidez d'écrire un programme multithread, l'accent est mis sur la construction d'une communication sûre et structurée. Garder les fils principalement isolés est un bon moyen de convaincre Rust que ce que vous faites est sûr. Il arrive que l'isolement soit également un bon moyen de s'assurer que ce que vous faites est correct et maintenable. Encore une fois, Rust vous guide vers de bons programmes.

Plus important encore, Rust vous permet de combiner des techniques et des expériences. Vous pouvez itérer rapidement : discuter avec le compilateur vous permet d'être opérationnel correctement beaucoup plus rapidement que de déboguer des courses de données.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

# Chapitre 20. Programmation asynchrone

Supposons que vous écrivez un serveur de chat. Pour chaque connexion réseau, il y a des paquets entrants à analyser, des paquets sortants à assembler, des paramètres de sécurité à gérer, des abonnements de groupe de discussion à suivre, etc. Gérer tout cela pour de nombreuses connexions simultanément va prendre une certaine organisation.

Idéalement, vous pouvez simplement démarrer un thread distinct pour chaque connexion entrante :

```
use std::net, thread;

let listener = net::TcpListener::bind(address)?;

for socket_result in listener.incoming() {
 let socket = socket_result?;
 let groups = chat_group_table.clone();
 thread::spawn(|| {
 log_error(serve(socket, groups));
 });
}
```

Pour chaque nouvelle connexion, cela génère un nouveau thread exécutant la fonction, qui est capable de se concentrer sur la gestion des besoins d'une seule connexion. `serve`

Cela fonctionne bien, jusqu'à ce que tout se passe beaucoup mieux que prévu et que vous ayez soudainement des dizaines de milliers d'utilisateurs. Il n'est pas rare que la pile d'un thread atteigne 100 Kio ou plus, et ce n'est probablement pas ainsi que vous souhaitez dépenser des gigaoctets de mémoire serveur. Les threads sont bons et nécessaires pour répartir le travail sur plusieurs processeurs, mais leurs demandes de mémoire sont telles que nous avons souvent besoin de moyens complémentaires, utilisés avec des threads, pour décomposer le travail.

Vous pouvez utiliser des *tâches asynchrones* Rust pour intercaler de nombreuses activités indépendantes sur un seul thread ou un pool de threads de travail. Les tâches asynchrones sont similaires aux threads, mais sont beaucoup plus rapides à créer, passent le contrôle entre elles plus efficacement et ont une surcharge de mémoire inférieure à celle d'un thread.

Il est parfaitement possible d'avoir des centaines de milliers de tâches asynchrones exécutées simultanément dans un seul programme. Bien sûr, votre application peut toujours être limitée par d'autres facteurs tels que la bande passante réseau, la vitesse de la base de données, le calcul ou les besoins en mémoire inhérents au travail, mais la surcharge de mémoire inhérente à l'utilisation des tâches est beaucoup moins importante que celle des threads.

Généralement, le code Rust asynchrone ressemble beaucoup au code multithread ordinaire, sauf que les opérations qui pourraient bloquer, comme les E/S ou l'acquisition de mutex, doivent être gérées un peu différemment. Le traitement de ceux-ci donne à Rust plus d'informations sur la façon dont votre code se comportera, ce qui rend possible l'amélioration des performances. La version asynchrone du code précédent ressemble à ceci :

```
use async_std::{net, task};

let listener = net::TcpListener::bind(address).await?;

let mut new_connections = listener.incoming();
while let Some(socket_result) = new_connections.next().await {
 let socket = socket_result?;
 let groups = chat_group_table.clone();
 task::spawn(async {
 log_error(serve(socket, groups).await);
 });
}
```

Cela utilise les modules de mise en réseau et de tâche de la caisse et ajoute après les appels qui peuvent être bloqués. Mais la structure globale est la même que la version basée sur les threads. `async_std .await`

L'objectif de ce chapitre n'est pas seulement de vous aider à écrire du code asynchrone, mais aussi de montrer comment il fonctionne suffisamment en détail pour que vous puissiez anticiper ses performances dans vos applications et voir où il peut être le plus précieux.

- Pour montrer la mécanique de la programmation asynchrone, nous présentons un ensemble minimal de fonctionnalités de langage qui couvre tous les concepts de base: futurs, fonctions asynchrones, expressions, tâches et exécuteurs. `await block_on spawn_local`
- Ensuite, nous présentons les blocs asynchrones et l'exécuteur. Ceux-ci sont essentiels pour faire un travail réel, mais conceptuellement, ce ne

sont que des variantes des fonctionnalités que nous venons de mentionner. Dans le processus, nous soulignons quelques problèmes que vous êtes susceptible de rencontrer qui sont uniques à la programmation asynchrone et expliquons comment les gérer. `spawn`

- Pour montrer toutes ces pièces travaillant ensemble, nous parcourons le code complet d'un serveur et d'un client de chat, dont le fragment de code précédent fait partie.
- Pour illustrer le fonctionnement des futurs primitifs et des exécuteurs, nous présentons des implémentations simples mais fonctionnelles de `spawn_blocking block_on`
- Enfin, nous expliquons le type, qui apparaît de temps en temps dans les interfaces asynchrones pour s'assurer que la fonction asynchrone et les futurs de bloc sont utilisés en toute sécurité. `Pin`

## Du synchrone à l'asynchrone

Considérez ce qui se passe lorsque vous appelez la fonction suivante (non asynchrone, complètement traditionnelle) :

```
use std::io::prelude::*;
use std::net;

fn cheapo_request(host: &str, port: u16, path: &str)
 -> std::io::Result<String>
{
 let mut socket = net::TcpStream::connect((host, port))?;

 let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
 socket.write_all(request.as_bytes())?;
 socket.shutdown(net::Shutdown::Write)?;
}

let mut response = String::new();
socket.read_to_string(&mut response)?;

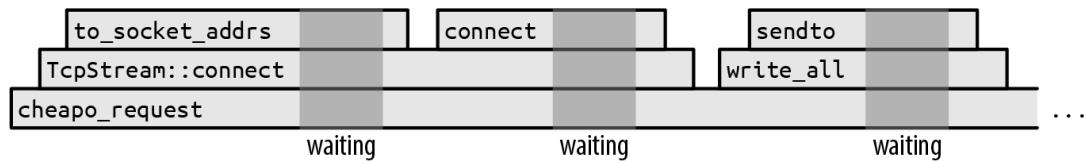
Ok(response)
}
```

Cela ouvre une connexion TCP à un serveur Web, lui envoie une requête HTTP nue dans un protocole obsolète,<sup>1</sup> puis lit la réponse. [La figure 20-1](#) montre l'exécution de cette fonction au fil du temps.

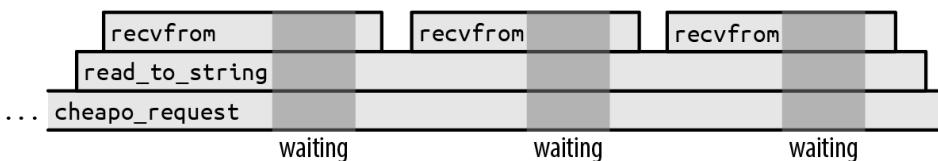
Ce diagramme montre comment la pile d'appels de fonction se comporte lorsque le temps s'écoule de gauche à droite. Chaque appel de fonction est une boîte, placée au-dessus de son appelant. Évidemment, la fonction

s'exécute tout au long de l'exécution. Il appelle des fonctions de la bibliothèque standard Rust comme et les implémentations de et . Ceux-ci appellent d'autres fonctions à leur tour, mais finalement le programme fait *des appels système*, des demandes au système d'exploitation pour faire quelque chose, comme ouvrir une connexion TCP, ou lire ou écrire des données.

```
cheapo_request TcpStream::connect TcpStream write_all
 read_to_string
```



(continued from above)



Graphique 20-1. Progression d'une requête HTTP synchrone (des zones grises plus sombres attendent le système d'exploitation)

Les arrière-plans gris plus foncé marquent les moments où le programme attend que le système d'exploitation termine l'appel système. Nous n'avons pas dessiné ces temps à l'échelle. Si nous l'avions fait, l'ensemble du diagramme serait gris plus foncé : en pratique, cette fonction passe presque tout son temps à attendre le système d'exploitation. L'exécution du code précédent serait des éclats étroits entre les appels système.

Pendant que cette fonction attend le retour des appels système, son thread unique est bloqué : il ne peut rien faire d'autre tant que l'appel système n'est pas terminé. Il n'est pas rare que la pile d'un thread ait une taille de dizaines ou de centaines de kilo-octets, donc s'il s'agissait d'un fragment d'un système plus grand, avec de nombreux threads travaillant à des tâches similaires, verrouiller les ressources de ces threads pour ne rien faire d'autre que d'attendre pourrait devenir assez coûteux.

Pour contourner ce problème, un thread doit être en mesure d'effectuer d'autres tâches en attendant la fin des appels système. Mais il n'est pas évident de savoir comment y parvenir. Par exemple, la signature de la fonction que nous utilisons pour lire la réponse du socket est la suivante :

```
fn read_to_string(&mut self, buf: &mut String) -> std::io::Result<usize>
```

C'est écrit directement dans le type: cette fonction ne revient pas tant que le travail n'est pas terminé ou que quelque chose ne va pas. Cette fonction

est *synchrone* : l'appelant reprend lorsque l'opération est terminée. Si nous voulons utiliser notre thread pour d'autres choses pendant que le système d'exploitation fait son travail, nous aurons besoin d'une nouvelle bibliothèque d'E/S qui fournit *une version asynchrone* de cette fonction.

## Contrats à terme

L'approche de Rust pour soutenir les opérations asynchrones consiste à introduire un trait , : std::future::Future

```
trait Future {
 type Output;
 // For now, read `Pin<&mut Self>` as `&mut Self`.
 fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<Self::Output>;
}

enum Poll<T> {
 Ready(T),
 Pending,
}
```

A représente une opération que vous pouvez tester pour l'achèvement. La méthode d'un futur n'attend jamais la fin de l'opération : elle revient toujours immédiatement. Si l'opération est terminée, renvoie , où est son résultat final. Sinon, il renvoie . Si et quand l'avenir vaut la peine d'être interrogé à nouveau, il promet de nous le faire savoir en appelant un *waker*, une fonction de rappel fournie dans le . Nous appelons cela le « modèle piñata » de la programmation asynchrone : la seule chose que vous pouvez faire avec un futur est de le frapper avec un jusqu'à ce qu'une valeur

tombe. Future poll poll Poll::Ready(output) output Pending Context poll

Tous les systèmes d'exploitation modernes incluent des variantes de leurs appels système que nous pouvons utiliser pour implémenter ce type d'interface d'interrogation. Sous Unix et Windows, par exemple, si vous mettez un socket réseau en mode non bloquant, les lectures et écritures renvoient une erreur si elles se bloquent ; vous devez réessayer plus tard.

Ainsi, une version asynchrone de aurait une signature à peu près comme ceci: read\_to\_string

```
fn read_to_string(&mut self, buf: &mut String)
-> impl Future<Output = Result<usize>>;
```

C'est la même chose que la signature que nous avons montrée précédemment, à l'exception du type de retour : la version asynchrone renvoie *un futur d'un*. Vous devrez sonder cet avenir jusqu'à ce que vous en obteniez un. Chaque fois qu'il est interrogé, la lecture se poursuit aussi loin que possible. La finale vous donne la valeur de réussite ou une valeur d'erreur, tout comme une opération d'E/S ordinaire. C'est le modèle général : la version asynchrone de n'importe quelle fonction prend les mêmes arguments que la version synchrone, mais le type de retour a un wrapped autour d'elle.

```
Result<usize> Ready(result) result Future
```

Appeler cette version de ne lit rien en fait; sa seule responsabilité est de construire et de rendre un avenir qui fera le vrai travail lorsqu'il sera sondé. Cet avenir doit contenir toutes les informations nécessaires à l'exécution de la demande faite par l'appel. Par exemple, l'avenir renvoyé par celui-ci doit se souvenir du flux d'entrée sur lequel il a été appelé et duquel il doit ajouter les données entrantes. En fait, puisque l'avenir contient les références et , la signature appropriée pour doit être:

```
read_to_string read_to_string String self buf read_to_string
```

```
fn read_to_string<'a>(&'a mut self, buf: &'a mut String)
-> impl Future<Output = Result<usize>> + 'a;
```

Cela ajoute des durées de vie pour indiquer que le rendement futur ne peut vivre que tant que les valeurs qui empruntent et empruntent. *self buf*

La caisse fournit des versions asynchrones de toutes les installations d'E/S de , y compris un trait asynchrone avec une méthode. suit de près la conception de , réutilisant les types de , dans ses propres interfaces chaque fois que possible, de sorte que les erreurs, les résultats, les adresses réseau et la plupart des autres données associées sont compatibles entre les deux mondes. La familiarité avec vous aide à utiliser , et vice versa.

```
async_std std Read read_to_string async-
std std std std async_std
```

L'une des règles du trait est que, une fois qu'un avenir est revenu, il peut supposer qu'il ne sera plus jamais interrogé. Certains contrats à terme reviennent pour toujours s'ils sont surpolés; d'autres peuvent paniquer ou pendre. (Ils ne doivent toutefois pas violer la sécurité de la mémoire ou du thread, ni provoquer un comportement non défini.) La méthode de l'adaptateur sur le trait transforme n'importe quel avenir en un avenir qui revient simplement pour toujours. Mais toutes les façons habituelles

de consommer des futurs respectent cette règle, donc n'est généralement pas nécessaire. Future Poll::Ready Poll::Pending fuse Future Poll::Pending fuse

Si les sondages semblent inefficaces, ne vous inquiétez pas. L'architecture asynchrone de Rust est soigneusement conçue pour que, tant que vos fonctions d'E/S de base sont correctement implémentées, vous n'interrogerez un avenir que lorsque cela en vaudra la peine. Chaque fois qu'on l'appelle, quelque chose quelque part devrait revenir, ou au moins faire des progrès vers cet objectif. Nous expliquerons comment cela fonctionne dans [« Primitive Futures and Executors: When Is a Future Worth Polling Again? »](#).

read\_to\_string poll Ready

Mais l'utilisation des contrats à terme semble être un défi : lorsque vous sondez, que devez-vous faire lorsque vous obtenez ? Vous devrez vous promener pour un autre travail que ce fil peut faire pour le moment, sans oublier de revenir sur ce futur plus tard et de le sonder à nouveau. L'ensemble de votre programme sera envahi par la plomberie en gardant une trace de qui est en attente et de ce qui devrait être fait une fois qu'ils sont prêts. La simplicité de notre fonction est ruinée.

Poll::Pending cheapo\_request

Bonne nouvelle! Ce n'est pas le cas.

## Fonctions asynchrones et expressions Await

Voici une version de écrit en tant que *fonction asynchrone* :

```
fn cheapo_request(&host: &str, port: u16, path: &str) -> std::io::Result<String>
```

```
use async_std::io::prelude::*;
use async_std::net;

async fn cheapo_request(host: &str, port: u16, path: &str)
 -> std::io::Result<String>
{
 let mut socket = net::TcpStream::connect((host, port)).await?;

 let request = format!("GET {} HTTP/1.1\r\nHost: {}\r\n\r\n", path, host);
 socket.write_all(request.as_bytes()).await?;
 socket.shutdown(net::Shutdown::Write)?;

 let mut response = String::new();
 socket.read_to_string(&mut response).await?;
}
```

```
Ok(response)
}
```

C'est token pour token le même que notre version originale, sauf:

- La fonction commence par au lieu de .`async fn fn`
- Il utilise les versions asynchrones de la caisse de , et . Ceux-ci renvoient tous des futurs de leurs résultats. (Les exemples de cette section utilisent la version de
  - .`) async_std TcpStream::connect write_all read_to_string 1`
  - .`7 async_std`
- Après chaque appel qui renvoie un futur, le code indique . Bien que cela ressemble à une référence à un champ struct nommé , il s'agit en fait d'une syntaxe spéciale intégrée dans le langage pour attendre qu'un avenir soit prêt. Une expression évalue jusqu'à la valeur finale de l'avenir. C'est ainsi que la fonction obtient les résultats de , et
  - .`..await await await connect write_all read_to_string`

Contrairement à une fonction ordinaire, lorsque vous appelez une fonction asynchrone, elle revient immédiatement, avant que le corps ne commence à s'exécuter. De toute évidence, la valeur de retour finale de l'appel n'a pas encore été calculée ; ce que vous obtenez est un *avenir de sa valeur finale*. Donc, si vous exécutez ce code :

```
let response = cheapo_request(host, port, path);
```

alors sera un avenir d'un , et le corps de n'a pas encore commencé l'exécution. Vous n'avez pas besoin d'ajuster le type de retour d'une fonction asynchrone ; Rust traite automatiquement comme une fonction qui renvoie un futur d'un , pas un

```
directement. response std::io::Result<String> cheapo_request a
sync fn f(...) -> T T T
```

Le futur renvoyé par une fonction asynchrone enveloppe toutes les informations dont le corps de la fonction aura besoin pour s'exécuter : les arguments de la fonction, l'espace pour ses variables locales, etc. (C'est comme si vous aviez capturé le cadre de pile de l'appel comme une valeur Rust ordinaire.) Il faut donc conserver les valeurs passées pour , , et , puisque le corps de ' va en avoir besoin pour fonctionner. `response host port path cheapo_request`

Le type spécifique du futur est généré automatiquement par le compilateur, en fonction du corps et des arguments de la fonction. Ce type n'a pas

de nom ; tout ce que vous savez à ce sujet, c'est qu'il implémente , où est le type de retour de la fonction asynchrone. En ce sens, les futurs des fonctions asynchrones sont comme des fermetures : les fermetures ont également des types anonymes, générés par le compilateur, qui implémentent le , et des traits. Future<Output=R> R FnOnce Fn FnMut

Lorsque vous interrogez pour la première fois le futur renvoyé par , l'exécution commence en haut du corps de la fonction et s'exécute jusqu'au premier du futur renvoyé par . L'expression sonde l'avenir, et si elle n'est pas prête, alors elle retourne à son propre appelant: l'avenir du sondage ne peut pas aller au-delà de cela d'abord jusqu'à ce qu'un sondage du futur revienne. Donc, un équivalent approximatif de l'expression pourrait être: cheapo\_request await TcpStream::connect await connect Poll::Pending cheapo\_request await TcpStream::connect Poll::Ready TcpStream::connect(...).await

```
{
 // Note: this is pseudocode, not valid Rust
 let connect_future = TcpStream::connect(...);
 'retry_point:
 match connect_future.poll(cx) {
 Poll::Ready(value) => value,
 Poll::Pending => {
 // Arrange for the next `poll` of `cheapo_request`'s
 // future to resume execution at 'retry_point'.
 ...
 return Poll::Pending;
 }
 }
}
```

Une expression s'approprie l'avenir et l'interroge ensuite. S'il est prêt, la valeur finale du futur est la valeur de l'expression, et l'exécution se poursuit. Sinon, il renvoie le à son propre appelant. await await Poll::Pending

Mais surtout, le prochain sondage de l'avenir de 'ne recommence pas en haut de la fonction: au lieu de cela, il *reprend* l'exécution à mi-fonction au point où il est sur le point de sonder . Nous ne progressons pas vers le reste de la fonction asynchrone tant que cet avenir n'est pas prêt. cheapo\_request connect\_future

Au fur et à mesure que l'avenir de ' continue d'être sondé, il se frayera un chemin à travers le corps de la fonction de l'un à l'autre, ne se déplaçant

que lorsque le sous-futur qu'il attend sera prêt. Ainsi, le nombre de fois que l'avenir doit être interrogé dépend à la fois du comportement des sous-futurs et du flux de contrôle de la fonction. suit le point auquel le prochain devrait reprendre, et tout l'état local – variables, arguments, temporaires – dont la reprise aura

besoin. `cheapo_request` avait `cheapo_request` `cheapo_request` po  
11

La possibilité de suspendre l'exécution en milieu de fonction, puis de reprendre plus tard est unique aux fonctions asynchrones. Lorsqu'une fonction ordinaire revient, son cadre de pile a disparu pour de bon. Étant donné que les expressions dépendent de la possibilité de reprendre, vous ne pouvez les utiliser que dans des fonctions asynchrones. `await`

Au moment d'écrire ces lignes, Rust ne permet pas encore aux traits d'avoir des méthodes asynchrones. Seules les fonctions libres et inhérentes à un type spécifique peuvent être asynchrones. La levée de cette restriction nécessitera un certain nombre de modifications de la langue. En attendant, si vous devez définir des traits qui incluent des fonctions asynchrones, envisagez d'utiliser la caisse, qui fournit une solution de contournement basée sur des macros. `async-trait`

## Appel de fonctions asynchrones à partir de code synchrone : `block_on`

Dans un sens, les fonctions asynchrones ne font que se renvoyer la balle. Certes, il est facile d'obtenir la valeur d'un avenir dans une fonction asynchrone: juste elle. Mais la fonction asynchrone *elle-même* renvoie un avenir, c'est donc maintenant le travail de l'appelant de faire le sondage d'une manière ou d'une autre. En fin de compte, quelqu'un doit réellement attendre une valeur. `await`

Nous pouvons appeler à partir d'une fonction synchrone ordinaire (comme , par exemple) en utilisant la fonction de 's, qui prend un futur et l'interroge jusqu'à ce qu'elle produise une valeur: `cheapo_request main` `async_std task::block_on`

```
fn main() -> std::io::Result<()> {
 use async_std::task;

 let response = task::block_on(cheapo_request("example.com", 80, "/"));
 println!("{}: {}", Ok(()), response);
}
```

Puisqu'il s'agit d'une fonction synchrone qui produit la valeur finale d'une fonction asynchrone, vous pouvez la considérer comme un adaptateur du monde asynchrone au monde synchrone. Mais son caractère bloquant signifie également que vous ne devez jamais utiliser dans une fonction asynchrone: il bloquerait tout le thread jusqu'à ce que la valeur soit prête. Utilisez plutôt `block_on` ou `await`

[La figure 20-2](#) montre une exécution possible de `.main`

La chronologie supérieure, « Vue simplifiée », affiche une vue abstraite des appels asynchrones du programme : d'abord des appels pour obtenir un socket, puis des appels et sur ce socket. Puis il revient. Ceci est très similaire à la chronologie de la version synchrone de plus haut dans ce chapitre.

```
cheapo_request TcpStream::connect write_all ...
cheapo_request
Implementation
TcpStream::connect (returns future B)
A.poll
(returns future A)
cheapo_request
async_std::block_on
main
```

`cheapo_request`

`TcpStream::connect`

`write_all`

`cheapo_request`

`Implementation`

`TcpStream::connect`

`B.poll`

`(returns future B)`

`A.poll`

`cheapo_request`

`async_std::block_on`

`main`

`wakeup`

`(ready)`

`B.poll`

`write_all`

`C.poll`

`(returns future C)`

`A.poll`

`wakeup`

`(ready)`

`C.poll`

`write_all`

`D.poll`

`(returns future D)`

`A.poll`

`wakeup`

`(partial read; not ready)`

`D.poll`

`A.poll`

`wakeup`

`(partial read; not ready)`

`D.poll`

`(ready)`

`A.poll`

`wakeup`

`(ready)`

`D.poll`

`(ready)`

`A.poll`

`waiting`

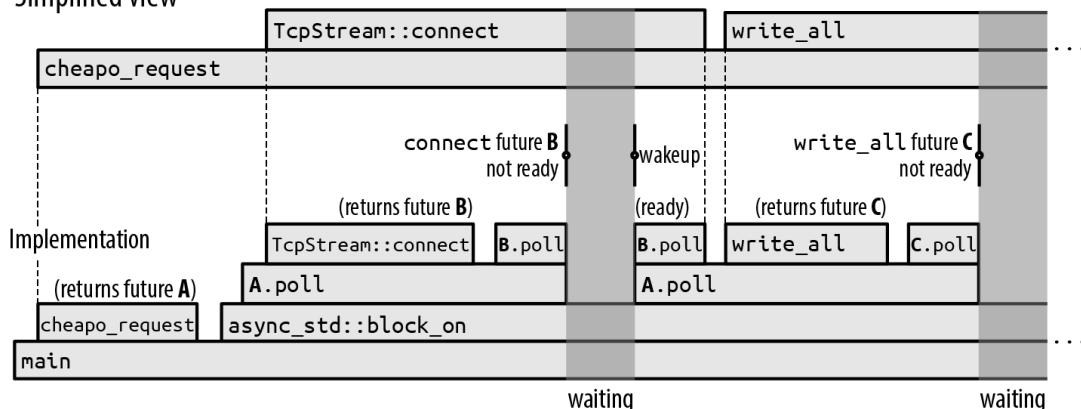
`waiting`

`waiting`

`waiting`

`waiting`

Simplified view



(Continued from above)

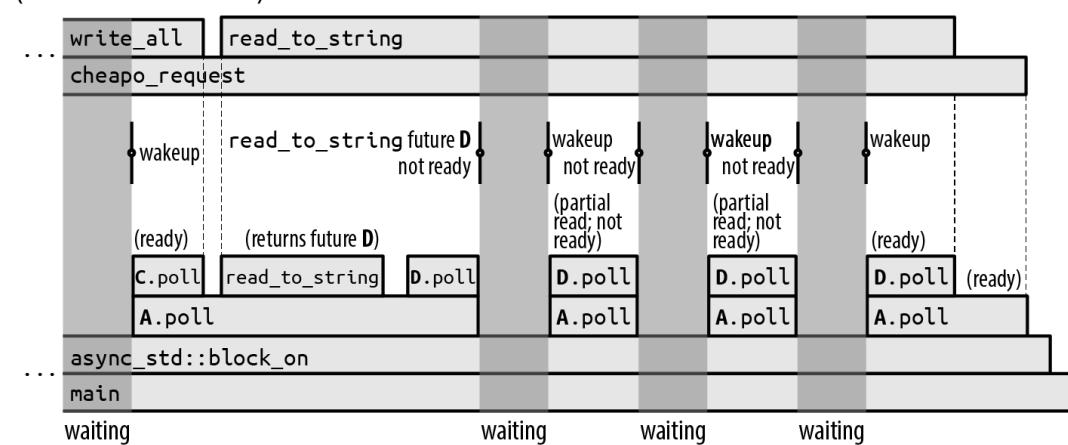


Figure 20-2. Blocage sur une fonction asynchrone

Mais chacun de ces appels asynchrones est un processus en plusieurs étapes : un futur est créé, puis interrogé jusqu'à ce qu'il soit prêt, créant et interrogeant peut-être d'autres sous-futurs dans le processus. La chronologie inférieure, « Implémentation », indique les appels synchrones réels qui implémentent ce comportement asynchrone. C'est une

bonne occasion de passer en revue exactement ce qui se passe dans l'exécution asynchrone ordinaire :

- Tout d'abord, les appels , qui renvoie l'avenir de son résultat final. Puis passe cet avenir à , qui le  
`sonde.main cheapo_request A main async_std::block_on`
- L'interrogation future permet au corps de commencer l'exécution. Il appelle à obtenir un avenir d'une prise et l'attend ensuite. Plus précisément, puisque pourrait rencontrer une erreur, est un futur d'un  
`.A cheapo_request TcpStream::connect B TcpStream::connect B R result<TcpStream, std::io::Error>`
- Future est sondé par le . Étant donné que la connexion réseau n'est pas encore établie, renvoie , mais s'arrange pour réveiller la tâche d'appel une fois que le socket est prêt.  
`B await B.poll Poll::Pending`
- Puisque l'avenir n'était pas prêt, retourne à son propre appelant,  
`.B A.poll Poll::Pending block_on`
- Puisqu'il n'a rien de mieux à faire, il s'endort. L'ensemble du thread est bloqué maintenant.  
`block_on`
- Lorsque la connexion est prête à être utilisée, elle réveille la tâche qui l'a interrogée. Cela se met en action, et il essaie de sonder à nouveau l'avenir.  
`B block_on A`
- Le sondage provoque la reprise dans son premier , où il sonde à nouveau.  
`A cheapo_request await B`
- Cette fois, est prêt : la création du socket est terminée, il revient donc à  
`.B Poll::Ready(Ok(socket)) A.poll`
- L'appel asynchrone à est maintenant terminé. La valeur de l'expression est donc  
`.TcpStream::connect TcpStream::connect(...).await Ok(socket)`
- L'exécution du corps de 'se déroule normalement, en créant la chaîne de requête à l'aide de la macro et en la transmettant à  
`.cheapo_request format! socket.write_all`
- Puisqu'il s'agit d'une fonction asynchrone, elle renvoie un futur de son résultat, qui attend dûment.  
`socket.write_all C cheapo_request`

Le reste de l'histoire est similaire. Dans l'exécution illustrée à [la figure 20-2](#), l'avenir de est interrogé quatre fois avant d'être prêt; chacun de ces réveils lit *certaines* données du socket, mais est spécifié pour lire jusqu'à la fin de l'entrée, ce qui nécessite plusieurs opérations.  
`socket.read_to_string read_to_string`

Il ne semble pas trop difficile d'écrire une boucle qui appelle encore et encore. Mais ce qui est précieux, c'est qu'il sait comment s'endormir jusqu'à ce que l'avenir vaille la peine d'être sondé à nouveau, plutôt que de perdre le temps de votre processeur et la durée de vie de votre batterie à faire des milliards d'appels infructueux. Les futurs renvoyés par les fonctions d'E/S de base comme et conservent le réveil fourni par le passé à et l'appellent quand doivent se réveiller et réessayer d'interroger. Nous montrerons exactement comment cela fonctionne en implémentant une version simple de nous-mêmes dans [« Primitive Futures and Executors: When Is a Future Worth Polling Again? »](#).

### When Is a Future Worth Polling Again?

```
poll async_std::task::block_on poll connect read_to
_string Context poll block_on block_on
```

Comme la version synchrone originale que nous avons présentée précédemment, cette version asynchrone passe presque tout son temps à attendre la fin des opérations. Si l'axe temporel était dessiné à l'échelle, le diagramme serait presque entièrement gris foncé, avec de minuscules éclats de calcul se produisant lorsque le programme est réveillé. `cheapo_request`

C'est beaucoup de détails. Heureusement, vous pouvez généralement penser en termes de chronologie supérieure simplifiée: certains appels de fonction sont synchronisés, d'autres sont asynchrones et ont besoin d'un , mais ce ne sont que des appels de fonction. Le succès du support asynchrone de Rust dépend de l'aide aux programmeurs pour travailler avec la vue simplifiée dans la pratique, sans être distraits par les allers-retours de l'implémentation. `await`

## Génération de tâches asynchrones

La fonction bloque jusqu'à ce que la valeur d'un futur soit prête. Mais bloquer complètement un thread sur un seul futur n'est pas mieux qu'un appel synchrone : le but de ce chapitre est de faire en sorte que le thread *fasse un autre travail* pendant qu'il attend. `async_std::task::block_on`

Pour cela, vous pouvez utiliser . Cette fonction prend un avenir et l'ajoute à un pool qui essaiera d'interroger chaque fois que l'avenir sur lequel elle bloque n'est pas prêt. Donc, si vous passez un tas de futurs à et appliquez ensuite à un futur de votre résultat final, interrogez chaque futur engendré chaque fois qu'il sera en mesure de progresser, en exécutant l'ensemble du pool simultanément jusqu'à ce que votre résultat soit

```
prêt.async_std::task::spawn_local block_on spawn_local bloc
k_on block_on
```

Au moment d'écrire ces lignes, n'est disponible que si vous activez la fonctionnalité de cette caisse. Pour ce faire, vous devrez vous référer à dans votre *Cargo.toml* avec une ligne comme celle-ci: `spawn_local async-std unstable async-std`

```
async-std = { version = "1", features = ["unstable"] }
```

La fonction est un analogue asynchrone de la fonction de la bibliothèque standard pour le démarrage des threads

```
: spawn_local std::thread::spawn
```

- `std::thread::spawn(c)` prend une fermeture et démarre un thread qui l'exécute, en renvoyant une méthode dont attend la fin du thread et renvoie tout ce qui a été renvoyé. `c std::thread::JoinHandle join c`
- `async_std::task::spawn_local(f)` prend l'avenir et l'ajoute au pool à interroger lorsque le thread actuel appelle `.join`. renvoie son propre type, lui-même un futur que vous pouvez attendre pour récupérer la valeur finale de

```
.f block_on spawn_local async_std::task::JoinHandle f
```

Par exemple, supposons que nous voulions créer simultanément un ensemble complet de requêtes HTTP. Voici une première tentative :

```
pub async fn many_requests(requests: Vec<(String, u16, String)>)
 -> Vec<std::io::Result<String>>
{
 use async_std::task;

 let mut handles = vec![];
 for (host, port, path) in requests {
 handles.push(task::spawn_local(cheapo_request(&host, port, &path));
 }

 let mut results = vec![];
 for handle in handles {
 results.push(handle.await);
 }

 results
}
```

Cette fonction appelle chaque élément de , passant l'avenir de chaque appel à . Il recueille les résultats dans un vecteur puis attend chacun d'eux. Il est bon d'attendre les poignées de jointure dans n'importe quel ordre : puisque les requêtes sont déjà générées, leurs futurs seront interrogés au besoin chaque fois que ce thread appelle et n'a rien de mieux à faire. Toutes les demandes s'exécuteront simultanément. Une fois qu'ils sont terminés, renvoie les résultats à son appelant.

```
cheapo_request requests spawn_local JoinHandle block_on many_requests
```

Le code précédent est presque correct, mais le vérificateur d'emprunt de Rust s'inquiète de la durée de vie de l'avenir de Rust: cheapo\_request

```
error: `host` does not live long enough

handles.push(task::spawn_local(cheapo_request(&host, port, &path)));
 ^^^^^^-----
| |
| | borrowed value does not
| | live long enough
| argument requires that `host` is borrowed for `*state`
}
- `host` dropped here while still borrowed
```

Il y a une erreur similaire pour aussi. path

Naturellement, si nous transmettons des références à une fonction asynchrone, l'avenir qu'elle renvoie doit contenir ces références, de sorte que l'avenir ne peut pas survivre en toute sécurité aux valeurs qu'ils empruntent. Il s'agit de la même restriction qui s'applique à toute valeur qui contient des références.

Le problème est que vous ne pouvez pas être sûr que vous attendrez que la tâche se termine avant et que vous êtes abandonné. En fait, n'accepte que les futurs dont la durée de vie est , car vous pouvez simplement ignorer les retours et laisser la tâche continuer à s'exécuter pour le reste de l'exécution du programme. Ce n'est pas propre aux tâches asynchrones : vous obtiendrez une erreur similaire si vous essayez d'utiliser pour démarrer un thread dont la fermeture capture des références à des variables

```
locales.spawn_local host path spawn_local 'static JoinHandle static td::thread::spawn
```

Une façon de résoudre ce problème consiste à créer une autre fonction asynchrone qui prend les versions propriétaires des arguments :

```
async fn cheapo_owning_request(host: String, port: u16, path: String)
 -> std::io::Result<String> {
 cheapo_request(&host, port, &path).await
}
```

Cette fonction prend s au lieu de références, de sorte que son futur possède le et les cordes lui-même, et sa durée de vie est . Le vérificateur d'emprunt peut voir qu'il attend immédiatement l'avenir de 's, et par conséquent, si cet avenir est sondé du tout, les variables qu'il emprunte doivent toujours exister. Tout va

bien. String &str host path 'static cheapo\_request host path

En utilisant , vous pouvez générer toutes vos demandes comme suit  
: cheapo\_owning\_request

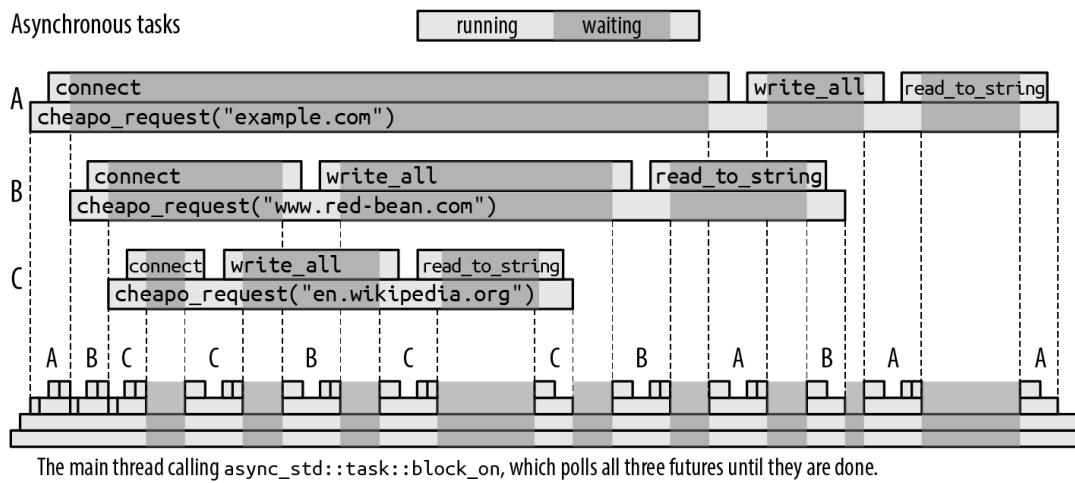
```
for (host, port, path) in requests {
 handles.push(task::spawn_local(cheapo_owning_request(host, port, path)));
}
```

Vous pouvez appeler depuis votre fonction synchrone, avec  
: many\_requests main block\_on

```
let requests = vec![
 ("example.com".to_string(), 80, "/".to_string()),
 ("www.red-bean.com".to_string(), 80, "/".to_string()),
 ("en.wikipedia.org".to_string(), 80, "/".to_string()),
];
let results = async_std::task::block_on(many_requests(requests));
for result in results {
 match result {
 Ok(response) => println!("{}: {}", response),
 Err(err) => eprintln!("error: {}", err),
 }
}
```

Ce code exécute les trois demandes simultanément à partir de l'appel à . Chacun progresse au fur et à mesure que l'occasion se présente tandis que les autres sont bloqués, le tout sur le fil d'appel. [La figure 20-3](#) montre une exécution possible des trois appels à .block\_on cheapo\_request

(Nous vous encourageons à essayer d'exécuter ce code vous-même, avec des appels ajoutés en haut et après chaque expression afin que vous puissiez voir comment les appels s'entrelacent différemment d'une exécution à l'autre.) `eprintln! cheapo_request await`



Graphique 20-3. Exécution de trois tâches asynchrones sur un seul thread

L'appel à (non affiché, pour plus de simplicité) a engendré trois tâches asynchrones, que nous avons étiquetées , et . commence par l'interrogation , qui commence à se connecter à . Dès que cela revient, tourne son attention vers la tâche suivante engendrée, l'interrogation future , et finalement , qui commencent chacun à se connecter à leurs serveurs respectifs. `many_requests A B C block_on A example.com Poll::Pending block_on B C`

Lorsque tous les futurs interrogables sont revenus, s'endort jusqu'à ce que l'un des futurs indique que sa tâche vaut la peine d'être sondée à nouveau. `Poll::Pending block_on TcpStream::connect`

Dans cette exécution, le serveur répond plus rapidement que les autres, de sorte que la tâche se termine en premier. Lorsqu'une tâche générée est terminée, elle enregistre sa valeur dans la sienne et la marque comme prête, afin qu'elle puisse continuer quand elle l'attend. Finalement, les autres appels à réussiront ou retourneront une erreur, et lui-même peut revenir. Enfin, reçoit le vecteur de résultats de

```
.en.wikipedia.org JoinHandle many_requests cheapo_request many_requests main block_on
```

Toute cette exécution se déroule sur un seul fil, les trois appels à être entrelacés les uns avec les autres à travers des sondages successifs de leur avenir. Un appel asynchrone offre l'apparence d'un appel de fonction unique s'exécutant jusqu'à la fin, mais cet appel asynchrone est réalisé par une série d'appels synchrones à la méthode du futur. Chaque appel

individuel revient rapidement, ce qui donne le fil de discussion afin qu'un autre appel asynchrone puisse prendre un tour. `cheapo_request poll poll`

Nous avons enfin atteint l'objectif que nous nous étions fixé au début du chapitre : laisser un thread prendre en charge d'autres tâches en attendant la fin des E/S afin que les ressources du thread ne soient pas bloquées à ne rien faire. Mieux encore, cet objectif a été atteint avec du code qui ressemble beaucoup au code Rust ordinaire: certaines des fonctions sont marquées, certains des appels de fonction sont suivis de , et nous utilisons des fonctions de au lieu de , mais sinon, c'est du code Rust ordinaire. `async .await async_std std`

Une différence importante à garder à l'esprit entre les tâches asynchrones et les threads est que le passage d'une tâche asynchrone à une autre ne se produit qu'au niveau des expressions, lorsque l'avenir attendu du revient . Cela signifie que si vous mettez un calcul de longue durée dans , aucune des autres tâches que vous avez passées n'aura la chance de s'exécuter tant qu'elle n'est pas terminée. Avec les threads, ce problème ne se pose pas : le système d'exploitation peut suspendre n'importe quel thread à n'importe quel point et définit des minuteries pour s'assurer qu'aucun thread ne monopolise le processeur. Le code asynchrone dépend de la coopération volontaire des futurs partageant le thread. Si vous avez besoin que des calculs de longue durée coexistent avec du code asynchrone, [« Long Running Computations: yield now and spawn blocking »](#) plus loin dans ce chapitre décrit certaines options. `await Poll::Pending cheapo_request spawn_local`

## Blocs asynchrones

En plus des fonctions asynchrones, Rust prend également en charge *les blocs asynchrones*. Alors qu'une instruction de bloc ordinaire renvoie la valeur de sa dernière expression, un bloc asynchrone renvoie *un futur de* la valeur de sa dernière expression. Vous pouvez utiliser des expressions dans un bloc asynchrone. `await`

Un bloc asynchrone ressemble à une instruction de bloc ordinaire, précédée du mot-clé : `async`

```
let serve_one = async {
 use async_std::net;

 // Listen for connections, and accept one.
```

```

let listener = net:::TcpListener:::bind("localhost:8087").await?;
let (mut socket, _addr) = listener.accept().await?;

 // Talk to client on `socket`.
 ...
};

}

```

Cela s'initialise avec un futur qui, lorsqu'il est interrogé, écoute et gère une seule connexion TCP. Le corps du bloc ne commence pas l'exécution tant qu'il n'est pas interrogé, tout comme un appel de fonction asynchrone ne commence pas à s'exécuter tant que son avenir n'est pas interrogé. `serve_one`

Si vous appliquez l'opérateur à une erreur dans un bloc asynchrone, il revient simplement à partir du bloc, pas de la fonction environnante. Par exemple, si l'appel précédent renvoie une erreur, l'opérateur la renvoie comme valeur finale. De même, les expressions reviennent du bloc asynchrone, et non de la fonction englobante. `? bind ? serve_one return`

Si un bloc asynchrone fait référence à des variables définies dans le code environnant, son avenir capture leurs valeurs, tout comme le ferait une fermeture. Et tout comme les fermetures ([voir « Fermetures qui volent »](#)), vous pouvez commencer le bloc avec `pour` pour prendre possession des valeurs capturées, plutôt que de simplement conserver des références à celles-ci. `move`

Les blocs asynchrones offrent un moyen concis de séparer une section de code que vous souhaitez exécuter de manière asynchrone. Par exemple, dans la section précédente, nécessitait un futur, nous avons donc défini la fonction wrapper pour nous donner un futur qui s'appropriait ses arguments. Vous pouvez obtenir le même effet sans la distraction d'une fonction wrapper simplement en appelant à partir d'un bloc asynchrone : `: spawn_local`

```

static cheapo_owning_request cheapo_request

pub async fn many_requests(requests: Vec<(String, u16, String)>)
 -> Vec<std::io::Result<String>>
{
 use async_std::task;

 let mut handles = vec![];
 for (host, port, path) in requests {
 handles.push(task:::spawn_local(async move {
 cheapo_request(&host, port, &path).await
 }));
 }
}

```

```
...
}
```

Puisqu'il s'agit d'un bloc, son avenir s'approprie les valeurs et, comme le ferait une fermeture. Il transmet ensuite les références à . Le vérificateur d'emprunt peut voir que l'expression du bloc prend possession de l'avenir de ', de sorte que les références aux variables capturées qu'ils empruntent et ne peuvent pas survivre. Le bloc asynchrone accomplit la même chose que , mais avec moins de passe-partout.

```
async
move String host path move cheapo_request await cheapo_request host path cheapo_owning_request
```

Un inconvénient que vous pouvez rencontrer est qu'il n'y a pas de syntaxe pour spécifier le type de retour d'un bloc asynchrone, analogue aux arguments suivants d'une fonction asynchrone. Cela peut entraîner des problèmes lors de l'utilisation de l'opérateur : -> T ?

```
let input = async_std::io::stdin();
let future = async {
 let mut line = String::new();

 // This returns `std::io::Result<usize>`.
 input.read_line(&mut line).await?;

 println!("Read line: {}", line);

 Ok()
};
```

Cela échoue avec l'erreur suivante :

```
error: type annotations needed
|
48 | let future = async {
| ----- consider giving `future` a type
|
|
|
60 | Ok()
| ^^^ cannot infer type for type parameter `E` declared
| on the enum `Result`
```

Rust ne peut pas dire quel doit être le type de retour du bloc asynchrone. La méthode renvoie , mais comme l'opérateur utilise le trait pour convertir le type d'erreur en question en fonction de la situation requise, le type de retour du bloc asynchrone peut être pour n'importe quel type qui im-

```
plémante.read_line Result<(), std::io::Error> ?
From Result<(), E> E From<std::io::Error>
```

Les futures versions de Rust ajouteront probablement une syntaxe pour indiquer le type de retour d'un bloc. Pour l'instant, vous pouvez contourner le problème en épelant le type de la finale du bloc : `async Ok`

```
let future = async {
 ...
 Ok::<(), std::io::Error>(())
};
```

Étant donné qu'il s'agit d'un type générique qui attend les types de réussite et d'erreur comme ses paramètres, nous pouvons spécifier ces paramètres de type lors de l'utilisation ou comme indiqué ici. `Result Ok Err`

## Création de fonctions asynchrones à partir de blocs asynchrones

Les blocs asynchrones nous donnent un autre moyen d'obtenir le même effet qu'une fonction asynchrone, avec un peu plus de flexibilité. Par exemple, nous pourrions écrire notre exemple comme une fonction synchrone ordinaire qui renvoie l'avenir d'un bloc asynchrone

```
: cheapo_request
```

```
use std::io;
use std::future::Future;

fn cheapo_request<'a>(host: &'a str, port: u16, path: &'a str)
 -> impl Future<Output = io::Result<String>> + 'a
{
 async move {
 ... function body ...
 }
}
```

Lorsque vous appelez cette version de la fonction, elle renvoie immédiatement l'avenir de la valeur du bloc asynchrone. Cela capture les arguments de la fonction et se comporte comme le futur que la fonction asynchrone aurait renvoyé. Comme nous n'utilisons pas la syntaxe, nous devons écrire le dans le type de retour, mais en ce qui concerne les appelants, ces deux définitions sont des implementations interchangeables de la même signature de fonction. `async fn impl Future`

Cette deuxième approche peut être utile lorsque vous souhaitez effectuer un calcul immédiatement lorsque la fonction est appelée, avant de créer l'avenir de son résultat. Par exemple, une autre façon de se réconcilier avec serait d'en faire une fonction synchrone renvoyant un futur qui capture des copies entièrement détenues de ses arguments

```
:cheapo_request spawn_local 'static

fn cheapo_request(host: &str, port: u16, path: &str)
 -> impl Future<Output = io::Result<String>> + 'static
{
 let host = host.to_string();
 let path = path.to_string();

 async move {
 ... use &*host, port, and path ...
 }
}
```

Cette version permet au bloc asynchrone de capturer et en tant que valeurs possédées, et non en tant que références. Étant donné que le futur possède toutes les données dont il a besoin pour s'exécuter, il est valable pour la durée de vie. (Nous l'avons précisé dans la signature montrée précédemment, mais c'est la valeur par défaut pour les types de retour, donc l'omettre n'aurait aucun

```
effet.) host path String &str 'static + 'static 'static ->
impl
```

Puisque cette version des rendements futures qui sont , on peut les passer directement à :cheapo\_request 'static spawn\_local

```
let join_handle = async_std::task::spawn_local(
 cheapo_request("areweasyncyet.rs", 80, "/")
);

... other work ...

let response = join_handle.await?;
```

## Génération de tâches asynchrones sur un pool de threads

Les exemples que nous avons montrés jusqu'à présent passent presque tout leur temps à attendre les E/S, mais certaines charges de travail sont plutôt un mélange de travail de processeur et de blocage. Lorsque vous

avez suffisamment de calcul pour faire ce qu'un seul processeur ne peut pas suivre, vous pouvez l'utiliser pour générer un avenir sur un pool de threads de travail dédiés à l'interrogation des futurs qui sont prêts à progresser. `async_std::task::spawn`

`async_std::task::spawn` est utilisé comme  
`:async_std::task::spawn_local`

```
use async_std::task;

let mut handles = vec![];
for (host, port, path) in requests {
 handles.push(task::spawn(async move {
 cheapo_request(&host, port, &path).await
 }));
}
...
...
```

Comme , renvoie une valeur que vous pouvez attendre pour obtenir la valeur finale de l'avenir. Mais contrairement à , l'avenir n'a pas besoin d'attendre que vous appeliez avant d'être interrogé. Dès que l'un des threads du pool de threads est libre, il essaiera de

l'interroger. `spawn_local` `spawn JoinHandle` `spawn_local block_on`

En pratique, est plus largement utilisé que , simplement parce que les gens aiment savoir que leur charge de travail, quel que soit son mélange de calcul et de blocage, est équilibrée dans les ressources de la machine. `spawn spawn_local`

Une chose à garder à l'esprit lors de l'utilisation est que le pool de threads essaie de rester occupé, de sorte que votre avenir est interrogé par le thread qui s'en occupe en premier. Un appel asynchrone peut commencer l'exécution sur un thread, bloquer sur une expression et être repris dans un autre thread. Ainsi, bien qu'il soit raisonnable de simplifier de considérer un appel de fonction asynchrone comme une exécution unique et connectée de code (en effet, le but des fonctions et expressions asynchrones est de vous encourager à y penser de cette façon), l'appel peut en fait être effectué par de nombreux threads différents. `spawn await await`

Si vous utilisez le stockage local de thread, il peut être surprenant de voir les données que vous y placez avant une expression remplacées par quelque chose de complètement différent par la suite, car votre tâche est maintenant interrogée par un thread différent du pool. S'il s'agit d'un

problème, vous devez plutôt utiliser *le stockage local de tâche* ; voir la documentation de la caisse pour la macro pour plus de détails. `await async_std::task::spawn_local!`

## Mais votre future implémentation envoie-t-elle ?

Il y a une restriction qui ne l'impose pas. Étant donné que l'avenir est envoyé à un autre thread pour s'exécuter, le futur doit implémenter le trait de marqueur. Nous avons présenté dans [« Thread Safety: Send and Sync »](#). Un futur n'est que si toutes les valeurs qu'il contient le sont : tous les arguments de fonction, les variables locales et même les valeurs temporaires anonymes doivent pouvoir être déplacés en toute sécurité vers un autre thread. `spawn spawn_local Send Send Send Send`

Comme précédemment, cette exigence n'est pas propre aux tâches asynchrones : vous obtiendrez une erreur similaire si vous essayez de l'utiliser pour démarrer un thread dont la fermeture capture des valeurs non. La différence est que, alors que la fermeture passée reste sur le thread qui a été créé pour l'exécuter, un futur engendré sur un pool de threads peut se déplacer d'un thread à l'autre à tout moment. `std::thread::spawn Send std::thread::spawn`

Cette restriction est facile à contourner par accident. Par exemple, le code suivant semble assez innocent :

```
use async_std::task;
use std::rc::Rc;

async fn reluctant() -> String {
 let string = Rc::new("ref-counted string".to_string());

 some_asynchronous_thing().await;

 format!("Your splendid string: {}", string)
}

task::spawn(reluctant());
```

L'avenir d'une fonction asynchrone doit contenir suffisamment d'informations pour que la fonction puisse continuer à partir d'une expression. Dans ce cas, le futur de ' doit utiliser après le , de sorte que le futur contiendra, au moins parfois, une valeur. Étant donné que les pointeurs ne peuvent pas être partagés en toute sécurité entre les threads, l'avenir lui-même ne peut pas être . Et puisque n'accepte que les futurs qui sont , les

objets Rust

```
: await reluctant string await Rc<String> Rc Send spawn Send

error: future cannot be sent between threads safely
|
17 | task::spawn(reluctant());
| ^^^^^^^^^^^^ future returned by `reluctant` is not `Send`
|
|
127 | T: Future + Send + 'static,
| ---- required by this bound in `async_std::task::spaw
|
= help: within `impl Future`, the trait `Send` is not implemented
 for `Rc<String>`

note: future is not `Send` as this value is used across an await
|
10 | let string = Rc::new("ref-counted string".to_string());
| ----- has type `Rc<String>` which is not `Send`
11 |
12 | some_asynchronous_thing().await;
| ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
|
| await occurs here, with `string` maybe used later
...
15 | }
| - `string` is later dropped here
```

Ce message d'erreur est long, mais il contient de nombreux détails utiles :

- Il explique pourquoi l'avenir doit être : l'exige. `Send` `task::spawn`
- Il explique quelle valeur n'est pas : la variable locale , dont le type est `.Send string Rc<String>`
- Il explique pourquoi affecte l'avenir: il est dans la portée à travers le `.string await`

Il existe deux façons de résoudre ce problème. L'une consiste à restreindre la portée de la non-valeur afin qu'elle ne couvre aucune expression et n'ait donc pas besoin d'être enregistrée dans le futur de la fonction  
`: Send await`

```
async fn reluctant() -> String {
 let return_value = {
 let string = Rc::new("ref-counted string".to_string());
 format!("Your splendid string: {}", string)
 // The `Rc<String>` goes out of scope here...
 };
}
```

```

 // ... and thus is not around when we suspend here.
 some_asynchronous_thing().await;

 return_value
}

```

Une autre solution consiste simplement à utiliser à la place de . utilise des mises à jour atomiques pour gérer ses nombres de références, ce qui le rend un peu plus lent, mais les pointeurs sont

```
.std::sync::Arc Rc Arc Arc Send
```

Bien que vous finirez par apprendre à reconnaître et à éviter les non-types, ils peuvent être un peu surprenants au début. (Au moins, vos auteurs ont souvent été surpris.) Par exemple, l'ancien code Rust utilise parfois des types de résultats génériques comme celui-ci : Send

```

// Not recommended!
type GenericError = Box<dyn std::error::Error>;
type GenericResult<T> = Result<T, GenericError>;

```

Ce type utilise un objet trait encadré pour contenir une valeur de n'importe quel type qui implémente . Mais cela n'impose pas d'autres restrictions: si quelqu'un avait un non-type implémenté, il pourrait convertir une valeur encadrée de ce type en un . En raison de cette possibilité, n'est pas , et le code suivant ne fonctionnera

```
pas: GenericError std::error::Error Send Error GenericError G
enericError Send
```

```

fn some_fallible_thing() -> GenericResult<i32> {
 ...
}

// This function's future is not `Send`...
async fn unfortunate() {
 // ... because this call's value ...
 match some_fallible_thing() {
 Err(error) => {
 report_error(error);
 }
 Ok(output) => {
 // ... is alive across this await ...
 use_output(output).await;
 }
 }
}

```

```
 }

 // ... and thus this `spawn` is an error.
 async_std::task::spawn(unfortunate());
```

Comme dans l'exemple précédent, le message d'erreur du compilateur explique ce qui se passe, en pointant vers le type comme coupable. Puisque Rust considère que le résultat de est présent pour l'ensemble de l'instruction, y compris l'expression, il détermine que l'avenir de n'est pas . Cette erreur est trop prudente de la part de Rust: bien qu'il soit vrai qu'il n'est pas sûr d'envoyer à un autre thread, le seul se produit lorsque le résultat est , de sorte que la valeur d'erreur n'existe jamais réellement lorsque nous attendons l'avenir de

```
.Result some_fallible_thing match await unfortunate Send GenericError await Ok use_output
```

La solution idéale consiste à utiliser des types d'erreur génériques plus stricts comme ceux que nous avons suggérés dans [« Travailler avec plusieurs types d'erreurs »](#):

```
type GenericError = Box<dyn std::error::Error + Send + Sync + 'static>;
type GenericResult<T> = Result<T, GenericError>;
```

Cet objet de trait nécessite explicitement le type d'erreur sous-jacent pour être implémenté , et tout va bien. Send

Si votre avenir ne l'est pas et que vous ne pouvez pas le faire facilement, vous pouvez toujours l'utiliser pour l'exécuter sur le fil de discussion actuel. Bien sûr, vous devrez vous assurer que le thread appelle à un moment donné, pour lui donner une chance de s'exécuter, et vous ne bénéfieriez pas de la distribution du travail sur plusieurs processeurs. Send spawn\_local block\_on

## Calculs de longue durée : yield\_now et spawn\_blocking

Pour qu'un futur partage bien son fil avec d'autres tâches, sa méthode doit toujours revenir le plus rapidement possible. Mais si vous effectuez un long calcul, cela peut prendre beaucoup de temps pour atteindre le suivant, ce qui fait que d'autres tâches asynchrones attendent plus longtemps que vous ne le souhaiteriez pour leur activation du thread. poll await

Une façon d'éviter cela est simplement de faire quelque chose de temps en temps. La fonction renvoie un futur simple conçu pour cela

```
: await async_std::task::yield_now
```

```
while computation_not_done() {
 ... do one medium-sized step of computation ...
 async_std::task::yield_now().await;
}
```

La première fois que l'avenir est sondé, il revient, mais dit que cela vaut la peine de sonder à nouveau bientôt. L'effet est que votre appel asynchrone abandonne le thread et que d'autres tâches ont une chance de s'exécuter, mais votre appel aura bientôt un autre tour. La deuxième fois, l'avenir de 's est interrogé, il retourne et votre fonction asynchrone peut reprendre

```
l'exécution. yield_now Poll::Pending yield_now Poll::Ready(())
```

Cependant, cette approche n'est pas toujours réalisable. Si vous utilisez une caisse externe pour effectuer le calcul de longue durée ou appelez C ou C ++, il peut ne pas être pratique de modifier ce code pour qu'il soit plus convivial. Ou il peut être difficile de s'assurer que chaque chemin à travers le calcul est sûr de frapper le de temps en temps. await

Pour des cas comme celui-ci, vous pouvez utiliser . Cette fonction prend une fermeture, la démarre en cours d'exécution sur son propre thread et renvoie un futur de sa valeur de retour. Le code asynchrone peut attendre cet avenir, cédant son thread à d'autres tâches jusqu'à ce que le calcul soit prêt. En mettant le travail difficile sur un thread séparé, vous pouvez laisser le système d'exploitation s'occuper de le faire partager le processeur gentiment. `async_std::task::spawn_blocking`

Par exemple, supposons que nous devions vérifier les mots de passe fournis par les utilisateurs par rapport aux versions hachées que nous avons stockées dans notre base de données d'authentification. Pour des raisons de sécurité, la vérification d'un mot de passe doit être intensive en calcul afin que, même si les attaquants obtiennent une copie de notre base de données, ils ne puissent pas simplement essayer des milliards de mots de passe possibles pour voir s'ils correspondent. La caisse fournit une fonction de hachage conçue spécifiquement pour stocker les mots de passe: un hachage correctement généré prend une fraction de seconde importante à vérifier. Nous pouvons utiliser (version ) dans notre application asynchrone comme ceci:

```
argonautica argonautica argonautica 0.2
```

```

async fn verify_password(password: &str, hash: &str, key: &str)
 -> Result<bool, argonautica::Error>
{
 // Make copies of the arguments, so the closure can be 'static.
 let password = password.to_string();
 let hash = hash.to_string();
 let key = key.to_string();

 async_std::task::spawn_blocking(move || {
 argonautica::Verifier::default()
 .with_hash(hash)
 .with_password(password)
 .with_secret_key(key)
 .verify()
 }).await
}

```

Cela renvoie si correspond , étant donné , une clé pour la base de données dans son ensemble. En effectuant la vérification dans la fermeture passée à , nous poussons le calcul coûteux sur son propre thread, en veillant à ce qu'il n'affecte pas notre réactivité aux demandes des autres utilisateurs. Ok(true) password hash key spawn\_blocking

## Comparaison des conceptions asynchrones

À bien des égards, l'approche de Rust en matière de programmation asynchrone ressemble à celle adoptée par d'autres langages. Par exemple, JavaScript, C# et Rust ont tous des fonctions asynchrones avec des expressions. Et tous ces langages ont des valeurs qui représentent des calculs incomplets : Rust les appelle « futures », JavaScript les appelle « promesses » et C# les appelle « tâches », mais ils représentent tous une valeur que vous devrez peut-être attendre. await

L'utilisation des sondages par Rust, cependant, est inhabituelle. En JavaScript et C#, une fonction asynchrone commence à s'exécuter dès qu'elle est appelée, et une boucle d'événements globale est intégrée à la bibliothèque système qui reprend les appels de fonction asynchrone suspendus lorsque les valeurs qu'ils attendaient deviennent disponibles. Dans Rust, cependant, un appel asynchrone ne fait rien jusqu'à ce que vous le transmettiez à une fonction comme , , ou qui l'interroge et conduise le travail à son terme. Ces fonctions, *appelées exéuteurs*, jouent le rôle que d'autres langages courent avec une boucle d'événements globale. block\_on spawn spawn\_local

Parce que Rust vous oblige, en tant que programmeur, à choisir un exécuteur pour sonder vos avenirs, Rust n'a pas besoin d'une boucle d'événements globale intégrée au système. La caisse offre les fonctions d'exécuteur que nous avons utilisées dans ce chapitre jusqu'à présent, mais la caisse, que nous utiliserons plus loin dans ce chapitre, définit son propre ensemble de fonctions d'exécuteur similaires. Et vers la fin de ce chapitre, nous mettrons en œuvre notre propre exécuteur testamentaire. Vous pouvez utiliser les trois dans le même programme. `async-std` `tokio`

## Un véritable client HTTP asynchrone

Nous serions négligents si nous ne montrions pas un exemple d'utilisation d'une caisse client HTTP asynchrone appropriée, car c'est si facile, et il y a plusieurs bonnes caisses à choisir, y compris `et`. `reqwest` `surf`

Voici une réécriture de `, encore plus simple que celle basée sur , qui permet d'exécuter une série de requêtes simultanément. Vous aurez besoin de ces dépendances dans votre fichier Cargo.toml`

```
:many_requests cheapo_request surf
```

```
[dependencies]
async-std = "1.7"
surf = "1.0"
```

Ensuite, nous pouvons définir comme suit: `many_requests`

```
pub async fn many_requests(urls: &[String])
 -> Vec<Result<String, surf::Exception>>
{
 let client = surf::Client::new();

 let mut handles = vec![];
 for url in urls {
 let request = client.get(&url).recv_string();
 handles.push(async_std::task::spawn(request));
 }

 let mut results = vec![];
 for handle in handles {
 results.push(handle.await);
 }

 results
}
```

```

fn main() {
 let requests = &["http://example.com".to_string(),
 "https://www.red-bean.com".to_string(),
 "https://en.wikipedia.org/wiki/Main_Page".to_string()

 let results = async_std::task::block_on(many_requests(requests));
 for result in results {
 match result {
 Ok(response) => println!("*** {}\n", response),
 Err(err) => eprintln!("error: {}\n", err),
 }
 }
}

```

L'utilisation d'un seul pour effectuer toutes nos requêtes nous permet de réutiliser les connexions HTTP si plusieurs d'entre elles sont dirigées vers le même serveur. Et aucun bloc asynchrone n'est nécessaire : puisqu'il s'agit d'une méthode asynchrone qui renvoie un futur, on peut passer son futur directement à `.surf::Client recv_string Send + 'static spawn`

## Un client et un serveur asynchrones

Il est temps de prendre les idées clés dont nous avons discuté jusqu'à présent et de les rassembler dans un programme de travail. Dans une large mesure, les applications asynchrones ressemblent à des applications multithread ordinaires, mais il existe de nouvelles possibilités de code compact et expressif que vous pouvez rechercher.

L'exemple de cette section est un serveur et un client de chat. Consultez le [code complet](#). Les vrais systèmes de chat sont compliqués, avec des préoccupations allant de la sécurité et de la reconnexion à la confidentialité et à la modération, mais nous avons réduit les nôtres à un ensemble austère de fonctionnalités afin de nous concentrer sur quelques points d'intérêt.

En particulier, nous voulons bien gérer *la contre-pression*. Nous entendons par là que si un client a une connexion Internet lente ou abandonne complètement sa connexion, cela ne doit jamais affecter la capacité des autres clients à échanger des messages à leur propre rythme. Et comme un client lent ne doit pas obliger le serveur à dépenser de la mémoire illimitée en conservant son arriéré de messages toujours croissant, notre serveur doit supprimer les messages pour les clients qui ne peuvent pas suivre, mais les informer que leur flux est incomplet. (Un vrai serveur de

chat enregistrerait les messages sur le disque et permettrait aux clients de récupérer ceux qu'ils ont manqués, mais nous avons laissé cela de côté.)

Nous commençons le projet avec la commande et mettons le texte suivant dans *async-chat/Cargo.toml*: cargo new --lib async-chat

```
[package]
name = "async-chat"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2021"

[dependencies]
async-std = { version = "1.7", features = ["unstable"] }
tokio = { version = "1.0", features = ["sync"] }
serde = { version = "1.0", features = ["derive", "rc"] }
serde_json = "1.0"
```

Nous dépendons de quatre caisses :

- La caisse est la collection de primitives d'E/S asynchrones et d'utilitaires que nous avons utilisés tout au long du chapitre. `async-std`
- La caisse est une autre collection de primitives asynchrones comme , l'une des plus anciennes et des plus matures. Il est largement utilisé et maintient sa conception et sa mise en œuvre à des normes élevées, mais nécessite un peu plus de soin à utiliser que . `tokio async-std`  
`tokio` est une grande caisse, mais nous n'en avons besoin que d'un seul composant, de sorte que le champ de la ligne de dépendance *Cargo.toml* se réduit aux pièces dont nous avons besoin, ce qui en fait une dépendance légère. `features = ["sync"] tokio`  
Lorsque l'écosystème des bibliothèques asynchrones était moins mature, les gens évitaient d'utiliser les deux et dans le même programme, mais les deux projets ont coopéré pour s'assurer que cela fonctionne, tant que les règles documentées de chaque caisse sont suivies. `tokio async-std`
- Les caisses que nous avons déjà vues, au [chapitre 18](#). Ceux-ci nous donnent des outils pratiques et efficaces pour générer et analyser JSON, que notre protocole de chat utilise pour représenter les données sur le réseau. Nous voulons utiliser certaines fonctionnalités facultatives de , nous les sélectionnons donc lorsque nous donnons la dépendance. `serde serde_json serde`

Toute la structure de notre application de chat, client et serveur, ressemble à ceci:

```
async-chat
├── Cargo.toml
└── src
 ├── lib.rs
 └── utils.rs
 └── bin
 ├── client.rs
 └── server
 ├── main.rs
 ├── connection.rs
 ├── group.rs
 └── group_table.rs
```

Cette mise en page de paquet utilise une fonctionnalité Cargo que nous avons abordée dans [« Le répertoire src / bin »](#): en plus de la caisse de bibliothèque principale, *src / lib.rs*, avec son sous-module *src / utils.rs*, il comprend également deux exécutables:

- *src/bin/client.rs* est un exécutable à fichier unique pour le client de chat.
- *src/bin/server* est l'exécutable du serveur, réparti sur quatre fichiers : *main.rs* contient la fonction, et il y a trois sous-modules, *connection.rs*, *group.rs* et *group\_table.rs*. *main*

Nous présenterons le contenu de chaque fichier source au cours du chapitre, mais une fois qu'ils sont tous en place, si vous tapez dans cette arborescence, cela compile la caisse de bibliothèque, puis construit les deux exécutables. Cargo inclut automatiquement la caisse de la bibliothèque en tant que dépendance, ce qui en fait un endroit pratique pour mettre des définitions partagées par le client et le serveur. De même, vérifie l'ensemble de l'arborescence source. Pour exécuter l'un des exécutables, vous pouvez utiliser des commandes telles que celles-ci : cargo  
build cargo check

```
$ cargo run --release --bin server -- localhost:8088
$ cargo run --release --bin client -- localhost:8088
```

L'option indique l'exécutable à exécuter et tous les arguments suivant l'option sont transmis à l'exécutable lui-même. Notre client et notre serveur veulent juste connaître l'adresse du serveur et le port TCP. --  
bin --

# Types d'erreurs et de résultats

Le module de la caisse de bibliothèque définit les types de résultats et d'erreurs que nous utiliserons dans toute l'application. *Depuis src/utils.rs*

```
:utils

use std::error::Error;

pub type ChatError = Box<dyn Error + Send + Sync + 'static>;
pub type ChatResult<T> = Result<T, ChatError>;
```

Ce sont les types d'erreurs à usage général que nous avons suggérés dans [« Utilisation de plusieurs types d'erreurs »](#). Les , et les caisses définissent chacune leurs propres types d'erreur, mais l'opérateur peut automatiquement les convertir tous en un , en utilisant l'implémentation de la bibliothèque standard du trait qui peut convertir tout type d'erreur approprié en . Les limites et garantissent que si une tâche générée sur un autre thread échoue, elle peut signaler l'erreur en toute sécurité au thread principal.

```
async_std serde_json tokio ? ChatError From Box<dyn Error + Send + Sync + 'static> Send Sync
```

Dans une application réelle, envisagez d'utiliser la caisse, qui fournit et des types similaires à ceux-ci. La caisse est facile à utiliser et offre de belles fonctionnalités au-delà de ce que nous pouvons offrir.

```
anyhow Error Result anyhow ChatError ChatResult
```

## Le Protocole

La caisse de bibliothèque capture l'ensemble de notre protocole de chat dans ces deux types, définis dans *lib.rs*:

```
use serde::{Deserialize, Serialize};
use std::sync::Arc;

pub mod utils;

#[derive(Debug, Deserialize, Serialize, PartialEq)]
pub enum FromClient {
 Join { group_name: Arc<String> },
 Post {
 group_name: Arc<String>,
 message: Arc<String>,
 },
}
```

```

#[derive(Debug, Deserialize, Serialize, PartialEq)]
pub enum FromServer {
 Message {
 group_name: Arc<String>,
 message: Arc<String>,
 },
 Error(String),
}

#[test]
fn test_fromclient_json() {
 use std::sync::Arc;

 let from_client = FromClient::Post {
 group_name: Arc::new("Dogs".to_string()),
 message: Arc::new("Samoyeds rock!".to_string()),
 };

 let json = serde_json::to_string(&from_client).unwrap();
 assert_eq!(json,
 r#"{"Post":{"group_name":"Dogs","message":"Samoyeds rock!"}}"#);

 assert_eq!(serde_json::from_str(&json).unwrap(),
 from_client);
}

```

L'énumération représente les paquets qu'un client peut envoyer au serveur : il peut demander à rejoindre un groupe et publier des messages dans n'importe quel groupe qu'il a rejoint. représente ce que le serveur peut renvoyer : les messages publiés dans un groupe et les messages d'erreur. L'utilisation d'une référence comptée au lieu d'une simple aide le serveur à éviter de faire des copies de chaînes lorsqu'il gère des groupes et distribue des

messages. FromClient FromServer Arc<String> String

Les attributs indiquent à la caisse de générer des implémentations de ses et traits pour et . Cela nous permet d'appeler pour les convertir en valeurs JSON, de les envoyer sur le réseau et enfin d'appeler pour les reconvertir dans leurs formulaires Rust. #

```
[derive] serde Serialize Deserialize FromClient FromServer
serde_json::to_string serde_json::from_str
```

Le test unitaire illustre comment cela est utilisé. Compte tenu de l'implémentation dérivée par , nous pouvons appeler pour transformer la valeur donnée en ce

```

JSON::test_fromclient_json Serialize serde serde_json::to_s
tring FromClient

{"Post": {"group_name": "Dogs", "message": "Samoyeds rock!"}}

```

Ensuite, l'implémentation dérivée analyse cela en une valeur équivalente. Notez que les pointeurs dans n'ont aucun effet sur le formulaire sérialisé : les chaînes comptées par référence apparaissent directement en tant que valeurs de membre d'objet

```
JSON.Deserialize FromClient Arc FromClient
```

## Prise en compte des entrées utilisateur : flux asynchrones

La première responsabilité de notre client de chat est de lire les commandes de l'utilisateur et d'envoyer les paquets correspondants au serveur.

La gestion d'une interface utilisateur appropriée dépasse le cadre de ce chapitre, nous allons donc faire la chose la plus simple possible qui fonctionne: lire des lignes directement à partir d'une entrée standard. Le code suivant va dans *src/bin/client.rs* :

```

use async_std::prelude::*;
use async_chat::utils::{self, ChatResult};
use async_std::io;
use async_std::net;

async fn send_commands(mut to_server: net::TcpStream) -> ChatResult<()>
 println!("Commands:\n\
 join GROUP\n\
 post GROUP MESSAGE...\n\
 Type Control-D (on Unix) or Control-Z (on Windows) \
 to close the connection.");
}

let mut command_lines = io::BufReader::new(io::stdin()).lines();
while let Some(command_result) = command_lines.next().await {
 let command = command_result?;
 // See the GitHub repo for the definition of `parse_command`.
 let request = match parse_command(&command) {
 Some(request) => request,
 None => continue,
 };

 utils::send_as_json(&mut to_server, &request).await?;
 to_server.flush().await?;
}

```

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

Cela appelle pour obtenir un handle asynchrone sur l'entrée standard du client, l'enveloppe dans un pour le mettre en mémoire tampon, puis appelle pour traiter l'entrée de l'utilisateur ligne par ligne. Il essaie d'analyser chaque ligne comme une commande correspondant à une valeur et, s'il réussit, envoie cette valeur au serveur. Si l'utilisateur entre une commande non reconnue, imprime un message d'erreur et renvoie , afin de pouvoir à nouveau faire le tour de la boucle. Si l'utilisateur tape une indication de fin de fichier, le flux renvoie et renvoie. Cela ressemble beaucoup au code que vous écririez dans un programme synchrone ordinaire, sauf qu'il utilise les versions de la

```
bibliothèque. async_std::io::stdin async_std::io::BufReader lines FromClient parse_command None send_commands lines None send_commands async_std
```

La méthode asynchrone est intéressante. Il ne peut pas renvoyer un itérateur, comme le fait la bibliothèque standard : la méthode est une fonction synchrone ordinaire, donc l'appel bloquerait le thread jusqu'à ce que la ligne suivante soit prête. Au lieu de cela, renvoie un *flux* de valeurs. Un flux est l'analogue asynchrone d'un itérateur : il produit une séquence de valeurs à la demande, de manière asynchrone. Voici la définition du trait, à partir du

```
module: BufReader lines Iterator::next command_lines.next() lines Result<String> Stream async_std::stream
```

```
trait Stream {
 type Item;

 // For now, read `Pin<&mut Self>` as `&mut Self`.
 fn poll_next(self: Pin<&mut Self>, cx: &mut Context<'_>)
 -> Poll<Option<Self::Item>>;
}
```

Vous pouvez considérer cela comme un hybride de la et des traits. Comme un itérateur, a a un type associé et utilise pour indiquer quand la séquence est terminée. Mais comme un futur, un flux doit être interrogé : pour obtenir l'élément suivant (ou apprendre que le flux est terminé), vous devez appeler jusqu'à ce qu'il revienne. L'implémentation d'un flux doit toujours revenir rapidement, sans blocage. Et si un flux revient, il doit avertir l'appelant lorsqu'il vaut la peine d'interroger à nouveau via le

fichier

```
. Iterator Future Stream Item Option poll_next Poll::Ready poll_next Poll::Pending Context
```

La méthode est difficile à utiliser directement, mais vous n'aurez généralement pas besoin de le faire. Comme les itérateurs, les flux ont une large collection de méthodes utilitaires telles que et . Parmi ceux-ci se trouve une méthode, qui renvoie un avenir du prochain flux. Plutôt que d'interroger explicitement le flux, vous pouvez appeler et attendre l'avenir qu'il renvoie à la

```
place.poll_next filter map next Option<Self::Item> next
```

En assemblant ces éléments, on consomme le flux de lignes d'entrée en bouclant les valeurs produites par un flux en utilisant avec : send\_commands next while let

```
while let Some(item) = stream.next().await {
 ... use item ...
}
```

(Les futures versions de Rust introduiront probablement une variante asynchrone de la syntaxe de boucle pour consommer des flux, tout comme une boucle ordinaire consomme des valeurs.) for for Iterator

Interroger un flux après sa fin, c'est-à-dire après son retour pour indiquer la fin du flux, revient à faire appel à un itérateur après son retour ou à interroger un futur après son retour : le trait ne spécifie pas ce que le flux doit faire, et certains flux peuvent mal se comporter. Comme les futurs et les itérateurs, les flux ont une méthode pour s'assurer que ces appels se comportent de manière prévisible, lorsque cela est nécessaire; voir la documentation pour plus de détails. Poll::Ready(None) next None Poll::Ready Stream fuse

Lorsque vous travaillez avec des flux, n'oubliez pas d'utiliser le prélude : async\_std

```
use async_std::prelude::*;


```

C'est parce que les méthodes d'utilité pour le trait, comme , , , et ainsi de suite, ne sont en fait pas définies sur elles-mêmes. Au lieu de cela, ce sont des méthodes par défaut d'un trait séparé, , qui est automatiquement implémenté pour tous les

```
S: Stream next map filter Stream StreamExt Stream
```

```

pub trait StreamExt: Stream {
 ... define utility methods as default methods ...
}

impl<T: Stream> StreamExt for T { }

```

Ceci est un exemple du modèle de *trait d'extension* que nous avons décrit dans « [Traits et types d'autres personnes](#) ». Le module met les méthodes en champ d'application, de sorte que l'utilisation du prélude garantit que ses méthodes sont visibles dans votre code.

```
async_std::prelude StreamExt
```

## Envoi de paquets

Pour transmettre des paquets sur une prise réseau, notre client et notre serveur utilisent la fonction du module de notre caisse de bibliothèque

```
: send_as_json utils
```

```

use async_std::prelude::*;
use serde::Serialize;
use std::marker::Unpin;

pub async fn send_as_json<S, P>(outbound: &mut S, packet: &P) -> ChatRes
where
 S: async_std::io::Write + Unpin,
 P: Serialize,
{
 let mut json = serde_json::to_string(&packet)?;
 json.push('\n');
 outbound.write_all(json.as_bytes()).await?;
 Ok(())
}

```

Cette fonction génère la représentation JSON de comme un , ajoute une nouvelle ligne à la fin, puis écrit le tout dans . packet String outbound

D'après son article, vous pouvez voir que c'est assez souple. Le type de paquet à envoyer, , peut être tout ce qui implémente . Le flux de sortie peut être tout ce qui implémente , la version asynchrone du trait pour les flux de sortie. Ceci est suffisant pour que nous envoyions et des valeurs sur un fichier asynchrone . Garder la définition de générique garantit qu'elle ne dépend pas des détails du flux ou des types de paquets de manière surprenante: ne peut utiliser que des méthodes à partir de ces traits.

```
where send_as_json P serde::Serialize S async_std::io:
```

```
:Write std::io::Write FromClient FromServer TcpStream send_
as_json send_as_json
```

La contrainte sur est requise pour utiliser la méthode. Nous aborderons l'épinglage et le dépouillement plus loin dans ce chapitre, mais pour le moment, il devrait suffire d'ajouter des contraintes pour taper des variables si nécessaire; le compilateur Rust signalera ces cas si vous oubliez.

```
Unpin S write_all Unpin
```

Plutôt que de sérialiser le paquet directement dans le flux, sérialisez-le dans un temporaire, puis l'écrit dans . La caisse fournit des fonctions permettant de sérialiser les valeurs directement dans les flux de sortie, mais ces fonctions ne prennent en charge que les flux synchrones. L'écriture dans des flux asynchrones nécessiterait des modifications fondamentales à la fois et au cœur indépendant du format de la caisse, car les caractéristiques autour desquelles elles sont conçues ont des méthodes synchrones.

```
outbound send_as_json String outbound serde_json s
erde_json serde
```

Comme pour les flux, de nombreuses méthodes de traits d'E/S sont en fait définies sur des traits d'extension, il est donc important de s'en souvenir chaque fois que vous les utilisez.

```
async_std use
async_std::prelude::*
```

## Réception de paquets : plus de flux asynchrones

Pour la réception de paquets, notre serveur et client utiliseront cette fonction du module pour recevoir et des valeurs d'un socket TCP asynchrone tamponné, un

```
:utils FromClient FromServer async_std::io::BufReader<TcpSt
ream>
```

```
use serde::de::DeserializeOwned;

pub fn receive_as_json<S, P>(inbound: S) -> impl Stream<Item = ChatResul
where S: async_std::io::BufRead + Unpin,
 P: DeserializeOwned,
{
 inbound.lines()
 .map(|line_result| -> ChatResult<P> {
 let line = line_result?;
 let parsed = serde_json::from_str::<P>(&line)?;
 Ok(parsed)
 })
}
```

Comme , cette fonction est générique dans les types de flux d'entrée et de paquets : `send_as_json`

- Le type de flux doit implémenter , l'analogue asynchrone de , représentant un flux d'octets d'entrée mis en mémoire tampon. `s` `async_std::io::BufRead` `std::io::BufRead`
- Le type de paquet doit implémenter , une variante plus stricte du trait de caractère de . Pour plus d'efficacité, peut produire et valoriser qui empruntent leur contenu directement à partir du tampon à partir duquel ils ont été déserialisés, afin d'éviter de copier des données. Dans notre cas, cependant, ce n'est pas bon: nous devons renvoyer les valeurs déserialisées à notre appelant, de sorte qu'ils doivent pouvoir survivre aux tampons à partir desquels nous les avons analysés. Un type qui implémente est toujours indépendant du tampon à partir duquel il a été

```
déserialisé. P DeserializeOwned serde Deserialize DeserializeOwned
e &str &[u8] DeserializeOwned
```

L'appel nous donne un de valeurs. Nous utilisons ensuite l'adaptateur du flux pour appliquer une fermeture à chaque élément, en gérant les erreurs et en analysant chaque ligne comme la forme JSON d'une valeur de type . Cela nous donne un flux de valeurs, que nous retournons directement. Le type de retour de la fonction est le suivant

```
: inbound.lines() Stream std::io::Result<String> map P ChatR
esult<P>
```

```
impl Stream<Item = ChatResult<P>>
```

Cela indique que nous *retournons un* type qui produit une séquence de valeurs de manière asynchrone, mais notre appelant ne peut pas dire exactement de quel type il s'agit. Étant donné que la fermeture à laquelle nous passons a de toute façon un type anonyme, c'est le type le plus spécifique qui pourrait éventuellement revenir. `ChatResult<P>` `map receive_as_json`

Notez qu'il ne s'agit pas, en soi, d'une fonction asynchrone. C'est une fonction ordinaire qui renvoie une valeur asynchrone, un flux. Comprendre la mécanique du support asynchrone de Rust plus profondément que « simplement ajouter et partout » ouvre la possibilité de définitions claires, flexibles et efficaces comme celle-ci qui tirent pleinement parti du langage. `receive_as_json` `async .await`

Pour voir comment il est utilisé, voici la fonction de notre client de chat de *src/bin/client.rs*, qui reçoit un flux de valeurs du réseau et les imprime pour que l'utilisateur puisse les voir:

```
receive_as_json handle_replies FromServer
```

```
use async_chat::FromServer;

async fn handle_replies(from_server: net::TcpStream) -> ChatResult<()> {
 let buffered = io::BufReader::new(from_server);
 let mut reply_stream = utils::receive_as_json(buffered);

 while let Some(reply) = reply_stream.next().await {
 match reply? {
 FromServer::Message { group_name, message } => {
 println!("message posted to {}: {}", group_name, message);
 }
 FromServer::Error(message) => {
 println!("error from server: {}", message);
 }
 }
 }

 Ok(())
}
```

Cette fonction prend un socket recevant des données du serveur, en enroule un autour de celui-ci (notez bien, la version), puis les transmet pour obtenir un flux de valeurs entrantes. Ensuite, il utilise une boucle pour gérer les réponses entrantes, vérifier les résultats des erreurs et imprimer chaque réponse du serveur pour que l'utilisateur puisse la

```
voir. BufReader async_std receive_as_json FromServer while
let
```

## Fonction principale du client

Puisque nous avons présenté les deux et , nous pouvons montrer la fonction principale du client de chat, à partir de *src/bin/client.rs*:

```
send_commands handle_replies
```

```
use async_std::task;

fn main() -> ChatResult<()> {
 let address = std::env::args().nth(1)
 .expect("Usage: client ADDRESS:PORT");
```

```

task::block_on(async {
 let socket = net::TcpStream::connect(address).await?;
 socket.set_nodelay(true)?;

 let to_server = send_commands(socket.clone());
 let from_server = handle_replies(socket);

 from_server.race(to_server).await?;

 Ok(())
})
}

```

Après avoir obtenu l'adresse du serveur à partir de la ligne de commande, dispose d'une série de fonctions asynchrones qu'il aimerait appeler, de sorte qu'il enveloppe le reste de la fonction dans un bloc asynchrone et passe le futur du bloc à exécuter.

```
main async_std::task::block_on
```

Une fois la connexion établie, nous voulons que les fonctions et s'exécutent en tandem, afin que nous puissions voir les messages des autres arriver pendant que nous tapons. Si nous entrons dans l'indicateur de fin de fichier ou si la connexion au serveur tombe, le programme devrait se fermer.

```
send_commands handle_replies
```

Compte tenu de ce que nous avons fait ailleurs dans le chapitre, vous pouvez vous attendre à un code comme celui-ci :

```

let to_server = task::spawn(send_commands(socket.clone()));
let from_server = task::spawn(handle_replies(socket));

to_server.await?;
from_server.await?;

```

Mais comme nous attendons les deux poignées de jointure, cela nous donne un programme qui se ferme une fois *les deux* tâches terminées. Nous voulons sortir dès que *l'un ou l'autre* aura terminé. La méthode sur les futurs accomplit cela. L'appel renvoie un nouvel avenir qui sonde à la fois et revient dès que l'un d'eux est prêt. Les deux contrats à terme doivent avoir le même type de sortie : la valeur finale est celle du futur terminé en premier. L'avenir inachevé est abandonné.

```
race from_server.race(to_server) from_server to_server Poll::Ready(v)
```

La méthode, ainsi que de nombreux autres utilitaires pratiques, est définie sur le trait, ce qui nous rend visible. `race` `async_std::prelude::FutureExt` `async_std::prelude`

À ce stade, la seule partie du code du client que nous n'avons pas montrée est la fonction. C'est un code de gestion de texte assez simple, nous ne montrerons donc pas sa définition ici. Consultez le code complet dans le référentiel Git pour plus de détails. `parse_command`

## Fonction principale du serveur

Voici tout le contenu du fichier principal pour le serveur, `src/bin/server/main.rs`:

```
use async_std::prelude::*;
use async_chat::utils::ChatResult;
use std::sync::Arc;

mod connection;
mod group;
mod group_table;

use connection::serve;

fn main() -> ChatResult<()> {
 let address = std::env::args().nth(1).expect("Usage: server ADDRESS");

 let chat_group_table = Arc::new(group_table::GroupTable::new());

 async_std::task::block_on(async {
 // This code was shown in the chapter introduction.
 use async_std::{net, task};

 let listener = net::TcpListener::bind(address).await?;

 let mut new_connections = listener.incoming();
 while let Some(socket_result) = new_connections.next().await {
 let socket = socket_result?;
 let groups = chat_group_table.clone();
 task::spawn(async {
 log_error(serve(socket, groups).await);
 });
 }
 })
}
```

```

 }

 fn log_error(result: ChatResult<()>) {
 if let Err(error) = result {
 eprintln!("Error: {}", error);
 }
 }
}

```

La fonction du serveur ressemble à celle du client: il fait un peu de configuration, puis appelle pour exécuter un bloc asynchrone qui fait le vrai travail. Pour gérer les connexions entrantes des clients, il crée un socket, dont la méthode renvoie un flux de

```
valeurs.main block_on TcpListener incoming std::io::Result<T
cpStream>
```

Pour chaque connexion entrante, nous générons une tâche asynchrone exécutant la fonction. Chaque tâche reçoit également une référence à une valeur représentant la liste actuelle des groupes de discussion de notre serveur, partagée par toutes les connexions via un pointeur compté par référence. connection::serve GroupTable Arc

Si une erreur est renvoyée, nous enregistrons un message dans la sortie d'erreur standard et laissons la tâche se terminer. Les autres connexions continuent de fonctionner comme d'habitude. connection::serve

## Gestion des connexions de chat : Mutex asynchrones

Voici le cheval de bataille du serveur: la fonction du module dans `src/bin/server/connection.rs`: serve connection

```

use async_chat::{FromClient, FromServer};
use async_chat::utils::{self, ChatResult};
use async_std::prelude::*;
use async_std::io::BufReader;
use async_std::net::TcpStream;
use async_std::sync::Arc;

use crate::group_table::GroupTable;

pub async fn serve(socket: TcpStream, groups: Arc<GroupTable>)
 -> ChatResult<()>
{
 let outbound = Arc::new(Outbound::new(socket.clone()));
}

```

```

let buffered = BufReader::new(socket);
let mut from_client = utils::receive_as_json(buffered);
while let Some(request_result) = from_client.next().await {
 let request = request_result?;

 let result = match request {
 FromClient::Join { group_name } => {
 let group = groups.get_or_create(group_name);
 group.join(outbound.clone());
 Ok(())
 }

 FromClient::Post { group_name, message } => {
 match groups.get(&group_name) {
 Some(group) => {
 group.post(message);
 Ok(())
 }
 None => {
 Err(format!("Group '{}' does not exist", group_name))
 }
 }
 }
 };

 if let Err(message) = result {
 let report = FromServer::Error(message);
 outbound.send(report).await?;
 }
}

Ok(())
}

```

Il s'agit presque d'une image miroir de la fonction du client : la majeure partie du code est une boucle gérant un flux entrant de valeurs, construit à partir d'un flux TCP tamponné avec . Si une erreur se produit, nous générerons un paquet pour transmettre la mauvaise nouvelle au client. handle\_replies FromClient receive\_as\_json FromServer: :Error

En plus des messages d'erreur, les clients souhaitent également recevoir des messages des groupes de discussion qu'ils ont rejoins, de sorte que la connexion au client doit être partagée avec chaque groupe. Nous pourrions simplement donner à tout le monde un clone de , mais si deux de

ces sources essaient d'écrire un paquet sur le socket en même temps, leur sortie pourrait être entrelacée et le client finirait par recevoir du JSON brouillé. Nous devons organiser un accès simultané sécurisé à la connexion. `TcpStream`

Ceci est géré avec le type, défini dans `src/bin/server/connection.rs` comme suit : `Outbound`

```
use async_std::sync::Mutex;

pub struct Outbound(Mutex<TcpStream>);

impl Outbound {
 pub fn new(to_client: TcpStream) -> Outbound {
 Outbound(Mutex::new(to_client))
 }

 pub async fn send(&self, packet: FromServer) -> ChatResult<()> {
 let mut guard = self.0.lock().await;
 utils::send_as_json(&mut *guard, &packet).await?;
 guard.flush().await?;
 Ok(())
 }
}
```

Lorsqu'elle est créée, une valeur prend possession d'un et l'enveloppe dans un pour s'assurer qu'une seule tâche peut l'utiliser à la fois. La fonction encapsule chacun dans un pointeur compté par référence afin que tous les groupes joints par le client puissent pointer vers la même instance

partagée. `Outbound` `TcpStream` `Mutex` serve `Outbound` `Arc` `Outbound`

Un appel à verrouiller d'abord le mutex, renvoyant une valeur de garde qui déréférence à l'intérieur. Nous avons l'habitude de transmettre , puis finalement nous appelons pour nous assurer qu'il ne languira pas à moitié transmis dans un tampon quelque part. (À notre connaissance, ne tamponne pas réellement les données, mais le trait permet à ses implementations de le faire, nous ne devrions donc prendre aucun risque.) `Outbound::send` `TcpStream` `send_as_json` `packet` `guard.flush()` `TcpStream` `Write`

L'expression nous permet de contourner le fait que Rust n'applique pas de coercitions deref pour répondre aux limites des traits. Au lieu de cela, nous déréférençons explicitement la garde mutex, puis empruntons une

référence mutable à la protection qu'elle protège, produisant ce qui nécessite. `&mut *guard TcpStream &mut TcpStream send_as_json`

Notez que cela utilise le type, pas le fichier . Il y a trois raisons à cela. `Outbound async_std::sync::Mutex Mutex`

Tout d'abord, la bibliothèque standard peut mal se comporter si une tâche est suspendue tout en tenant une garde mutex. Si le thread qui avait exécuté cette tâche récupère une autre tâche qui tente de verrouiller la même chose, des problèmes s'ensuivent: du point de vue du point de vue, le thread qui le possède déjà essaie de le verrouiller à nouveau. La norme n'est pas conçue pour gérer ce cas, il panique donc ou bloque. (Il n'accordera jamais le verrou de manière inappropriée.) Des travaux sont en cours pour que Rust détecte ce problème au moment de la compilation et émette un avertissement chaque fois qu'un garde est en direct sur une expression. Puisqu'il doit tenir la serrure pendant qu'elle attend les futurs de et , il doit utiliser 's

```
.Mutex Mutex Mutex Mutex std::sync::Mutex await Outbound:::s
end send_as_json guard.flush async_std Mutex
```

Deuxièmement, la méthode asynchrone renvoie un futur d'un garde, de sorte qu'une tâche en attente de verrouillage d'un mutex cède son fil pour d'autres tâches à utiliser jusqu'à ce que le mutex soit prêt. (Si le mutex est déjà disponible, l'avenir est prêt immédiatement et la tâche ne se suspend pas du tout.) La méthode standard, d'autre part, épingle tout le fil en attendant d'acquérir la serrure. Étant donné que le code précédent contient le mutex pendant qu'il transmet un paquet sur le réseau, cela peut prendre un certain temps. `Mutex lock lock Mutex lock`

Enfin, la norme ne doit être déverrouillée que par le même thread qui l'a verrouillée. Pour faire respecter cela, le type de garde du mutex standard n'implémente pas : il ne peut pas être transmis à d'autres threads. Cela signifie qu'un futur tenant une telle garde ne s'implémente pas lui-même , et ne peut pas être passé à s'exécuter sur un pool de threads; il ne peut être exécuté qu'avec ou . La garde pour un implémente donc il n'y a aucun problème à l'utiliser dans les tâches

```
engendrées. Mutex Send Send spawn block_on spawn_local async_
std Mutex Send
```

## La table de groupe : Mutex synchrones

Mais la morale de l'histoire n'est pas aussi simple que « Toujours utiliser dans du code asynchrone ». Souvent, il n'est pas nécessaire d'attendre

quoi que ce soit tout en tenant un mutex, et la serrure n'est pas maintenue longtemps. Dans de tels cas, la bibliothèque standard peut être beaucoup plus efficace. Le type de notre serveur de chat illustre ce cas.

Voici le contenu complet de

```
src/bin/server/group_table.rs: async_std::sync::Mutex Mutex GroupTable
```

```
use crate::group::Group;
use std::collections::HashMap;
use std::sync::{Arc, Mutex};

pub struct GroupTable(Mutex<HashMap<Arc<String>, Arc<Group>>>);

impl GroupTable {
 pub fn new() -> GroupTable {
 GroupTable(Mutex::new(HashMap::new()))
 }

 pub fn get(&self, name: &String) -> Option<Arc<Group>> {
 self.0.lock()
 .unwrap()
 .get(name)
 .cloned()
 }

 pub fn get_or_create(&self, name: Arc<String>) -> Arc<Group> {
 self.0.lock()
 .unwrap()
 .entry(name.clone())
 .or_insert_with(|| Arc::new(Group::new(name)))
 .clone()
 }
}
```

A est simplement une table de hachage protégée par mutex, mappant les noms des groupes de discussion aux groupes réels, tous deux gérés à l'aide de pointeurs comptés par référence. Les méthodes et verrouillent le mutex, effectuent quelques opérations de table de hachage, peut-être quelques allocations, et retournent. GroupTable get get\_or\_create

Dans , nous utilisons un vieux simple . Il n'y a pas de code asynchrone dans ce module, il n'y a donc pas de s à éviter. En effet, si nous voulions utiliser ici, nous aurions besoin de faire et dans des fonctions asynchrones, ce qui introduit la surcharge de la création future, des suspensions et des reprises pour peu d'avantages: le mutex n'est verrouillé que

pour certaines opérations de hachage et peut-être quelques allocations. `GroupTable std::sync::Mutex await async_std::sync ::Mutex get get_or_create`

Si notre serveur de chat se retrouvait avec des millions d'utilisateurs et que le mutex devenait un goulot d'étranglement, le rendre asynchrone ne résoudrait pas ce problème. Il serait probablement préférable d'utiliser une sorte de type de collection spécialisé pour l'accès simultané au lieu de . Par exemple, la caisse fournit un tel type. `GroupTable HashMap dashmap`

## Groupes de discussion : les chaînes de diffusion de tokio

Dans notre serveur, le type représente un groupe de discussion. Ce type doit uniquement prendre en charge les deux méthodes qui appellent :, pour ajouter un nouveau membre et , pour publier un message. Chaque message posté doit être distribué à tous les membres. `group::Group connection::serve join post`

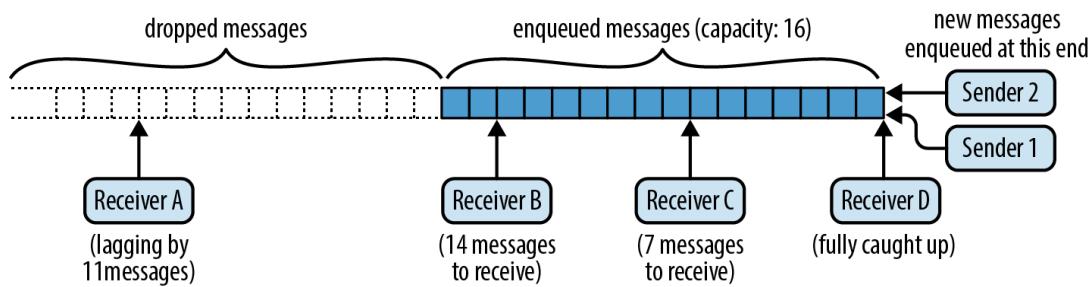
C'est là que nous abordons le défi mentionné plus tôt de la *contre-pression*. Il y a plusieurs besoins en tension les uns avec les autres:

- Si un membre ne peut pas suivre les messages publiés dans le groupe (s'il a une connexion réseau lente, par exemple), les autres membres du groupe ne doivent pas être affectés.
- Même si un membre prend du retard, il devrait y avoir des moyens pour lui de se joindre à la conversation et de continuer à participer d'une manière ou d'une autre.
- La mémoire mise en mémoire tampon des messages ne doit pas croître sans limite.

Étant donné que ces défis sont courants lors de la mise en œuvre de modèles de communication plusieurs-à-plusieurs, la caisse fournit un type de *canal de diffusion* qui implémente un ensemble raisonnable de compromis. Un canal de diffusion est une file d'attente de valeurs (dans notre cas, des messages de chat) qui permet à un nombre illimité de fils de discussion ou de tâches différents d'envoyer et de recevoir des valeurs. C'est ce qu'on appelle un canal de « diffusion » car chaque consommateur obtient sa propre copie de chaque valeur envoyée. (Le type de valeur doit être implémenté.) `tokio tokio Clone`

Normalement, un canal de diffusion conserve un message dans la file d'attente jusqu'à ce que chaque consommateur ait reçu sa copie. Mais si la longueur de la file d'attente dépasse la capacité maximale du canal, spécifiée lors de sa création, les messages les plus anciens sont supprimés. Tous les consommateurs qui n'ont pas pu suivre reçoivent une erreur la prochaine fois qu'ils essaient d'obtenir leur prochain message, et le canal les rattrape au message le plus ancien encore disponible.

Par exemple, [la figure 20-4](#) montre un canal de diffusion d'une capacité maximale de 16 valeurs.



Graphique 20-4. Une chaîne de diffusion tokio

Il y a deux expéditeurs qui mettent les messages en file d'attente et quatre destinataires qui les mettent en file d'attente, ou plus précisément, qui copient les messages hors de la file d'attente. Le récepteur B a encore 14 messages à recevoir, le récepteur C en a 7 et le récepteur D est complètement rattrapé. Le récepteur A a pris du retard et 11 messages ont été supprimés avant qu'il ne puisse les voir. Sa prochaine tentative de réception d'un message échouera, renvoyant une erreur indiquant la situation, et il sera rattrapé à la fin actuelle de la file d'attente.

Notre serveur de chat représente chaque groupe de chat comme un canal de diffusion porteur de valeurs : la publication d'un message au groupe le diffuse à tous les membres actuels. Voici la définition du type, définie dans `src/bin/server/group.rs` :

```
use async_std::task;
use crate::connection::Outbound;
use std::sync::Arc;
use tokio::sync::broadcast;
```

```
pub struct Group {
 name: Arc<String>,
 sender: broadcast::Sender<Arc<String>>
}

impl Group {
 pub fn new(name: Arc<String>) -> Group {
 let (tx, rx) = broadcast::channel(16);
 let name = name.clone();
 task::spawn(async move {
 while let Some(message) = rx.recv().await {
 tx.send(Arc::new(message)).unwrap();
 }
 });
 Group { name, sender: tx }
 }
}
```

```

 let (sender, _receiver) = broadcast::channel(1000);
 Group { name, sender }
}

pub fn join(&self, outbound: Arc<Outbound>) {
 let receiver = self.sender.subscribe();

 task::spawn(handle_subscriber(self.name.clone(),
 receiver,
 outbound));
}

pub fn post(&self, message: Arc<String>) {
 // This only returns an error when there are no subscribers. A
 // connection's outgoing side can exit, dropping its subscriptio
 // slightly before its incoming side, which may end up trying to
 // message to an empty group.
 let _ignored = self.sender.send(message);
}
}

```

Une structure contient le nom du groupe de discussion, ainsi qu'une `diffusion::Expéditeur` représentant l'extrémité d'envoi du canal de diffusion du groupe. La fonction appelle à créer un canal de diffusion d'une capacité maximale de 1 000 messages. La fonction renvoie à la fois un expéditeur et un récepteur, mais nous n'avons pas besoin du récepteur à ce stade, car le groupe n'a pas encore de

membres. `Group Group::new broadcast::channel channel`

Pour ajouter un nouveau membre au groupe, la méthode appelle la méthode de l'expéditeur pour créer un nouveau récepteur pour le canal. Ensuite, il génère une nouvelle tâche asynchrone pour surveiller ce récepteur à la recherche de messages et les réécrire dans le client, dans la fonction. `Group::join subscribe handle_subscribe`

Avec ces détails en main, la méthode est simple: elle envoie simplement le message à la chaîne de diffusion. Étant donné que les valeurs véhiculées par le canal sont des valeurs, donner à chaque destinataire sa propre copie d'un message ne fait qu'augmenter le nombre de références du message, sans aucune copie ni allocation de tas. Une fois que tous les abonnés ont transmis le message, le nombre de références tombe à zéro et le message est libéré. `Group::post Arc<String>`

Voici la définition de : `handle_subscriber`

```

use async_chat::FromServer;
use tokio::sync::broadcast::error::RecvError;

async fn handle_subscriber(group_name: Arc<String>,
 mut receiver: broadcast::Receiver<Arc<String>>,
 outbound: Arc<Outbound>)
{
 loop {
 let packet = match receiver.recv().await {
 Ok(message) => FromServer::Message {
 group_name: group_name.clone(),
 message: message.clone(),
 },
 Err(RecvError::Lagged(n)) => FromServer::Error(
 format!("Dropped {} messages from {}", n, group_name)
),
 Err(RecvError::Closed) => break,
 };

 if outbound.send(packet).await.is_err() {
 break;
 }
 }
}

```

Bien que les détails soient différents, la forme de cette fonction est familière : il s'agit d'une boucle qui reçoit des messages du canal de diffusion et les transmet au client via la valeur partagée. Si la boucle ne peut pas suivre le canal de diffusion, elle reçoit une erreur, qu'elle signale consciencieusement au client. `Outbound Lagged`

Si l'envoi d'un paquet au client échoue complètement, peut-être parce que la connexion est fermée, quitte sa boucle et revient, ce qui entraîne la fermeture de la tâche asynchrone. Cela supprime le canal de diffusion, en le désabonnant de la chaîne. De cette façon, lorsqu'une connexion est interrompue, chacune de ses appartennances à un groupe est nettoyée la prochaine fois que le groupe tente de lui envoyer un message. `handle_subscriber Receiver`

Nos groupes de discussion ne ferment jamais, car nous ne supprimons jamais un groupe de la table de groupe, mais juste pour être complet, est prêt à gérer une erreur en quittant la tâche. `handle_subscriber Closed`

Notez que nous créons une nouvelle tâche asynchrone pour chaque appartenance au groupe de chaque client. Cela est possible parce que les tâches asynchrones utilisent beaucoup moins de mémoire que les threads et parce que le passage d'une tâche asynchrone à une autre dans un processus est assez efficace.

Il s'agit donc du code complet du serveur de chat. C'est un peu spartiate, et il y a beaucoup plus de fonctionnalités précieuses dans le , , et les caisses que nous pouvons couvrir dans ce livre, mais idéalement cet exemple étendu parvient à illustrer comment certaines des caractéristiques de l'écosystème asynchrone fonctionnent ensemble: les tâches asynchrones, les flux, les traits d'E / S asynchrones, les canaux et les mutex des deux saveurs. `async_std tokio futures`

## Futurs primitifs et exécuteurs testamentaires: quand un avenir vaut-il la peine d'être sondé à nouveau?

Le serveur de chat montre comment nous pouvons écrire du code en utilisant des primitives asynchrones comme et le canal, et utiliser des exécuteurs comme et pour piloter leur exécution. Maintenant, nous pouvons jeter un coup d'œil à la façon dont ces choses sont mises en œuvre. La question clé est, lorsqu'un futur revient, comment se coordonne-t-il avec l'exécuteur testamentaire pour l'interroger à nouveau au bon moment?

```
TcpListener broadcast block_on spawn Poll::Pending
```

Pensez à ce qui se passe lorsque nous exécutons du code comme celui-ci, à partir de la fonction du client de chat: `main`

```
task::block_on(async {
 let socket = net::TcpStream::connect(address).await?;
 ...
})
```

La première fois qu'elle sonde l'avenir du bloc asynchrone, la connexion réseau n'est presque certainement pas prête immédiatement, alors elle se met en veille. Mais quand devrait-il se réveiller? D'une manière ou d'une autre, une fois que la connexion réseau est prête, doit dire qu'il doit essayer d'interroger à nouveau l'avenir du bloc asynchrone, car il sait que cette fois, la volonté se terminera et l'exécution du bloc asynchrone peut progresser. `block_on block_on TcpStream block_on await`

Lorsqu'un exécuteur testamentaire interroge un futur, il doit passer dans un rappel appelé *waker*. Si l'avenir n'est pas encore prêt, les règles du trait disent qu'il doit revenir pour l'instant, et faire en sorte que le réveil soit invoqué plus tard, si et quand l'avenir vaut la peine d'être sondé à nouveau.`block_on Future Poll::Pending`

Donc, une implémentation manuscrite de ressemble souvent à ceci:`Future`

```
use std::task::Waker;

struct MyPrimitiveFuture {
 ...
 waker: Option<Waker>,
}

impl Future for MyPrimitiveFuture {
 type Output = ...;

 fn poll(mut self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<...>
 {
 ...

 if ... future is ready ... {
 return Poll::Ready(final_value);
 }

 // Save the waker for later.
 self.waker = Some(cx.waker().clone());
 Poll::Pending
 }
}
```

En d'autres termes, si la valeur de l'avenir est prête, retournez-la. Sinon, cachez un clone du waker de 's quelque part, et revenez`.Context Poll::Pending`

Lorsque l'avenir vaut à nouveau la peine d'être interrogé, l'avenir doit en informer le dernier exécuteur testamentaire qui l'a interrogé en appelant la méthode de son waker :`wake`

```
// If we have a waker, invoke it, and clear `self.waker`.
if let Some(waker) = self.waker.take() {
 waker.wake();
}
```

Idéalement, l'exécuteur testamentaire et le futur se relaient pour sonder et se réveiller : l'exécuteur sonde l'avenir et s'endort, puis l'avenir invoque le réveil, de sorte que l'exécuteur se réveille et sonde à nouveau l'avenir.

Les futurs des fonctions et des blocs asynchrones ne traitent pas des réveils eux-mêmes. Ils transmettent simplement le contexte qui leur est donné aux sous-futurs qu'ils attendent, leur déléguant l'obligation de sauver et d'invoquer des réveils. Dans notre client de chat, le premier sondage de l'avenir du bloc asynchrone ne fait que transmettre le contexte lorsqu'il attend l'avenir de . Les sondages suivants transmettent également leur contexte à l'avenir que le bloc attend ensuite. `TcpStream::connect`

`TcpStream::connect` l'avenir gère l'interrogation comme le montre l'exemple précédent : il remet le réveil à un thread d'assistance qui attend que la connexion soit prête, puis appelle le réveil.

`waker` implémente et , afin qu'un futur puisse toujours faire sa propre copie du waker et l'envoyer à d'autres threads si nécessaire. La méthode consomme le waker. Il existe également une méthode qui ne le fait pas, mais certains exécuteurs peuvent implémenter la version consommatrice un peu plus efficacement. (La différence est tout au plus un )  
`Clone Send Waker::wake wake_by_ref clone`

Il est inoffensif pour un exécuteur testamentaire de surplomber un avenir, tout simplement inefficace. Les futurs, cependant, devraient faire attention à n'invoquer un réveil que lorsque l'interrogation ferait des progrès réels: un cycle de réveils et de sondages fallacieux peut empêcher un exécuteur de dormir du tout, gaspillant de l'énergie et laissant le processeur moins réactif à d'autres tâches.

Maintenant que nous avons montré comment les exécuteurs et les futurs primitifs communiquent, nous allons implémenter nous-mêmes un avenir primitif, puis parcourir une implémentation de l'exécuteur testamentaire. `block_on`

## Invoquer les wakers : `spawn_blocking`

Plus haut dans le chapitre, nous avons décrit la fonction, qui démarre une fermeture donnée en cours d'exécution sur un autre thread et renvoie un futur de sa valeur de retour. Nous avons maintenant toutes les pièces dont nous avons besoin pour nous mettre en œuvre. Pour plus de simplicité, notre version crée un nouveau thread pour chaque fermeture, plutôt

que d'utiliser un pool de threads, comme le fait la version de

```
. spawn_blocking spawn_blocking async_std
```

Bien que renvoie un futur, nous n'allons pas l'écrire comme un fichier . Il s'agira plutôt d'une fonction synchrone ordinaire qui renvoie une struct , sur laquelle nous nous implémenterons nous-mêmes. `spawn_blocking` `fn` `SpawnBlocking Future`

La signature de notre est la suivante: `spawn_blocking`

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
 F: Send + 'static,
 T: Send + 'static,
```

Étant donné que nous devons envoyer la fermeture à un autre thread et ramener la valeur de retour, la fermeture et sa valeur de retour doivent implémenter . Et comme nous n'avons aucune idée de la durée du thread, ils doivent tous les deux l'être également. Ce sont les mêmes limites qu'elle-même impose. `F` `T` `Send` `'static` `std::thread::spawn`

`SpawnBlocking<T>` est un avenir de la valeur de rendement de la fermeture. Voici sa définition :

```
use std::sync::{Arc, Mutex};
use std::task::Waker;

pub struct SpawnBlocking<T>(Arc<Mutex<Shared<T>>>);

struct Shared<T> {
 value: Option<T>,
 waker: Option<Waker>,
}
```

La structure doit servir de rendez-vous entre le futur et le fil qui exécute la fermeture, elle est donc détenue par un et protégé par un . (Un mutex synchrone est bien ici.) L'interrogation de l'avenir vérifie si elle est présente et enregistre le réveil dans le cas contraire. Le thread qui exécute la fermeture enregistre sa valeur de retour dans, puis appelle , le cas échéant. `Shared Arc Mutex value waker value waker`

Voici la définition complète de : `spawn_blocking`

```
pub fn spawn_blocking<T, F>(closure: F) -> SpawnBlocking<T>
where F: FnOnce() -> T,
```

```

F: Send + 'static,
T: Send + 'static,
{
 let inner = Arc::new(Mutex::new(Shared {
 value: None,
 waker: None,
 }));
}

std::thread::spawn({
 let inner = inner.clone();
 move || {
 let value = closure();

 let maybe_waker = {
 let mut guard = inner.lock().unwrap();
 guard.value = Some(value);
 guard.waker.take()
 };

 if let Some(waker) = maybe_waker {
 waker.wake();
 }
 }
});

SpawnerBlocking(inner)
}

```

Après avoir créé la valeur, cela génère un thread pour exécuter la fermeture, stocker le résultat dans le champ de ' et appeler le waker, le cas échéant. Shared Shared value

Nous pouvons mettre en œuvre pour ce qui suit: Future SpawnerBlocking

```

use std::future::Future;
use std::pin::Pin;
use std::task::{Context, Poll};

impl<T: Send> Future for SpawnerBlocking<T> {
 type Output = T;

 fn poll(self: Pin<&mut Self>, cx: &mut Context<'_>) -> Poll<T> {
 let mut guard = self.0.lock().unwrap();
 if let Some(value) = guard.value.take() {
 return Poll::Ready(value);
 }
 }
}

```

```

 guard.waker = Some(cx.waker().clone());
 Poll::Pending
}
}

```

L'interrogation d'un est vérifie si la valeur de la fermeture est prête, en prenant possession et en la renvoyant si c'est le cas. Sinon, l'avenir est toujours en attente, il enregistre donc un clone du waker du contexte dans le champ du futur. `SpawnBlocking` waker

Une fois que a retourné , vous n'êtes pas censé l'interroger à nouveau. Les façons habituelles de consommer les futurs, comme et , respectent toutes cette règle. Si un avenir est surexploité, rien de particulièrement terrible ne se produit, mais cela ne nécessite aucun effort pour gérer ce cas non plus. Ceci est typique pour les futurs

`manuscrits.Future Poll::Ready await block_on SpawnBlocking`

## mise en œuvre `block_on`

En plus de pouvoir implémenter des futurs primitifs, nous avons également toutes les pièces dont nous avons besoin pour construire un exécuteur simple. Dans cette section, nous allons écrire notre propre version de . Ce sera un peu plus simple que la version de ' ; par exemple, il ne prend pas en charge , les variables locales de tâche ou les appels imbriqués (appel à partir de code asynchrone). Mais il suffit d'exécuter notre client et serveur de chat.`block_on` `async_std` `spawn_local` `block_on`

Voici le code :

```

use waker_fn::waker_fn; // Cargo.toml: waker-fn = "1.1"
use futures_lite::pin; // Cargo.toml: futures-lite = "1.11"
use crossbeam::sync::Parker; // Cargo.toml: crossbeam = "0.8"
use std::future::Future;
use std::task::{Context, Poll};

fn block_on<F: Future>(future: F) -> F::Output {
 let parker = Parker::new();
 let unparker = parker.unparker().clone();
 let waker = waker_fn(move || unparker.unpark());
 let mut context = Context::from_waker(&waker);

 pin!(future);

 loop {
 match future.as_mut().poll(&mut context) {

```

```

 Poll::Ready(value) => return value,
 Poll::Pending => parker.park(),
 }
}

```

C'est assez court, mais il se passe beaucoup de choses, alors prenons-le un morceau à la fois.

```

let parker = Parker::new();
let unparker = parker.unparker().clone();

```

Le type de caisse est une primitive de blocage simple : l'appel bloque le thread jusqu'à ce que quelqu'un d'autre appelle le correspondant, que vous obtenez au préalable en appelant . Si vous avez un thread qui n'est pas encore garé, son prochain appel à revenir immédiatement, sans blocage. Notre volonté d'attendre chaque fois que l'avenir n'est pas prêt, et le réveil que nous passons aux futurs le dégarera. crossbeam Parker park() .unpark() Unparker parker.unparker() unpark park block\_on Parker

```

let waker = waker_fn(move || unparker.unpark());

```

La fonction, à partir de la caisse du même nom, crée un à partir d'une fermeture donnée. Ici, nous faisons un qui, lorsqu'il est invoqué, appelle la fermeture . Vous pouvez également créer des wakers en implémentant le trait, mais c'est un peu plus pratique ici. waker\_fn Waker Waker move || unparker.unpark() std::task::Wake waker\_fn

```

pin!(future);

```

Étant donné qu'une variable détient un futur de type , la macro prend possession du futur et déclare une nouvelle variable du même nom dont le type est et qui emprunte le futur. Cela nous donne l'exigence de la méthode. Pour les raisons que nous expliquerons dans la section suivante, les futurs des fonctions et blocs asynchrones doivent être référencés via un avant de pouvoir être interrogés. F pin! Pin<&mut F> Pin<&mut Self> poll Pin

```

loop {
 match future.as_mut().poll(&mut context) {
 Poll::Ready(value) => return value,
 Poll::Pending => parker.park(),
 }
}

```

```
}
```

Enfin, la boucle de sondage est assez simple. Passant un contexte portant notre sillage, nous sondons l'avenir jusqu'à ce qu'il revienne. S'il retourne , nous garons le thread, qui bloque jusqu'à ce qu'il soit appelé. Ensuite, nous réessayons. `Poll::Ready` `Poll::Pending` `waker`

L'appel nous permet de sonder sans renoncer à la propriété; nous expliquerons cela plus en détail dans la section suivante. `as_mut` `future`

## Épinglage

Bien que les fonctions et les blocs asynchrones soient essentiels pour écrire du code asynchrone clair, la gestion de leur avenir nécessite un peu de soin. Le type aide Rust à s'assurer qu'ils sont utilisés en toute sécurité. `Pin`

Dans cette section, nous montrerons pourquoi les futurs des appels et des blocs de fonction asynchrones ne peuvent pas être gérés aussi librement que les valeurs Rust ordinaires. Ensuite, nous montrerons comment sert de « sceau d'approbation » sur les pointeurs sur lesquels on peut compter pour gérer ces contrats à terme en toute sécurité. Enfin, nous montrerons quelques façons de travailler avec des valeurs. `Pin` `Pin`

## Les deux étapes de la vie d'un avenir

Considérez cette fonction asynchrone simple :

```
use async_std::io::prelude::*;
use async_std::{io, net};

async fn fetch_string(address: &str) -> io::Result<String> {
 ❶
 let mut socket = net::TcpStream::connect(address).await❷?;
 let mut buf = String::new();
 socket.read_to_string(&mut buf).await❸?;
 Ok(buf)
}
```

Cela ouvre une connexion TCP à l'adresse donnée et renvoie, sous forme de , tout ce que le serveur veut envoyer. Les points étiquetés ❶, ❷ et ❸ sont les *points de reprise*, les points du code de la fonction asynchrone auxquels l'exécution peut être suspendue. `String`

Supposons que vous l'appeliez, sans attendre, comme suit:

```
let response = fetch_string("localhost:6502");
```

Maintenant est un futur prêt à commencer l'exécution au début de , avec l'argument donné. En mémoire, l'avenir ressemble à [la figure 20-](#)

### 5. response fetch\_string

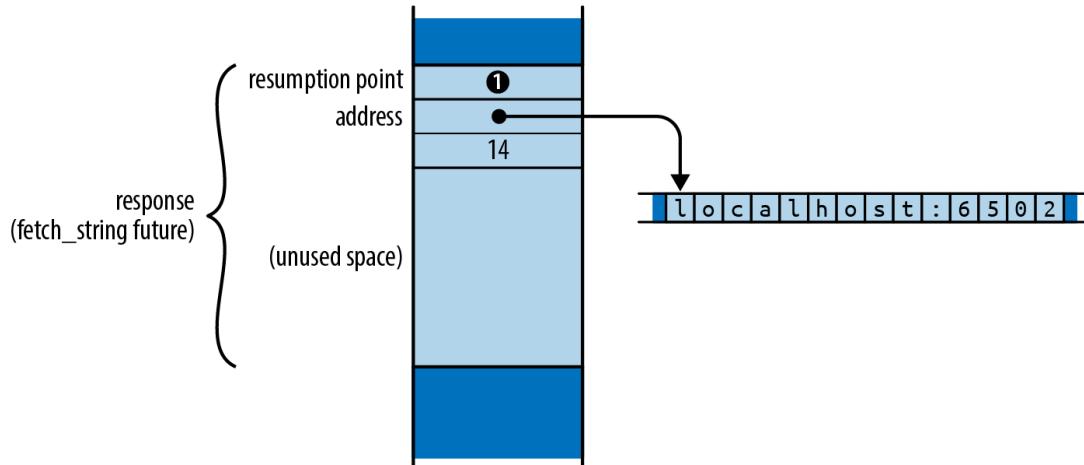


Figure 20-5. L'avenir construit pour un appel à `fetch_string`

Puisque nous venons de créer ce futur, il est dit que l'exécution devrait commencer au point de reprise ①, au sommet du corps de la fonction. Dans cet état, les seules valeurs dont un futur a besoin pour continuer sont les arguments de fonction.

Supposons maintenant que vous interrogez plusieurs fois et qu'il atteigne ce point dans le corps de la fonction: `response`

```
socket.read_to_string(&mut buf).await③?;
```

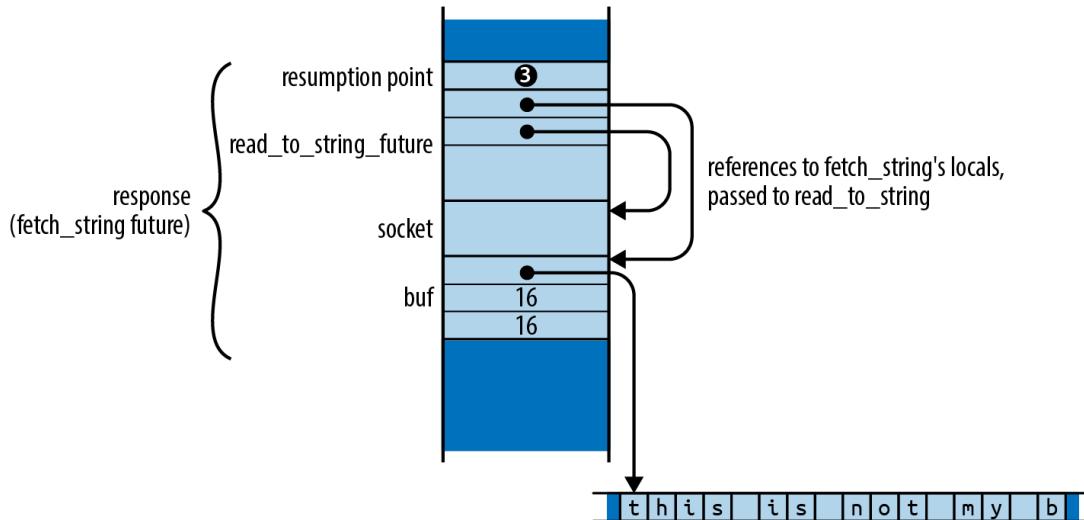
Supposons en outre que le résultat de n'est pas prêt, de sorte que le sondage retourne . À ce stade, l'avenir ressemble à [la figure 20-](#)

### 6. read\_to\_string Poll::Pending

Un futur doit toujours contenir toutes les informations nécessaires pour reprendre l'exécution la prochaine fois qu'il est interrogé. Dans ce cas, c'est:

- Point de reprise ③, disant que l'exécution devrait reprendre dans l'avenir du scrutin. `await read_to_string`
- Les variables qui sont vivantes à ce point de reprise : et . La valeur de n'est plus présente dans le futur, puisque la fonction n'en a plus besoin. `socket buf address`

- Le sous-futur, dont l'expression est au milieu des sondages. `read_to_string` attend



Graphique 20-6. Le même avenir, en pleine attente `read_to_string`

Notez que l'appel à `a` emprunté des références à `et` et `.`. Dans une fonction synchrone, toutes les variables locales vivent sur la pile, mais dans une fonction asynchrone, les variables locales qui sont vivantes sur un doivent être localisées à l'avenir, de sorte qu'elles seront disponibles lorsqu'elles seront à nouveau interrogées. Emprunter une référence à une telle variable emprunte une partie de l'avenir. `read_to_string socket buf await`

Cependant, Rust exige que les valeurs ne soient pas déplacées pendant qu'elles sont empruntées. Supposons que vous deviez déplacer cet avenir vers un nouvel emplacement :

```
let new_variable = response;
```

Rust n'a aucun moyen de trouver toutes les références actives et de les ajuster en conséquence. Au lieu de pointer vers `et` et vers leurs nouveaux emplacements, les références continuent de pointer vers leurs anciens emplacements dans le maintenant non initialisé. Ils sont devenus des pointeurs pendants, comme le montre [la figure 20-7](#).

## 7. socket buf response

Empêcher le déplacement des valeurs empruntées relève généralement de la responsabilité du vérificateur d'emprunt. Le vérificateur d'emprunt traite les variables comme les racines des arbres de propriété, mais contrairement aux variables stockées sur la pile, les variables stockées dans les contrats à terme sont déplacées si le futur lui-même se déplace. Cela signifie que les emprunts et affectent non seulement ce qui peut faire avec ses propres variables, mais ce que son appelant peut faire en toute

sécurité avec , l'avenir qui les réserve. Les futurs des fonctions asynchrones sont un angle mort pour le vérificateur d'emprunt, que Rust doit couvrir d'une manière ou d'une autre s'il veut tenir ses promesses de sécurité de la mémoire. `socket buf fetch_string response`

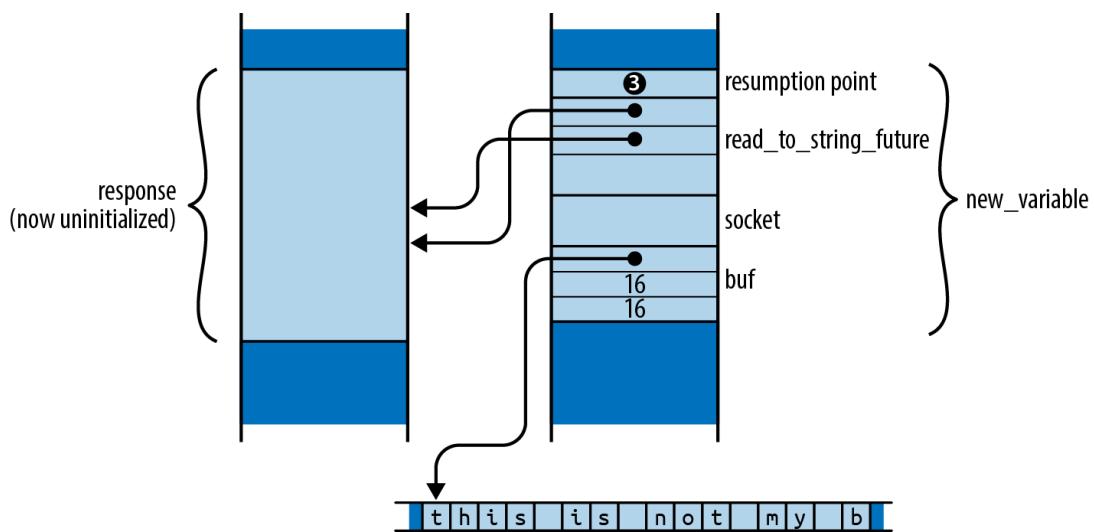


Figure 20-7. (.) , déplacé alors qu'il est emprunté (Rust empêche cela) `fetch_string`

La solution de Rust à ce problème repose sur l'idée que les futurs sont toujours sûrs à se déplacer lorsqu'ils sont créés pour la première fois, et ne deviennent dangereux à déplacer que lorsqu'ils sont interrogés. Un futur qui vient d'être créé en appelant une fonction asynchrone contient simplement un point de reprise et les valeurs d'argument. Ceux-ci ne sont que dans la portée du corps de la fonction asynchrone, qui n'a pas encore commencé l'exécution. Seul le sondage d'un avenir peut emprunter son contenu.

À partir de là, nous pouvons voir que chaque avenir a deux étapes de vie:

- La première étape commence lorsque le futur est créé. Parce que le corps de la fonction n'a pas encore commencé à s'exécuter, aucune partie de celle-ci ne pourrait encore être empruntée. À ce stade, il est aussi sûr de se déplacer que n'importe quelle autre valeur rust.
- La deuxième étape commence la première fois que l'avenir est sondé. Une fois que le corps de la fonction a commencé à s'exécuter, il pourrait emprunter des références à des variables stockées dans le futur, puis attendre, laissant cette partie du futur empruntée. À partir de son premier sondage, nous devons supposer que l'avenir pourrait ne pas être sûr de bouger.

La flexibilité de la première étape de la vie est ce qui nous permet de passer des futures à et et appeler des méthodes d'adaptateur comme et , qui prennent toutes des futures par valeur. En fait, même l'appel de fonc-

tion asynchrone qui a créé l'avenir en premier lieu devait le renvoyer à l'appelant; c'était aussi une mesure. `block_on spawn race fuse`

Pour entrer dans sa deuxième étape de vie, l'avenir doit être sondé. La méthode exige que l'avenir soit transmis sous forme de valeur. `Pin` est un wrapper pour les types de pointeurs (comme `Box`) qui limite la façon dont les pointeurs peuvent être utilisés, garantissant que leurs référents (comme `Cell`) ne peuvent plus jamais être déplacés. Vous devez donc produire un pointeur enveloppé vers l'avenir avant de pouvoir

l'interroger. `poll Pin<&mut Self> Pin &mut Self Self Pin`

C'est donc la stratégie de Rust pour assurer la sécurité de l'avenir : un avenir ne peut pas devenir dangereux à déplacer tant qu'il n'est pas sondé ; vous ne pouvez pas interroger un futur tant que vous n'avez pas construit un pointeur encapsulé vers celui-ci ; et une fois que vous avez fait cela, l'avenir ne peut pas être déplacé. `Pin`

« Une valeur que vous ne pouvez pas déplacer » semble impossible: les mouvements sont partout dans Rust. Nous expliquerons exactement comment protéger les futurs dans la section suivante. `Pin`

Bien que cette section ait discuté des fonctions asynchrones, tout ici s'applique également aux blocs asynchrones. Un futur nouvellement créé d'un bloc asynchrone capture simplement les variables qu'il utilisera à partir du code environnant, comme une fermeture. Seul l'interrogation de l'avenir peut créer des références à son contenu, ce qui le rend dangereux à déplacer.

Gardez à l'esprit que cette fragilité de déplacement est limitée aux futurs de fonctions et de blocs asynchrones, avec leurs implémentations spéciales générées par le compilateur. Si vous implémentez à la main pour vos propres types, comme nous l'avons fait pour notre type [dans « Invoking Wakers: spawning blocking »](#), ces futurs sont parfaitement sûrs à déplacer avant et après avoir été interrogés. Dans toute implémentation manuscrite, le vérificateur d'emprunt s'assure que toutes les références que vous aviez empruntées à des parties de sont parties disparues au moment du retour. Ce n'est que parce que les fonctions et les blocs asynchrones ont le pouvoir de suspendre l'exécution au milieu d'un appel de fonction, avec des emprunts en cours, que nous devons gérer leur avenir avec soin. `Future Future SpawnBlocking poll self poll`

## Pointeurs épingleés

Le type est un wrapper pour les pointeurs vers les futurs qui limite la façon dont les pointeurs peuvent être utilisés pour s'assurer que les futurs ne peuvent pas être déplacés une fois qu'ils ont été interrogés. Ces restrictions peuvent être levées pour les futurs qui ne craignent pas d'être déplacés, mais elles sont essentielles pour interroger en toute sécurité les futurs des fonctions et des blocs asynchrones. Pin

Par *pointeur*, nous entendons tout type qui implémente , et éventuellement . Un pointeur enroulé autour d'un pointeur est appelé *pointeur épingle*. et sont typiques. Deref DerefMut Pin Pin<&mut T> Pin<Box<T>>

La définition de dans la bibliothèque standard est simple : Pin

```
pub struct Pin<P> {
 pointer: P,
}
```

Notez que le champ *n'est pas* . Cela signifie que la seule façon de construire ou d'utiliser un est à travers les méthodes soigneusement choisies que le type fournit. pointer pub Pin

Compte tenu de l'avenir d'une fonction ou d'un bloc asynchrone, il n'y a que quelques façons d'obtenir un pointeur épingle vers celui-ci :

- La macro, à partir de la caisse, ombre une variable de type avec une nouvelle de type . La nouvelle variable pointe vers la valeur de l'original, qui a été déplacée vers un emplacement temporaire anonyme sur la pile. Lorsque la variable sort de la portée, la valeur est supprimée. Nous avons utilisé dans notre mise en œuvre pour épingle l'avenir que nous voulions sonder. pin! futures-lite T Pin<&mut T> pin! block\_on
- Le constructeur de la bibliothèque standard prend possession d'une valeur de n'importe quel type, la déplace dans le tas et renvoie un fichier. Box::pin T Pin<Box<T>>
- Pin<Box<T>> implémente , prend donc possession et vous redonne une boîte épingle pointant vers la même sur le tas. From<Box<T>> Pin::from(boxed) boxed T

Chaque façon d'obtenir un pointeur épingle vers ces futurs implique de renoncer à la propriété de l'avenir, et il n'y a aucun moyen de le récupérer. Le pointeur épingle lui-même peut être déplacé comme bon vous semble, bien sûr, mais déplacer un pointeur ne déplace pas son référent. Ainsi, la possession d'un pointeur épingle vers un avenir sert de preuve que

vous avez définitivement renoncé à la capacité de déplacer cet avenir. C'est tout ce que nous devons savoir pour qu'il puisse être interrogé en toute sécurité.

Une fois que vous avez épinglé un futur, si vous souhaitez l'interroger, tous les types ont une méthode qui déréférence le pointeur et renvoie ce qui l'exige. `Pin<pointer to T> as_mut Pin<&mut T> poll`

La méthode peut également vous aider à sonder un avenir sans renoncer à la propriété. Notre implémentation l'a utilisé dans ce rôle: `as_mut block_on`

```
pin!(future);

loop {
 match future.as_mut().poll(&mut context) {
 Poll::Ready(value) => return value,
 Poll::Pending => parker.park(),
 }
}
```

Ici, la macro a été redéclarée en tant que , nous pourrions donc simplement passer cela à . Mais les références mutables ne sont pas , donc ne peuvent pas l'être non plus, ce qui signifie que l'appel direct prendrait possession de , laissant l'itération suivante de la boucle avec une variable non initialisée. Pour éviter cela, nous appelons à réembarquer un nouveau pour chaque itération de boucle. `pin! future Pin<&mut F> poll Copy Pin<&mut`

```
F> Copy future.poll() future future.as_mut() Pin<&mut F>
```

Il n'y a aucun moyen d'obtenir une référence à un futur épinglé: si vous le pouviez, vous pourriez l'utiliser ou le déplacer et mettre un avenir différent à sa place. `&mut std::mem::replace std::mem::swap`

La raison pour laquelle nous n'avons pas à nous soucier d'épingler des futurs dans du code asynchrone ordinaire est que les moyens les plus courants d'obtenir la valeur d'un futur – en l'attendant ou en passant à un exécuteur testamentaire – prennent tous possession du futur et gèrent l'épinglage en interne. Par exemple, notre implémentation s'approprie l'avenir et utilise la macro pour produire le nécessaire pour interroger. Une expression s'approprie également l'avenir et utilise une approche similaire à la macro en interne. `block_on pin! Pin<&mut F> await pin!`

## Le trait Unpin

Cependant, tous les contrats à terme ne nécessitent pas ce type de manipulation prudente. Pour toute implémentation manuscrite d'un type ordinaire, comme notre type mentionné précédemment, les restrictions sur la construction et l'utilisation de pointeurs épinglés sont inutiles. Future SpawnBlocking

Ces types durables mettent en œuvre le trait de marqueur: Unpin

```
trait Unpin { }
```

Presque tous les types dans Rust implémentent automatiquement , en utilisant un support spécial dans le compilateur. La fonction asynchrone et les contrats à terme de bloc sont les exceptions à cette règle. Unpin

Pour les types, n'impose aucune restriction que ce soit. Vous pouvez créer un pointeur épingle à partir d'un pointeur ordinaire avec et le récupérer avec . Le lui-même passe le long des propres implémentations et des implémentations du

```
pointeur. Unpin Pin Pin::new Pin::into_inner Pin Deref DerefMut
```

Par exemple, implémente , afin que nous puissions écrire: String Unpin

```
let mut string = "Pinned?".to_string();
let mut pinned: Pin<&mut String> = Pin::new(&mut string);

pinned.push_str(" Not");
Pin::into_inner(pinned).push_str(" so much.");

let new_home = string;
assert_eq!(new_home, "Pinned? Not so much.");
```

Même après avoir créé un , nous avons un accès mutable complet à la chaîne et pouvons la déplacer vers une nouvelle variable une fois que la a été consommée par et que la référence mutable a disparu. Donc, pour les types qui sont – qui sont presque tous – est une enveloppe ennuyeuse autour des pointeurs vers ce type. Pin<&mut String> Pin into\_inner Unpin Pin

Cela signifie que lorsque vous implémentez pour vos propres types, votre implémentation peut traiter comme si elle était , pas . L'épinglage devient

quelque chose que vous pouvez la plupart du temps ignorer. Future Unpin poll self &mut Self Pin<&mut Self>

Il peut être surprenant d'apprendre cela et de mettre en œuvre , même si ce n'est pas le cas. Cela ne se lit pas bien – comment peut-on être ? – mais si vous réfléchissez bien à ce que chaque terme signifie, cela a du sens. Même s'il n'est pas sûr de se déplacer une fois qu'il a été interrogé, un pointeur vers celui-ci est toujours sûr à déplacer, interrogé ou non. Seul le pointeur se déplace ; son référent fragile reste en place. Pin<&mut F> Pin<Box<F>> Unpin F Pin Unpin F

Ceci est utile pour savoir quand vous souhaitez passer l'avenir d'une fonction asynchrone ou d'un bloc à une fonction qui n'accepte que les futurs. (De telles fonctions sont rares dans , mais moins ailleurs dans l'écosystème asynchrone.) est même si ce n'est pas le cas, donc l'application à une fonction asynchrone ou à un futur de bloc vous donne un avenir que vous pouvez utiliser n'importe où, au prix d'une allocation de tas. Unpin async\_std Pin<Box<F>> Unpin F Box::pin

Il existe différentes méthodes dangereuses pour travailler avec qui vous permettent de faire ce que vous voulez avec le pointeur et sa cible, même pour les types de cible qui ne le sont pas . Mais comme expliqué au [chapitre 22](#), Rust ne peut pas vérifier que ces méthodes sont utilisées correctement; vous devenez responsable d'assurer la sécurité du code qui les utilise. Pin Unpin

## Quand le code asynchrone est-il utile ?

Le code asynchrone est plus difficile à écrire que le code multithread. Vous devez utiliser les bonnes primitives d'E/S et de synchronisation, diviser les calculs de longue durée à la main ou les faire tourner sur d'autres threads, et gérer d'autres détails comme l'épinglage qui ne se produisent pas dans le code threadé. Alors, quels sont les avantages spécifiques du code asynchrone ?

Deux affirmations que vous entendrez souvent ne résistent pas à une inspection minutieuse:

- « Le code asynchrone est idéal pour les E/S. » Ce n'est pas tout à fait exact. Si votre application passe son temps à attendre des E/S, le fait de l'asynchroniser ne permettra pas d'exécuter ces E/S plus rapidement. Il n'y a rien dans les interfaces d'E/S asynchrones généralement utilisées aujourd'hui qui les rende plus efficaces que leurs homologues syn-

chrones. Le système d'exploitation a le même travail à faire dans les deux sens. (En fait, une opération d'E/S asynchrone qui n'est pas prête doit être réessayée ultérieurement, il faut donc deux appels système pour terminer au lieu d'un.)

- « Le code asynchrone est plus facile à écrire que le code multithread. » Dans des langages comme JavaScript et Python, cela pourrait bien être vrai. Dans ces langages, les programmeurs utilisent `async/await` comme une forme de concurrence bien conduite : il n'y a qu'un seul thread d'exécution, et les interruptions ne se produisent qu'au niveau des expressions, il n'y a donc souvent pas besoin d'un mutex pour garder les données cohérentes : n'attendez pas pendant que vous êtes en train de l'utiliser ! Il est beaucoup plus facile de comprendre votre code lorsque les changements de tâches ne se produisent qu'avec votre autorisation explicite. `await`

Mais cet argument ne se répercute pas sur Rust, où les fils ne sont pas aussi gênants. Une fois votre programme compilé, il est exempt de courses de données. Le comportement non déterministe se limite aux fonctionnalités de synchronisation telles que les mutex, les canaux, les atomes, etc., qui ont été conçues pour y faire face. Le code asynchrone n'a donc aucun avantage unique pour vous aider à voir quand d'autres threads pourraient vous affecter ; c'est clair dans *tout* code Rust sûr. Et bien sûr, le support asynchrone de Rust brille vraiment lorsqu'il est utilisé en combinaison avec des fils. Il serait dommage d'y renoncer.

Alors, quels sont les vrais avantages du code asynchrone ?

- *Les tâches asynchrones peuvent utiliser moins de mémoire.* Sous Linux, l'utilisation de la mémoire d'un thread commence à 20 Kio, en comptant à la fois l'espace utilisateur et l'espace noyau.<sup>2</sup> Les futurs peuvent être beaucoup plus petits: les futurs de notre serveur de chat ont une taille de quelques centaines d'octets et sont de plus en plus petits à mesure que le compilateur Rust s'améliore.
- *Les tâches asynchrones sont plus rapides à créer.* Sous Linux, la création d'un thread prend environ 15 µs. La génération d'une tâche asynchrone prend environ 300 ns, soit environ un cinquantième du temps.
- *Les changements de contexte sont plus rapides entre les tâches asynchrones qu'entre les threads du système d'exploitation,* 0,2 µs contre 1,7 µs sous Linux.<sup>3</sup> Cependant, ce sont les meilleurs chiffres pour chacun : si le commutateur est dû à la disponibilité des E/S, les deux coûts augmentent à 1,7 µs. Que le basculement soit entre les threads ou les tâches sur différents cœurs de processeur fait également une grande différence: la communication entre les cœurs est très lente.

Cela nous donne un indice sur les types de problèmes que le code asynchrone peut résoudre. Par exemple, un serveur asynchrone peut utiliser moins de mémoire par tâche et ainsi être en mesure de gérer plus de connexions simultanées. (C'est probablement là que le code asynchrone a la réputation d'être « bon pour les E/S ».) Ou, si votre conception est naturellement organisée comme de nombreuses tâches indépendantes communiquant entre elles, alors de faibles coûts par tâche, des temps de création courts et des changements de contexte rapides sont tous des avantages importants. C'est pourquoi les serveurs de chat sont l'exemple classique de programmation asynchrone, mais les jeux multi-joueurs et les routeurs réseau seraient probablement également de bonnes utilisations.

Dans d'autres situations, les arguments en faveur de l'utilisation de l'async sont moins clairs. Si votre programme dispose d'un pool de threads effectuant des calculs lourds ou restant inactifs en attendant la fin des E/S, les avantages énumérés précédemment n'ont probablement pas une grande influence sur ses performances. Vous devrez optimiser votre calcul, trouver une connexion Internet plus rapide ou faire autre chose qui affecte réellement le facteur limitant.

Dans la pratique, chaque compte rendu de la mise en œuvre de serveurs à volume élevé que nous avons pu trouver soulignait l'importance de la mesure, du réglage et d'une campagne incessante pour identifier et supprimer les sources de conflit entre les tâches. Une architecture asynchrone ne vous permettra pas d'ignorer ce travail. En fait, bien qu'il existe de nombreux outils prêts à l'emploi pour évaluer le comportement des programmes multithread, les tâches asynchrones Rust sont invisibles pour ces outils et nécessitent donc leur propre outillage. (Comme un sage aîné l'a dit un jour : « Maintenant, vous avez *deux* problèmes. »)

Même si vous n'utilisez pas de code asynchrone maintenant, il est bon de savoir que l'option est là si jamais vous avez la chance d'être beaucoup plus occupé que vous ne l'êtes maintenant.

- 1** Si vous avez réellement besoin d'un client HTTP, envisagez d'utiliser l'une des nombreuses excellentes caisses comme ou qui feront le travail correctement et de manière asynchrone. Ce client parvient principalement à obtenir des redirections HTTPS. `surf reqwest`
- 2** Cela inclut la mémoire du noyau et compte les pages physiques allouées au thread, et non les pages virtuelles qui doivent encore être allouées. Les chiffres sont similaires sur macOS et Windows.

- 3** Les commutateurs de contexte Linux étaient également de l'ordre de 0,2 µs, jusqu'à ce que le noyau soit obligé d'utiliser des techniques plus lentes en raison de failles de sécurité du processeur.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

# Chapitre 21. Macros

*A cento (du latin pour « patchwork ») est un poème entièrement composé de vers cités d'un autre poète.*

—Matt Madden

Rust prend en charge les *macros*, un moyen d'étendre le langage d'une manière qui va au-delà de ce que vous pouvez faire avec les seules fonctions. Par exemple, nous avons vu la macro, ce qui est pratique pour les tests: `assert_eq!`!

```
assert_eq!(gcd(6, 10), 2);
```

Cela aurait pu être écrit comme une fonction générique, mais la macro fait plusieurs choses que les fonctions ne peuvent pas faire. La première est que lorsqu'une assertion échoue, génère un message d'erreur contenant le nom de fichier et le numéro de ligne de l'assertion. Les fonctions n'ont aucun moyen d'obtenir cette information. Les macros le peuvent, car leur façon de travailler est complètement différente. `assert_eq!` `assert_eq!`!

Les macros sont une sorte de raccourci. Pendant la compilation, avant que les types ne soient vérifiés et bien avant qu'un code machine ne soit généré, chaque appel de macro est *développé*, c'est-à-dire remplacé par du code Rust. L'appel de macro précédent s'étend à quelque chose d'à peu près comme ceci :

```
match (&gcd(6, 10), &2) {
 (left_val, right_val) => {
 if !(*left_val == *right_val) {
 panic!("assertion failed: `{} == {}`",
 left_val, right_val);
 }
 }
}
```

`panic!` est également une macro, qui elle-même s'étend à encore plus de code Rust (non montré ici). Ce code utilise deux autres macros et . Une fois que chaque appel macro dans la caisse est complètement étendu, Rust passe à la phase suivante de compilation. `file!()` `line!()`

Au moment de l'exécution, un échec d'assertion ressemblerait à ceci (et indiquerait un bogue dans la fonction, car c'est la bonne réponse)

```
: gcd() 2
```

```
thread 'main' panicked at 'assertion failed: `!(left == right)`', (left: `1
right: `2`)', gcd.rs:7
```

Si vous venez de C++, vous avez peut-être eu de mauvaises expériences avec les macros. Les macros Rust adoptent une approche différente, similaire à celle de Scheme. Par rapport aux macros C++, les macros Rust sont mieux intégrées au reste du langage et donc moins sujettes aux erreurs. Les appels de macro sont toujours marqués d'un point d'exclamation, de sorte qu'ils se démarquent lorsque vous lisez du code, et ils ne peuvent pas être appelés accidentellement lorsque vous vouliez appeler une fonction. Les macros Rust n'insèrent jamais de crochets ou de parenthèses inégalés. Et les macros Rust sont livrées avec une correspondance de motifs, ce qui facilite l'écriture de macros à la fois maintenables et attrayantes à utiliser. `syntax-rules`

Dans ce chapitre, nous allons montrer comment écrire des macros à l'aide de plusieurs exemples simples. Mais comme beaucoup de Rust, les macros récompensent une compréhension profonde, nous allons donc parcourir la conception d'une macro plus compliquée qui nous permet d'intégrer des littéraux JSON directement dans nos programmes. Mais il y a plus dans les macros que ce que nous pouvons couvrir dans ce livre, nous terminerons donc avec quelques conseils pour une étude plus approfondie, à la fois des techniques avancées pour les outils que nous vous avons montrés ici, et pour une installation encore plus puissante appelée *macros procédurales*.

## Principes de base des macros

[La figure 21-1](#) montre une partie du code source de la macro `assert_eq!`

`macro_rules!` est le principal moyen de définir des macros dans Rust. Notez qu'il n'y a pas d'après dans cette définition de macro : le n'est inclus que lors de l'appel d'une macro, pas lors de sa définition. `! assert_eq !`

Toutes les macros ne sont pas définies de cette façon : quelques-unes, comme `,`, et elle-même, sont intégrées dans le compilateur, et nous parlerons d'une autre approche, appelée macros procédurales, à la fin de ce chapitre. Mais pour la plupart, nous nous concentrerons sur `,` qui est

(jusqu'à présent) le moyen le plus simple d'écrire le vôtre. `file! line! macro_rules! macro_rules!`

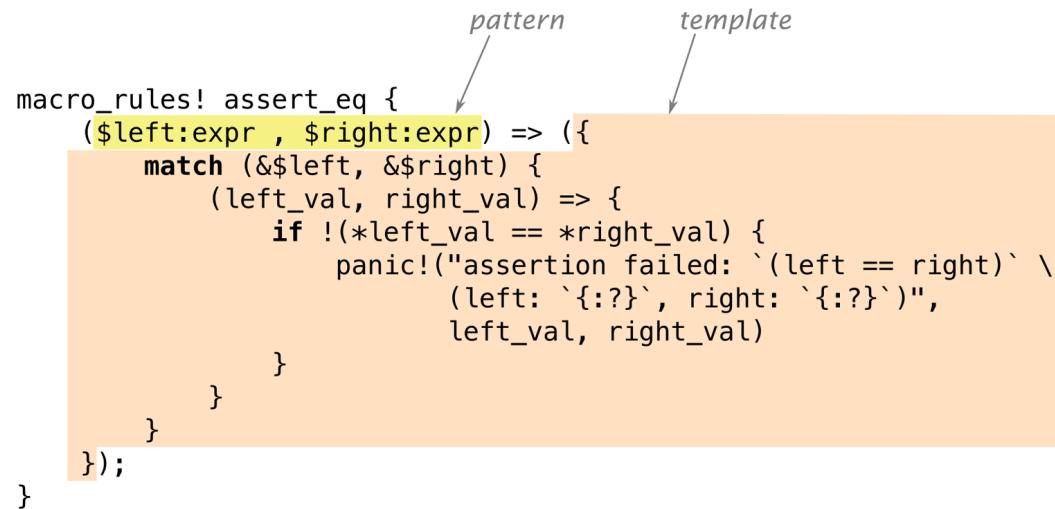
Une macro définie avec fonctionne entièrement par correspondance de motifs. Le corps d'une macro n'est qu'une série de règles : `macro_rules!`

```
(pattern1) => (template1);

(pattern2) => (template2);

...

macro_rules! assert_eq {
 ($left:expr , $right:expr) => ({
 match (&$left, &$right) {
 (left_val, right_val) => {
 if !(*left_val == *right_val) {
 panic!("assertion failed: `($left == $right)`\n($left: `{:?}` , $right: `{:?}`)",
 left_val, right_val)
 }
 }
 }
 });
}
```



The diagram shows two arrows pointing from labels to specific parts of the code. One arrow points from the label 'pattern' to the first argument of the macro call '\$left:expr , \$right:expr'. Another arrow points from the label 'template' to the body of the macro rule, which contains a match expression.

Graphique 21-1. La macro `assert_eq!`

La version de la [figure 21-1](#) n'a qu'un seul modèle et un seul modèle. `assert_eq!`

Incidentement, vous pouvez utiliser des crochets ou des accolades au lieu de parenthèses autour du motif ou du modèle; cela ne fait aucune différence pour Rust. De même, lorsque vous appelez une macro, ceux-ci sont tous équivalents :

```
assert_eq!(gcd(6, 10), 2);
assert_eq![gcd(6, 10), 2];
assert_eq!{gcd(6, 10), 2}
```

La seule différence est que les points-virgules sont généralement facultatifs après les accolades. Par convention, nous utilisons des parenthèses lors de l'appel, des crochets pour , et des accolades pour  
. `assert_eq! vec! macro_rules!`

Maintenant que nous avons montré un exemple simple de l'expansion d'une macro et la définition qui l'a générée, nous pouvons entrer dans les détails nécessaires pour le mettre en œuvre:

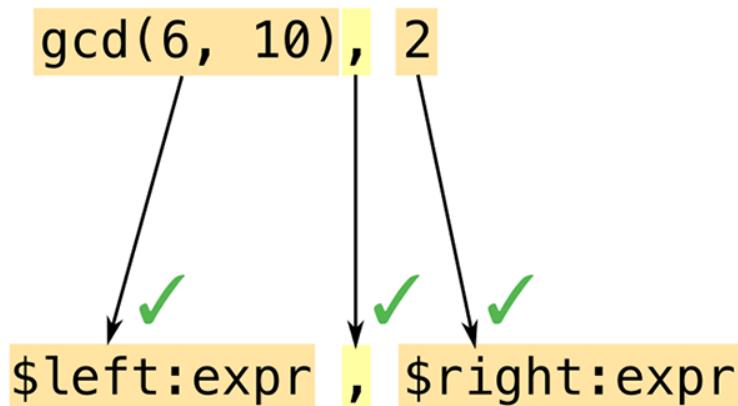
- Nous vous expliquerons exactement comment Rust s'y prend pour trouver et développer des définitions de macro dans votre programme.
- Nous soulignerons quelques subtilités inhérentes au processus de génération de code à partir de modèles de macro.
- Enfin, nous montrerons comment les modèles gèrent la structure répétitive.

## Principes de base de l'expansion macro

Rust développe les macros très tôt pendant la compilation. Le compilateur lit votre code source du début à la fin, en définissant et en développant les macros au fur et à mesure. Vous ne pouvez pas appeler une macro avant qu'elle ne soit définie, car Rust développe chaque appel de macro avant même d'examiner le reste du programme. (En revanche, les fonctions et autres éléments n'ont pas besoin d'être dans un ordre particulier. Vous pouvez appeler une fonction qui ne sera définie que plus tard dans la caisse.)

Lorsque Rust développe un appel macro, ce qui se passe ressemble beaucoup à l'évaluation d'une expression. Rust correspond d'abord aux arguments contre le modèle, comme le montre [la figure 21-](#)

[2. assert\\_eq! match](#)



Graphique 21-2. Développement d'une macro, partie 1 : correspondance des modèles avec les arguments

Les modèles de macro sont un mini-langage dans Rust. Ce sont essentiellement des expressions régulières pour faire correspondre le code. Mais là où les expressions régulières fonctionnent sur des caractères, les modèles fonctionnent sur des *jetons* – *les nombres*, les noms, les signes de ponctuation, etc. qui sont les éléments constitutifs des programmes Rust. Cela signifie que vous pouvez utiliser librement les commentaires et les espaces blancs dans les modèles de macro pour les rendre aussi lisibles que possible. Les commentaires et les espaces blancs ne sont pas des jetons, ils n'affectent donc pas la correspondance.

Une autre différence importante entre les expressions régulières et les modèles de macro est que les parenthèses, les crochets et les accolades se produisent toujours par paires correspondantes dans Rust. Ceci est vérifié avant que les macros ne soient développées, non seulement dans les modèles de macro, mais dans toute la langue.

Dans cet exemple, notre motif contient le *fragment*, qui indique à Rust de correspondre à une expression (dans ce cas, `)` et de lui attribuer le nom . Rust fait ensuite correspondre la virgule dans le motif avec la virgule suivant les arguments de `.` Tout comme les expressions régulières, les motifs n'ont que quelques caractères spéciaux qui déclenchent un comportement de correspondance intéressant; tout le reste, comme cette virgule, doit correspondre textuellement, sinon la correspondance échoue. Enfin, Rust correspond à l'expression et lui donne le nom `. $left:expr gcd(6, 10) $left gcd 2 $right`

Les deux fragments de code de ce modèle sont de type : ils attendent des expressions. Nous verrons d'autres types de fragments de code dans [« Types de fragments »](#), `expr`

Étant donné que ce modèle correspond à tous les arguments, Rust développe le *modèle* correspondant ([Figure 21-3](#)).

```
{
 match (&$left, &$right) {
 (left_val, right_val) => {
 if !(*left_val == *right_val) {
 panic!("assertion failed: `left == right`" \
 (left: `{:?}`, right: `{:?}`)",
 left_val, right_val)
 }
 }
 }
}
```

Graphique 21-3. Développement d'une macro, partie 2 : remplissage du modèle

Rust remplace et avec les fragments de code qu'il a trouvés lors de la correspondance. `$left $right`

C'est une erreur courante d'inclure le type de fragment dans le modèle de sortie : écrire plutôt que simplement `.` Rust ne détecte pas immédiatement ce genre d'erreur. Il voit comme une substitution, puis il traite comme tout le reste dans le modèle: les jetons à inclure dans la sortie de la macro. Ainsi, les erreurs ne se produiront pas tant que vous *n'aurez pas appelé* la macro ; ensuite, il générera une fausse sortie qui ne se compilera pas. Si vous recevez des messages d'erreur comme et lors de l'utilisation d'une nouvelle macro, vérifiez-la pour cette erreur. ([« Débogage des](#)

[macros](#) » offre des conseils plus généraux pour des situations comme celle-ci.) \$left:expr \$left \$left :expr cannot find type `expr` in this scope help: maybe you meant to use a path separator here

Les modèles de macro ne sont pas très différents de l'un des douze langages de modèles couramment utilisés dans la programmation Web. La seule différence – et c'est une différence significative – est que la sortie est du code Rust.

## Conséquences imprévues

Le branchement de fragments de code dans des modèles est subtilement différent du code normal qui fonctionne avec des valeurs. Ces différences ne sont pas toujours évidentes au début. La macro que nous avons examinée, , contient des morceaux de code légèrement étranges pour des raisons qui en disent long sur la programmation macro. Regardons deux morceaux amusants en particulier. `assert_eq!`

Tout d'abord, pourquoi cette macro crée-t-elle les variables `et` ? Y a-t-il une raison pour laquelle nous ne pouvons pas simplifier le modèle pour ressembler à ceci? `left_val right_val`

```
if !($left == $right) {
 panic!("assertion failed: `($left == right)`\n"
 "(left: `{:?}`, right: `{:?}`)", $left, $right)
}
```

Pour répondre à cette question, essayez d'étendre mentalement l'appel macro . Quel serait le résultat? Naturellement, Rust brancherait les expressions correspondantes dans le modèle à plusieurs endroits. Cela semble être une mauvaise idée d'évaluer à nouveau les expressions lors de la création du message d'erreur, et pas seulement parce que cela prendrait deux fois plus de temps: puisque supprime une valeur d'un vecteur, cela produira une valeur différente la deuxième fois que nous l'appellerons! C'est pourquoi la macro réelle calcule et ne calcule qu'une seule fois et stocke ses valeurs. `assert_eq!(letters.pop(), Some('z'))`

Passons à la deuxième question : pourquoi cette macro emprunte-t-elle des références aux valeurs de et ? Pourquoi ne pas simplement stocker les valeurs dans des variables, comme celle-ci ? `$left $right`

```
macro_rules! bad_assert_eq {
 ($left:expr, $right:expr) => ({
```

```

 match ($left, $right) {
 (left_val, right_val) => {
 if !(left_val == right_val) {
 panic!("assertion failed" /* ... */);
 }
 }
 });
 }
}

```

Pour le cas particulier que nous avons examiné, où les arguments macro sont des entiers, cela fonctionnerait bien. Mais si l'appelant passait, disons, une variable comme `ou`, ce code déplacerait la valeur hors de la variable!

```
String $left $right
```

```

fn main() {
 let s = "a rose".to_string();
 bad_assert_eq!(s, "a rose");
 println!("confirmed: {} is a rose", s); // error: use of moved value
}

```

Comme nous ne voulons pas que les assertions déplacent des valeurs, la macro emprunte plutôt des références.

(Vous vous êtes peut-être demandé pourquoi la macro utilise plutôt que de définir les variables. Nous nous sommes demandés aussi. Il s'avère qu'il n'y a pas de raison particulière à cela. aurait été équivalent.)

```
match let let
```

En bref, les macros peuvent faire des choses surprenantes. Si des choses étranges se produisent autour d'une macro que vous avez écrite, il y a fort à parier que la macro est à blâmer.

Un bogue que vous *ne verrez pas* est ce bogue de macro C++ classique :

```
// buggy C++ macro to add 1 to a number
#define ADD_ONE(n) n + 1
```

Pour des raisons familières à la plupart des programmeurs C++, et qui ne valent pas la peine d'être expliquées en détail ici, un code banal, comme ou produit des résultats très surprenants avec cette macro. Pour le ré-soudre, vous devez ajouter d'autres parenthèses à la définition de macro. Ce n'est pas nécessaire dans Rust, car les macros Rust sont mieux intégrées au langage. Rust sait quand il manipule des expressions, il ajoute

donc efficacement des parenthèses chaque fois qu'il colle une expression dans une autre. `ADD_ONE(1) * 10 ADD_ONE(1 << 4)`

## Répétition

La macro standard se présente sous deux formes : `vec!` !

```
// Repeat a value N times
let buffer = vec![0_u8; 1000];

// A list of values, separated by commas
let numbers = vec!["udon", "ramen", "soba"];
```

Il peut être mis en œuvre comme ceci:

```
macro_rules! vec {
 ($elem:expr ; $n:expr) => {
 ::std::vec::from_elem($elem, $n)
 };
 ($($x:expr),*) => {
 <[_]>::into_vec(Box::new([$($x),*]))
 };
 ($($x:expr),+ ,) => {
 vec![$($x),*]
 };
}
```

Il y a trois règles ici. Nous expliquerons le fonctionnement de plusieurs règles, puis examinerons chaque règle à tour de rôle.

Lorsque Rust développe un appel de macro comme `vec!`, il commence par essayer de faire correspondre les arguments avec le modèle de la première règle, dans ce cas `.` . Cela ne correspond pas: `est` une expression, mais le motif nécessite un point-virgule après cela, et nous n'en avons pas. Rust passe donc à la deuxième règle, et ainsi de suite. Si aucune règle ne correspond, c'est une erreur. `vec![1, 2, 3] 1, 2, 3 $elem:expr ; $n:expr 1`

La première règle gère des utilisations telles que `.` . Il arrive qu'il existe une fonction standard (mais non documentée), qui fait exactement ce qui est nécessaire ici, donc cette règle est simple. `vec![0u8; 1000] std::vec::from_elem`

La deuxième règle gère `.` . Le modèle, `,` , utilise une fonctionnalité que nous n'avons jamais vue auparavant: la répétition. Il correspond à 0 expression ou plus, séparées par des virgules. Plus généralement, la syntaxe est

```
utilisée pour faire correspondre n'importe quelle liste séparée par des virgules, où chaque élément de la liste correspond à . vec! ["udon" , "ramen" , "soba"] $($x:expr),* $(PATTERN),* PATTERN
```

Le ici a la même signification que dans les expressions régulières (« 0 ou plus ») bien qu'il soit vrai que les regexps n'ont pas de répétiteur spécial. Vous pouvez également utiliser pour exiger au moins une correspondance, ou pour zéro ou une correspondance. [Le tableau 21-1](#) présente l'ensemble des modèles de répétition. \* , \* + ?

Tableau 21-1. Modèles de répétition

| Modèle                    | Signification                                                      |
|---------------------------|--------------------------------------------------------------------|
| <code>\$( ... )</code>    | Faire correspondre 0 fois ou plus sans séparateur                  |
| <code>*</code>            |                                                                    |
| <code>\$( ... ) ,*</code> | Faire correspondre 0 fois ou plus, séparées par des virgules       |
| <code>\$( ... ) ;*</code> | Faire correspondre 0 fois ou plus, séparés par des points-virgules |
| <code>\$( ... )</code>    | Faire correspondre 1 fois ou plus sans séparateur                  |
| <code>+</code>            |                                                                    |
| <code>\$( ... ) ,+</code> | Faire correspondre 1 fois ou plus, séparées par des virgules       |
| <code>\$( ... ) ;+</code> | Faire correspondre 1 fois ou plus, séparés par des points-virgules |
| <code>\$( ... ) ?</code>  | Faire correspondre 0 ou 1 fois sans séparateur                     |
| <code>\$( ... ) ,?</code> | Faire correspondre 0 ou 1 fois, séparés par des virgules           |
| <code>\$( ... ) ;?</code> | Faire correspondre 0 ou 1 fois, séparés par des points-virgules    |

Le fragment de code n'est pas seulement une expression unique, mais une liste d'expressions. Le modèle de cette règle utilise également la syn-

taxe de répétition : \$x

```
<[_]>::into_vec(Box::new([$($x),*]))
```

Encore une fois, il existe des méthodes standard qui font exactement ce dont nous avons besoin. Ce code crée un tableau en boîte, puis utilise la méthode pour convertir le tableau en boîte en vecteur. [T]::into\_vec

Le premier bit, , est une façon inhabituelle d'écrire le type « tranche de quelque chose », tout en s'attendant à ce que Rust déduise le type d'élément. Les types dont les noms sont des identificateurs simples peuvent être utilisés dans des expressions sans tracas, mais des types tels que , , ou doivent être enveloppés dans des crochets. <[\_]> fn() &str [\_]

La répétition entre à la fin du modèle, où nous avons . C'est la même syntaxe que nous avons vue dans le modèle. Il parcourt la liste des expressions que nous avons appariées et les insère toutes dans le modèle, séparées par des virgules. \$( \$x ),\* \$( ... ),\* \$x

Dans ce cas, la sortie répétée ressemble à l'entrée. Mais cela ne doit pas nécessairement être le cas. Nous aurions pu écrire la règle comme ceci :

```
($($x:expr),*) => {
 {
 let mut v = Vec::new();
 $(v.push($x);)*
 v
 }
};
```

Ici, la partie du modèle qui lit insère un appel à pour chaque expression dans . Un bras macro peut s'étendre à une séquence d'expressions, mais ici nous n'avons besoin que d'une seule expression, nous enveloppons donc l'assemblage du vecteur dans un bloc. \$( v.push(\$x);
)\* v.push() \$x

Contrairement au reste de Rust, les motifs utilisés ne prennent pas automatiquement en charge une virgule de fin facultative. Cependant, il existe une astuce standard pour prendre en charge les virgules de fin en ajoutant une règle supplémentaire. C'est ce que fait la troisième règle de notre macro : \$( ... ),\* vec !

```
($($x:expr),+ ,) => { // if trailing comma is present,
 vec![$($x),*] // retry without it
};
```

Nous avons l'habitude de faire correspondre une liste avec une virgule supplémentaire. Ensuite, dans le modèle, nous appelons récursivement, en laissant la virgule supplémentaire de côté. Cette fois, la deuxième règle correspondra. `$( ... ),+ , vec!`

## Macros intégrées

Le compilateur Rust fournit plusieurs macros qui sont utiles lorsque vous définissez vos propres macros. Aucun d'entre eux n'a pu être mis en œuvre en utilisant seul. Ils sont codés en dur en : `macro_rules! rustc`

`file!(), line!() column!()`

`file!()` se développe en un littéral de chaîne : le nom de fichier actuel. et développez jusqu'aux littéraux donnant la ligne et la colonne actuelles (en comptant à partir de 1). `line!() column!() u32`

Si une macro en appelle une autre, qui en appelle une autre, le tout dans des fichiers différents, et que la dernière macro appelle `, , ou ,` elle se développera pour indiquer l'emplacement du *premier* appel de macro. `file!() line!() column!()`

`stringify!(...tokens...)`

Se développe en un littéral de chaîne contenant les jetons donnés. La macro l'utilise pour générer un message d'erreur qui inclut le code de l'assertion. `assert!`

Les appels de macro dans l'argument *ne sont pas* développés : se développe à la chaîne `.stringify!(line!()) "line!()"`

Rust construit la chaîne à partir des jetons, de sorte qu'il n'y a pas de sauts de ligne ou de commentaires dans la chaîne.

`concat!(str0, str1, ...)`

Se développe à un littéral de chaîne unique créé en concaténant ses arguments.

Rust définit également ces macros pour interroger l'environnement de génération :

`cfg!(...)`

Se développe à une constante booléenne, si la configuration de build actuelle correspond à la condition entre parenthèses. Par exemple, est vrai si vous compilez avec des assertions de débogage activées. `true cfg!(debug_assertions)`

Cette macro prend en charge exactement la même syntaxe que l'attribut décrit dans « [Attributs](#) », mais au lieu de la compilation conditionnelle, vous obtenez une réponse vraie ou fausse. #`[cfg(...)]`

```
env!("VAR_NAME")
```

Se développe en une chaîne : valeur de la variable d'environnement spécifiée au moment de la compilation. Si la variable n'existe pas, il s'agit d'une erreur de compilation.

Cela ne vaudrait pas grand-chose, sauf que Cargo définit plusieurs variables d'environnement intéressantes lorsqu'il compile une caisse. Par exemple, pour obtenir la chaîne de version actuelle de votre caisse, vous pouvez écrire :

```
let version = env!("CARGO_PKG_VERSION");
```

Une liste complète de ces variables d'environnement est incluse dans la [documentation Cargo](#).

```
option_env!("VAR_NAME")
```

C'est la même chose que sauf qu'il renvoie un qui est si la variable spécifiée n'est pas définie. `env! Option<&'static str> None`

Trois autres macros intégrées vous permettent d'importer du code ou des données à partir d'un autre fichier :

```
include!("file.rs")
```

Développe le contenu du fichier spécifié, qui doit être du code Rust valide, soit une expression, soit une séquence [d'éléments](#).

```
include_str!("file.txt")
```

Se développe jusqu'à un contenant le texte du fichier spécifié. Vous pouvez l'utiliser comme ceci: `&'static str`

```
const COMPOSITOR_SHADER: &str =
 include_str!("../resources/compositor.glsl");
```

Si le fichier n'existe pas ou n'est pas valide UTF-8, vous obtiendrez une erreur de compilation.

```
include_bytes!("file.dat")
```

C'est la même chose, sauf que le fichier est traité comme des données binaires, et non comme du texte UTF-8. Le résultat est un fichier `.&'static [u8]`

Comme toutes les macros, celles-ci sont traitées au moment de la compilation. Si le fichier n'existe pas ou ne peut pas être lu, la compilation échoue. Ils ne peuvent pas échouer au moment de l'exécution. Dans tous les cas, si le nom de fichier est un chemin d'accès relatif, il est résolu par rapport au répertoire qui contient le fichier actif.

Rust fournit également plusieurs macros pratiques que nous n'avons pas couvertes auparavant:

```
todo!(), unimplemented!()
```

Ceux-ci sont équivalents à , mais transmettent une intention différente. va dans les clauses, les armes et d'autres cas qui ne sont pas encore traités. Il panique toujours. est à peu près la même chose, mais transmet l'idée que ce code n'a tout simplement pas encore été écrit; certains IDE le signalent pour avis. panic!

```
() unimplemented!() if match todo!()
```

```
matches!(value, pattern)
```

Compare une valeur à un modèle et renvoie si elle correspond ou non. C'est l'équivalent d'écrire : true false

```
match value {
 pattern => true,
 _ => false
}
```

Si vous recherchez un exercice d'écriture de macro de base, c'est une bonne macro à répliquer, d'autant plus que l'implémentation réelle, que vous pouvez voir dans la documentation standard de la bibliothèque, est assez simple.

## Débogage des macros

Le débogage d'une macro capricieuse peut être difficile. Le plus gros problème est le manque de visibilité sur le processus d'expansion macro. Rust va souvent développer toutes les macros, trouver une sorte d'erreur, puis imprimer un message d'erreur qui n'affiche pas le code entièrement développé qui contient l'erreur!

Voici trois outils pour vous aider à dépanner les macros. (Ces fonctionnalités sont toutes instables, mais comme elles sont vraiment conçues pour être utilisées pendant le développement, pas dans le code que vous archiveriez, ce n'est pas un gros problème dans la pratique.)

Tout d'abord, et le plus simple, vous pouvez demander à montrer à quoi ressemble votre code après avoir développé toutes les macros. Permet de

voir comment Cargo appelle . Copiez la ligne de commande et ajoutez-la en tant qu'options. Le code entièrement étendu est vidé sur votre terminal. Malheureusement, cela ne fonctionne que si votre code est exempt d'erreurs de syntaxe. `rustc cargo build --verbose rustc rustc -Z unstable-options --pretty expanded`

Deuxièmement, Rust fournit une macro qui imprime simplement ses arguments sur le terminal au moment de la compilation. Vous pouvez l'utiliser pour le débogage de style -. Cette macro nécessite l'indicateur de fonctionnalité. `log_syntax!() println! #![feature(log_syntax)]`

Troisièmement, vous pouvez demander au compilateur Rust d'enregistrer tous les appels de macro sur le terminal. Insérez quelque part dans votre code. À partir de ce moment, chaque fois que Rust développe une macro, il imprime le nom et les arguments de la macro. Par exemple, considérez ce programme : `trace_macros!(true);`

```
#![feature(trace_macros)]

fn main() {
 trace_macros!(true);
 let numbers = vec![1, 2, 3];
 trace_macros!(false);
 println!("total: {}", numbers.iter().sum::<u64>());
}
```

Il produit cette sortie:

```
$ rustup override set nightly
...
$ rustc trace_example.rs
note: trace_macro
--> trace_example.rs:5:19
|
5 | let numbers = vec![1, 2, 3];
| ^^^^^^
|
|= note: expanding `vec! { 1 , 2 , 3 }`
= note: to `< [_] > :: into_vec (box [1 , 2 , 3])`
```

Le compilateur affiche le code de chaque appel de macro, avant et après l'expansion. La ligne désactive à nouveau le traçage, de sorte que l'appel à n'est pas tracé. `trace_macros!(false); println!()`

## Construire le json! Macro

Nous avons maintenant discuté des fonctionnalités de base de . Dans cette section, nous allons développer de manière incrémentielle une macro pour créer des données JSON. Nous utiliserons cet exemple pour montrer ce que c'est que de développer une macro, présenter les quelques éléments restants de , et offrir quelques conseils sur la façon de s'assurer que vos macros se comportent comme souhaité. `macro_rules! macro_rules!`

Au [chapitre 10](#), nous avons présenté cet énumérateur pour représenter les données JSON :

```
#[derive(Clone, PartialEq, Debug)]
enum Json {
 Null,
 Boolean(bool),
 Number(f64),
 String(String),
 Array(Vec<Json>),
 Object(Box<HashMap<String, Json>>)
}
```

La syntaxe d'écriture des valeurs est malheureusement plutôt verbeuse

:  
:Json

```
let students = Json::Array(vec![
 Json::Object(Box::new(vec![
 ("name".to_string(), Json::String("Jim Blandy".to_string())),
 ("class_of".to_string(), Json::Number(1926.0)),
 ("major".to_string(), Json::String("Tibetan throat singing".to_string()))
].into_iter().collect())),
 Json::Object(Box::new(vec![
 ("name".to_string(), Json::String("Jason Orendorff".to_string())),
 ("class_of".to_string(), Json::Number(1702.0)),
 ("major".to_string(), Json::String("Knots".to_string()))
].into_iter().collect()))
]);

```

Nous aimerais pouvoir écrire ceci en utilisant une syntaxe plus JSON:

```
let students = json!([
 {
 "name": "Jim Blandy",
 "class_of": 1926,
 "major": "Tibetan throat singing"
 },
 {

```

```

 "name": "Jason Orendorff",
 "class_of": 1702,
 "major": "Knots"
 }
];


```

Ce que nous voulons, c'est une macro qui prend une valeur JSON comme argument et se développe à une expression Rust comme celle de l'exemple précédent. json!

## Types de fragments

Le premier travail d'écriture d'une macro complexe consiste à déterminer comment faire *correspondre ou analyser* l'entrée souhaitée.

Nous pouvons déjà voir que la macro aura plusieurs règles, car il y a plusieurs sortes de choses différentes dans les données JSON : objets, tableaux, nombres, etc. En fait, nous pourrions deviner que nous aurons une règle pour chaque type JSON :

```

macro_rules! json {
 (null) => { Json::Null };
 ([...]) => { Json::Array(...) };
 ({ ... }) => { Json::Object(...) };
 (???) => { Json::Boolean(...) };
 (???) => { Json::Number(...) };
 (???) => { Json::String(...) };
}

```

Ce n'est pas tout à fait correct, car les modèles macro n'offrent aucun moyen de séparer les trois derniers cas, mais nous verrons comment y faire face plus tard. Les trois premiers cas, au moins, commencent clairement par des jetons différents, alors commençons par ceux-ci.

La première règle fonctionne déjà :

```

macro_rules! json {
 (null) => {
 Json::Null
 }
}

#[test]
fn json_null() {
 assert_eq!(json!(null), Json::Null); // passes!
}

```

Pour ajouter la prise en charge des tableaux JSON, nous pouvons essayer de faire correspondre les éléments en tant que `s : expr`

```
macro_rules! json {
 (null) => {
 Json::Null
 };
 ([$($element:expr),*]) => {
 Json::Array(vec![$($element),*])
 };
}
```

Malheureusement, cela ne correspond pas à tous les tableaux JSON. Voici un test qui illustre le problème :

```
#[test]
fn json_array_with_json_element() {
 let macro_generated_value = json!(
 [
 // valid JSON that doesn't match `{$element:expr}`
 {
 "pitch": 440.0
 }
]
);
 let hand_coded_value =
 Json::Array(vec![
 Json::Object(Box::new(vec![
 ("pitch".to_string(), Json::Number(440.0))
].into_iter().collect()))
]);
 assert_eq!(macro_generated_value, hand_coded_value);
}
```

Le motif signifie « une liste d'expressions Rust séparées par des virgules ». Mais de nombreuses valeurs JSON, en particulier les objets, ne sont pas des expressions Rust valides. Ils ne correspondront pas. `$( $element:expr ),*`

Étant donné que tous les bits de code que vous souhaitez faire correspondre ne sont pas une expression, Rust prend en charge plusieurs autres types de fragments, répertoriés dans [le tableau 21-2](#).

Tableau 21-2. Types de fragments pris en charge par `macro_rules!`

| Type de fragment | Correspondances (avec exemples)                                                                                                          | Peut être suivi de...                                          |
|------------------|------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------|
| expr             | Une expression :<br><code>2 + 2,, "udon" x.len()</code>                                                                                  | <code>=&gt; , ;</code>                                         |
| stmt             | Expression ou déclaration, à l'exception d'un point-virgule de fin (difficile à utiliser; essayez ou à la place) <code>expr block</code> | <code>=&gt; , ;</code>                                         |
| ty               | Un type :<br><code>String,,, Vec&lt;u8&gt; (&amp;str, bool) dyn Read + Send</code>                                                       | <code>=&gt; , ; =   { [</code><br><code>: &gt; as where</code> |
| path             | Un chemin (discuté) :<br><code>ferns, ::std::sync::mpsc</code>                                                                           | <code>=&gt; , ; =   { [</code><br><code>: &gt; as where</code> |
| pat              | Un modèle (discuté) :<br><code>_ , Some(ref x)</code>                                                                                    | <code>=&gt; , =   if in</code>                                 |
| item             | Un point (discuté) :<br><code>struct Point { x: f64, y: f64 }, mod ferns;</code>                                                         | Rien                                                           |
| block            | Un bloc (discuté) :<br><code>{ s += "ok\n"; true }</code>                                                                                | Rien                                                           |
| meta             | Le corps d'un attribut (discuté) :<br><code>inline,, derive(Copy, Clone) doc="3D models."</code>                                         | Rien                                                           |
| literal          | Valeur littérale :<br><code>1024,, "Hello, world!" 1_000_000f64</code>                                                                   | Rien                                                           |
| lifetim e        | Une vie :<br><code>'a,, 'item 'static</code>                                                                                             | Rien                                                           |

| Type de fragment | Correspondances (avec exemples)                                                | Peut être suivi de... |
|------------------|--------------------------------------------------------------------------------|-----------------------|
| vis              | Un spécificateur de visibilité :<br>pub,, pub(crate) pub(in module::submodule) | Rien                  |
| ident            | Un identifiant :<br>std,, Json longish_variable_name                           | Rien                  |
| tt               | Une arborescence de jetons (voir texte) :<br>;,,, >= {} [ 0 1 (+ 0 1)]         | Rien                  |

La plupart des options de ce tableau appliquent strictement la syntaxe Rust. Le type correspond uniquement aux expressions Rust (pas aux valeurs JSON), ne correspond qu'aux types Rust, etc. Ils ne sont pas extensibles : il n'y a aucun moyen de définir de nouveaux opérateurs arithmétiques ou de nouveaux mots-clés qui reconnaîtraient. Nous ne serons pas en mesure de faire correspondre l'une de ces données JSON arbitraires. `expr ty expr`

Les deux derniers, et , prennent en charge les arguments de macro correspondants qui ne ressemblent pas au code Rust. correspond à n'importe quel identificateur. correspond à un seul *arbre de jetons* : soit une paire de crochets correctement assortie, , ou , et tout ce qui se trouve entre les deux, y compris les arbres de jetons imbriqués, soit un seul jeton qui n'est pas un crochet, comme ou . ident tt ident tt (...) [...] { ... } 1926 "Knots"

Les arbres à jetons sont exactement ce dont nous avons besoin pour notre macro. Chaque valeur JSON est une arborescence de jeton unique : nombres, chaînes, valeurs booléennes et sont tous des jetons uniques ; les objets et les tableaux sont entre crochets. Nous pouvons donc écrire les modèles comme ceci: `json! null`

```
macro_rules! json {
 (null) => {
 Json::Null
 };
 ([$($element:tt),*]) => {
 Json::Array(...)
 };
}
```

```

 };
 ({ $($key:tt : $value:tt),* }) => {
 Json::Object(...)

 };
 ($other:tt) => {
 ... // TODO: Return Number, String, or Boolean
 };
}

```

Cette version de la macro peut correspondre à toutes les données JSON. Maintenant, il ne nous reste plus qu'à produire le code Rust correct. json!

Pour s'assurer que Rust peut acquérir de nouvelles fonctionnalités syntaxiques à l'avenir sans casser les macros que vous écrivez aujourd'hui, Rust restreint les jetons qui apparaissent dans des modèles juste après un fragment. Le « Peut être suivi de... » du [tableau 21-2](#) indique quels jetons sont autorisés. Par exemple, le modèle est une erreur, car il n'est pas autorisé après un fichier . Le motif est OK, car il est suivi de la flèche , l'un des jetons autorisés pour un , et est suivi de rien, ce qui est toujours autorisé.

```
$x:expr ~ $y:expr ~ expr $vars:pat =>
$handler:expr $vars:pat => pat $handler:expr
```

## Récursivité dans les macros

Vous avez déjà vu un cas trivial d'une macro s'appelant elle-même : notre implémentation utilise la récursivité pour prendre en charge les virgules de fin. Ici, nous pouvons montrer un exemple plus significatif: doit s'appeler de manière récursive. `vec! json!`

Nous pourrions essayer de prendre en charge les baies JSON sans utiliser la récursivité, comme ceci :

```

([$($element:tt),*]) => {
 Json::Array(vec![$($element),*])
};

```

Mais cela ne fonctionnerait pas. Nous collerions des données JSON (les arbres de jetons) directement dans une expression Rust. Ce sont deux langues différentes. `$element`

Nous devons convertir chaque élément du tableau de la forme JSON en Rust. Heureusement, il y a une macro qui fait ça : celle que nous écrivons !

```
([$($element:tt),*]) => {
 Json::Array(vec![$(json!($element)),*])
};
```

Les objets peuvent être pris en charge de la même manière :

```
({ $($key:tt : $value:tt),* }) => {
 Json::Object(Box::new(vec![
 $(($key.to_string(), json!($value))),*
].into_iter().collect())))
};
```

Le compilateur impose une limite de récursivité aux macros : 64 appels, par défaut. C'est plus que suffisant pour les utilisations normales de , mais les macros récursives complexes atteignent parfois la limite. Vous pouvez l'ajuster en ajoutant cet attribut en haut de la caisse où la macro est utilisée : `json!`

```
#![recursion_limit = "256"]
```

Notre macro est presque terminée. Il ne reste plus qu'à prendre en charge les valeurs booléennes, numériques et de chaîne. `json!`

## Utilisation de traits avec des macros

Écrire des macros complexes pose toujours des énigmes. Il est important de se rappeler que les macros elles-mêmes ne sont pas le seul outil de résolution d'énigmes à votre disposition.

Ici, nous devons soutenir , et , convertir la valeur, quelle qu'elle soit, en un type de valeur approprié. Mais les macros ne sont pas bonnes pour distinguer les types. On peut imaginer écrire : `json!(true)` `json!`

```
(1.0) json!("yes") Json
```

```
macro_rules! json {
 (true) => {
 Json::Boolean(true)
 };
 (false) => {
 Json::Boolean(false)
 };
 ...
}
```

Cette approche s'effondre tout de suite. Il n'y a que deux valeurs booléennes, mais plutôt plus de nombres que cela, et encore plus de chaînes.

Heureusement, il existe un moyen standard de convertir des valeurs de différents types en un type spécifié: le trait, couvert . Nous devons simplement implémenter ce trait pour quelques types: From

```
impl From<bool> for Json {
 fn from(b: bool) -> Json {
 Json::Boolean(b)
 }
}

impl From<i32> for Json {
 fn from(i: i32) -> Json {
 Json::Number(i as f64)
 }
}

impl From<String> for Json {
 fn from(s: String) -> Json {
 Json::String(s)
 }
}

impl<'a> From<&'a str> for Json {
 fn from(s: &'a str) -> Json {
 Json::String(s.to_string())
 }
}

...
```

En fait, les 12 types numériques devraient avoir des implémentations très similaires, il pourrait donc être logique d'écrire une macro, juste pour éviter le copier-coller:

```
macro_rules! impl_from_num_for_json {
 ($($t:ident)*) => {
 $($(
 impl From<$t> for Json {
 fn from(n: $t) -> Json {
 Json::Number(n as f64)
 }
 }
)*
 };
}
```

```

 }

impl_from_num_for_json!(u8 i8 u16 i16 u32 i32 u64 i64 u128 i128
 usize isize f32 f64);

```

Maintenant, nous pouvons utiliser pour convertir un de n'importe quel type pris en charge en . Dans notre macro, cela ressemblera à ceci: `Json::from(value)` value `Json`

```

($other:tt) => {
 Json::from($other) // Handle Boolean/number/string
};

```

L'ajout de cette règle à notre macro lui permet de réussir tous les tests que nous avons écrits jusqu'à présent. En rassemblant toutes les pièces, il ressemble actuellement à ceci: `json!`

```

macro_rules! json {
 (null) => {
 Json::Null
 };
 ([$($element:tt),*]) => {
 Json::Array(vec![$(json!($element)),*])
 };
 ({ $($key:tt : $value:tt),* }) => {
 Json::Object(Box::new(vec![
 $($key.to_string(), json!($value)),*
].into_iter().collect())))
 };
 ($other:tt) => {
 Json::from($other) // Handle Boolean/number/string
 };
}

```

Il s'avère que la macro prend en charge de manière inattendue l'utilisation de variables et même d'expressions Rust arbitraires dans les données JSON, une fonctionnalité supplémentaire pratique:

```

let width = 4.0;
let desc =
 json!({
 "width": width,
 "height": (width * 9.0 / 4.0)
 });

```

Parce qu'il s'agit d'une arborescence de jetons unique, la macro lui correspond donc avec succès lors de l'analyse de l'objet. (width \* 9.0 / 4.0) \$value:tt

## Cadrage et hygiène

Un aspect étonnamment délicat de l'écriture de macros est qu'elles impliquent de coller du code de différentes étendues ensemble. Ainsi, les pages suivantes couvrent les deux façons dont Rust gère la portée: une façon pour les variables et les arguments locaux, et une autre façon pour tout le reste.

Pour montrer pourquoi cela est important, réécrivons notre règle d'analyse des objets JSON (la troisième règle de la macro montrée précédemment) pour éliminer le vecteur temporaire. Nous pouvons l'écrire comme ceci: json!

```
({ $($key:tt : $value:tt),* }) => {
{
 let mut fields = Box::new(HashMap::new());
 $($fields.insert($key.to_string(), json!($value));)*
 Json::Object(fields)
}
};
```

Maintenant, nous remplissons le non pas en utilisant mais en appelant à plusieurs reprises la méthode. Cela signifie que nous devons stocker la carte dans une variable temporaire, que nous avons appelée .HashMap collect() .insert() fields

Mais alors que se passe-t-il si le code qui appelle utilise une variable qui lui est propre, également nommée ? json! fields

```
let fields = "Fields, W.C.";
let role = json!({
 "name": "Larson E. Whipsnade",
 "actor": fields
});
```

Développer la macro collerait ensemble deux bits de code, les deux en utilisant le nom pour des choses différentes! fields

```
let fields = "Fields, W.C.";
let role = {
 let mut fields = Box::new(HashMap::new());
 fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
}
```

```
 fields.insert("actor".to_string(), Json::from(fields));
Json::Object(fields)
};
```

Cela peut sembler un piège inévitable chaque fois que les macros utilisent des variables temporaires, et vous réfléchissez peut-être déjà aux correctifs possibles. Peut-être devrions-nous renommer la variable que la macro définit en quelque chose que ses appelants ne sont pas susceptibles de transmettre: au lieu de `fields`, nous pourrions l'appeler

```
.json! fields __json$fields
```

La surprise ici est que *la macro fonctionne telle quelle*. Rust renomme la variable pour vous! Cette fonctionnalité, d'abord implémentée dans les macros Scheme, s'appelle *hygiène*, et on dit donc que Rust a des *macros hygiéniques*.

La façon la plus simple de comprendre l'hygiène macro est d'imaginer que chaque fois qu'une macro est développée, les parties de l'expansion qui proviennent de la macro elle-même sont peintes d'une couleur différente.

Les variables de différentes couleurs sont alors traitées comme si elles avaient des noms différents :

```
let fields = "Fields, W.C.";
let role = {
 let mut fields = Box::new(HashMap::new());
 fields.insert("name".to_string(), Json::from("Larson E. Whipsnade"));
 fields.insert("actor".to_string(), Json::from(fields));
Json::Object(fields)
};
```

Notez que les bits de code qui ont été transmis par l'appelant de macro et collés dans la sortie, tels que `fields` et `Json::Object(fields)`, conservent leur couleur d'origine (noir). Seuls les jetons provenant du modèle de macro sont peints. `"name"` et `"actor"`

Maintenant, il y a une variable nommée (déclarée dans l'appelant) et une variable distincte nommée (introduite par la macro). Comme les noms sont de couleurs différentes, les deux variables ne sont pas confondues. `fields` et `fields`

Si une macro a vraiment besoin de faire référence à une variable dans la portée de l'appelant, l'appelant doit transmettre le nom de la variable à la macro.

(La métaphore de la peinture n'est pas censée être une description exacte du fonctionnement de l'hygiène. Le mécanisme réel est même un peu plus intelligent que cela, reconnaissant deux identificateurs comme identiques, indépendamment de la « peinture », s'ils se réfèrent à une variable commune qui est à la fois dans la portée de la macro et de son appelant. Mais des cas comme celui-ci sont rares à Rust. Si vous comprenez l'exemple précédent, vous en savez assez pour utiliser des macros hygiéniques.)

Vous avez peut-être remarqué que de nombreux autres identificateurs ont été peints d'une ou plusieurs couleurs au fur et à mesure que les macros étaient développées : , et , par exemple. Malgré la peinture, Rust n'a eu aucun mal à reconnaître ces noms de type. C'est parce que l'hygiène dans Rust est limitée aux variables et arguments locaux. En ce qui concerne les constantes, les types, les méthodes, les modules, la statique et les noms de macros, Rust est « daltonien ». Box HashMap Json

Cela signifie que si notre macro est utilisée dans un module où , , ou n'est pas dans la portée, la macro ne fonctionnera pas. Nous montrerons comment éviter ce problème dans la section suivante. json! Box HashMap Json

Tout d'abord, nous examinerons un cas où l'hygiène stricte de Rust nous gêne, et nous devons le contourner. Supposons que nous ayons de nombreuses fonctions qui contiennent cette ligne de code :

```
let req = ServerRequest::new(server_socket.session());
```

Copier et coller cette ligne est pénible. Peut-on utiliser une macro à la place ?

```
macro_rules! setup_req {
 () => {
 let req = ServerRequest::new(server_socket.session());
 }
}

fn handle_http_request(server_socket: &ServerSocket) {
 setup_req!(); // declares `req`, uses `server_socket`
 ... // code that uses `req`
}
```

Tel qu'il est écrit, cela ne fonctionne pas. Il faudrait que le nom dans la macro fasse référence au local déclaré dans la fonction, et vice versa pour la variable . Mais l'hygiène empêche les noms dans les macros de « se heurter » avec des noms dans d'autres étendues, même dans des cas

comme celui-ci, où c'est ce que vous voulez.

```
server_socket server_socket req
```

La solution consiste à transmettre à la macro tous les identificateurs que vous prévoyez d'utiliser à l'intérieur et à l'extérieur du code de macro :

```
macro_rules! setup_req {
 ($req:ident, $server_socket:ident) => {
 let $req = ServerRequest::new($server_socket.session());
 }
}

fn handle_http_request(server_socket: &ServerSocket) {
 setup_req!(req, server_socket);
 ... // code that uses `req`
}
```

Depuis et sont maintenant fournis par la fonction, ils sont la bonne « couleur » pour cette portée. `req` `server_socket`

L'hygiène rend cette macro un peu plus verbeuse à utiliser, mais c'est une fonctionnalité, pas un bug: il est plus facile de raisonner sur les macros hygiéniques en sachant qu'elles ne peuvent pas jouer avec les variables locales derrière votre dos. Si vous recherchez un identifiant comme dans une fonction, vous trouverez tous les endroits où il est utilisé, y compris les appels de macro. `server_socket`

## Importation et exportation de macros

Étant donné que les macros sont développées au début de la compilation, avant que Rust ne connaisse la structure complète du module de votre projet, le compilateur dispose d'affordances spéciales pour les exporter et les importer.

Les macros visibles dans un module sont automatiquement visibles dans ses modules enfants. Pour exporter des macros d'un module « vers le haut » vers son module parent, utilisez l'attribut. Par exemple, supposons que notre `lib.rs` ressemble à ceci : `#[macro_use]`

```
#[macro_use] mod macros;
mod client;
mod server;
```

Toutes les macros définies dans le module sont importées dans `lib.rs` et donc visibles dans le reste de la caisse, y compris dans et

```
.macros client server
```

Les macros marquées par sont automatiquement et peuvent être référencées par chemin, comme d'autres éléments. #

```
[macro_export] pub
```

Par exemple, la caisse fournit une macro appelée , qui est marquée par . Pour utiliser cette macro dans votre propre caisse, vous devez écrire

```
:lazy_static lazy_static #[macro_export]
```

```
use lazy_static::lazy_static;
lazy_static!{ }
```

Une fois qu'une macro est importée, elle peut être utilisée comme n'importe quel autre élément :

```
use lazy_static::lazy_static;

mod m {
 crate::lazy_static!{ }
}
```

Bien sûr, faire l'une de ces choses signifie que votre macro peut être appelée dans d'autres modules. Une macro exportée ne doit donc pas dépendre de quoi que ce soit dans la portée - il est impossible de dire ce qui sera dans la portée où elle est utilisée. Même les caractéristiques du prélude standard peuvent être ombragées.

Au lieu de cela, la macro doit utiliser des chemins absolus vers tous les noms qu'elle utilise. fournit le fragment spécial pour aider à cela. Ce n'est pas la même chose que , qui est un mot-clé qui peut être utilisé dans les chemins n'importe où, pas seulement dans les macros. agit comme un chemin absolu vers le module racine de la caisse où la macro a été définie. Au lieu de dire , on peut écrire , ce qui fonctionne même si `Json` n'a pas été importé. peut être remplacé par ou . Dans ce dernier cas, nous devrons réexporter , car ne peut pas être utilisé pour accéder aux fonctionnalités privées d'une caisse. Il s'étend vraiment à quelque chose comme , un chemin ordinaire. Les règles de visibilité ne sont pas affectées.

```
macro_rules! $crate crate $crate Json $crate::Json Hash
Map ::std::collections::HashMap $crate::macros::HashMap Has
hMap $crate ::jsonlib
```

Après avoir déplacé la macro vers son propre module et l'avoir modifiée pour l'utiliser , elle ressemble à ceci. Voici la version finale

```
:macros $crate
```

```

// macros.rs

pub use std::collections::HashMap;
pub use std::boxed::Box;
pub use std::string::ToString;

#[macro_export]
macro_rules! json {
 (null) => {
 $crate::Json::Null
 };
 ([$($element:tt),*]) => {
 $crate::Json::Array(vec![$(json!($element)),*])
 };
 ({ $($key:tt : $value:tt),* }) => {
 {
 let mut fields = $crate::macros::Box::new(
 $crate::macros::HashMap::new());
 $($
 fields.insert($crate::macros::ToString::to_string($key),
 json!($value)));
)*
 $crate::Json::Object(fields)
 }
 };
 ($other:tt) => {
 $crate::Json::from($other)
 };
}

```

Étant donné que la méthode fait partie du trait standard, nous utilisons également pour nous y référer, en utilisant la syntaxe que nous avons introduite dans [« Appels de méthode complets »](#). Dans notre cas, ce n'est pas strictement nécessaire pour que la macro fonctionne, car c'est dans le prélude standard. Mais si vous appelez des méthodes d'un trait qui n'est peut-être pas dans la portée au point où la macro est appelée, un appel de méthode complet est le meilleur moyen de le faire.

```
.to_string() ToString $crate $crate::macros::ToString:
:to_string($key) ToString
```

## Éviter les erreurs de syntaxe lors de la correspondance

La macro suivante semble raisonnable, mais elle pose quelques problèmes à Rust :

```

macro_rules! complain {
 ($msg:expr) => {
 println!("Complaint filed: {}", $msg)
 };
 ($user : $userid:tt , $msg:expr) => {
 println!("Complaint from user {}: {}", $userid, $msg)
 };
}

```

Supposons que nous l'appelions comme ceci:

```
complain!(user: "jimb", "the AI lab's chatbots keep picking on me");
```

Pour les yeux humains, cela correspond évidemment au deuxième modèle. Mais Rust essaie d'abord la première règle, en essayant de faire correspondre toutes les entrées avec . C'est là que les choses commencent à mal tourner pour nous. n'est pas une expression, bien sûr, donc nous obtenons une erreur de syntaxe. Rust refuse de balayer une erreur de syntaxe sous le tapis - les macros sont déjà assez difficiles à déboguer. Au lieu de cela, il est signalé immédiatement et la compilation s'arrête. \$msg:expr user: "jimb"

Si un autre jeton d'un modèle ne correspond pas, Rust passe à la règle suivante. Seules les erreurs de syntaxe sont fatales, et elles ne se produisent que lorsque vous essayez de faire correspondre des fragments.

Le problème ici n'est pas si difficile à comprendre: nous essayons de faire correspondre un fragment, , dans la mauvaise règle. Ça ne va pas correspondre parce que nous ne sommes même pas censés être ici. L'appelant voulait l'autre règle. Il existe deux façons simples d'éviter cela. \$msg:expr

Tout d'abord, évitez les règles confuses. Nous pourrions, par exemple, modifier la macro afin que chaque modèle commence par un identificateur différent :

```

macro_rules! complain {
 ($msg : $msg:expr) => {
 println!("Complaint filed: {}", $msg);
 };
 ($user : $userid:tt , $msg : $msg:expr) => {
 println!("Complaint from user {}: {}", $userid, $msg);
 };
}

```

Lorsque les arguments de macro commencent par , nous obtenons la règle 1. Quand ils commenceront par , nous obtiendrons la règle 2. Quoi qu'il en soit, nous savons que nous avons la bonne règle avant d'essayer de faire correspondre un fragment. `msg user`

L'autre façon d'éviter les erreurs de syntaxe fallacieuses est de mettre en premier des règles plus spécifiques. Mettre la règle en premier résout le problème avec , car la règle qui provoque l'erreur de syntaxe n'est jamais atteinte. `user: complain!`

## Au-delà de macro\_rules!

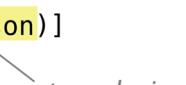
Les modèles de macro peuvent analyser des entrées encore plus complexes que JSON, mais nous avons constaté que la complexité devient rapidement incontrôlable.

[The Little Book of Rust Macros](#), par Daniel Keep et al., est un excellent manuel de programmation avancée. Le livre est clair et intelligent, et il décrit tous les aspects de l'expansion macro plus en détail que nous ne l'avons ici. Il présente également plusieurs techniques très intelligentes pour presser des motifs en service comme une sorte de langage de programmation ésotérique, pour analyser des entrées complexes. Nous sommes moins enthousiastes à ce sujet. Utiliser avec précaution. `macro_rules! macro_rules!`

Rust 1.15 a introduit un mécanisme distinct appelé *macros procédurales*. Les macros procédurales prennent en charge l'extension de l'attribut pour gérer les dérivations personnalisées, comme illustré à [la figure 21-4](#), ainsi que la création d'attributs personnalisés et de nouvelles macros appelées comme les macros décrites précédemment. #

```
[derive] macro_rules!
```

```
#[derive(Copy, Clone, PartialEq, Eq, IntoJson)]
struct Money {
 dollars: u32,
 cents: u16,
}
```



Graphique 21-4. Appel d'une macro procédurale hypothétique via un attribut `IntoJson` #  
[derive]

Il n'y a pas de trait, mais ce n'est pas grave : une macro procédurale peut utiliser ce hook pour insérer le code qu'elle veut (dans ce cas, probablement). `IntoJson impl From<Money> for Json { ... }`

Ce qui rend une macro procédurale « procédurale », c'est qu'elle est implémentée en tant que fonction Rust, et non en tant qu'ensemble de rè-

gles déclaratives. Cette fonction interagit avec le compilateur à travers une fine couche d'abstraction et peut être arbitrairement complexe. Par exemple, la bibliothèque de base de données utilise des macros procédurales pour se connecter à une base de données et générer du code basé sur le schéma de cette base de données au moment de la compilation. `diesel`

Étant donné que les macros procédurales interagissent avec les internes du compilateur, l'écriture de macros efficaces nécessite une compréhension du fonctionnement du compilateur qui est hors de la portée de ce livre. Il est cependant largement couvert dans la [documentation en ligne](#).

Peut-être, après avoir lu tout cela, avez-vous décidé que vous détestez les macros. Et alors ? Une alternative consiste à générer du code Rust à l'aide d'un script de build. La [documentation Cargo](#) montre comment le faire étape par étape. Cela implique d'écrire un programme qui génère le code Rust que vous voulez, d'ajouter une ligne à `Cargo.toml` pour exécuter ce programme dans le cadre du processus de génération et d'utiliser pour obtenir le code généré dans votre caisse. `include!`

[Soutien](#)    [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)

# Chapitre 22. Code dangereux

*Que personne ne pense à moi que je suis humble, faible ou passif;  
Faites-leur comprendre que je suis d'un autre genre :  
dangereux pour mes ennemis, loyal envers mes amis.  
À une telle vie appartient la gloire.*

—Euripide, *Médée*

La joie secrète de la programmation de systèmes est que, sous chaque langage sûr et abstraction soigneusement conçue se trouve un maelström tourbillonnant de langage machine extrêmement dangereux et de bricolage de bits. Vous pouvez également écrire cela dans Rust.

Le langage que nous avons présenté jusqu'à présent dans le livre garantit que vos programmes sont exempts d'erreurs de mémoire et de courses de données entièrement automatiques, grâce aux types, aux durées de vie, aux vérifications des limites, etc. Mais ce genre de raisonnement automatisé a ses limites; il existe de nombreuses techniques précieuses que Rust ne peut pas reconnaître comme sûres.

*Le code dangereux* vous permet de dire à Rust : « J'opte pour l'utilisation de fonctionnalités dont vous ne pouvez pas garantir la sécurité. » En marquant un bloc ou une fonction comme dangereux, vous acquérez la possibilité d'appeler des fonctions dans la bibliothèque standard, de déréférencer des pointeurs dangereux et d'appeler des fonctions écrites dans d'autres langages comme C et C++, entre autres pouvoirs. Les autres contrôles de sécurité de Rust s'appliquent toujours: les contrôles de type, les contrôles de durée de vie et les contrôles de limites sur les indices se produisent tous normalement. Le code dangereux permet simplement un petit ensemble de fonctionnalités supplémentaires. `unsafe`

Cette capacité à sortir des limites de Rust en toute sécurité est ce qui permet d'implémenter bon nombre des fonctionnalités les plus fondamentales de Rust dans Rust lui-même, tout comme C et C++ sont utilisés pour implémenter leurs propres bibliothèques standard. Le code dangereux est ce qui permet au type de gérer efficacement sa mémoire tampon; le module pour parler au système d'exploitation; et les modules et pour fournir des primitives d'accès concurrentiel. `vec std::io std::thread std::sync`

Ce chapitre couvre l'essentiel de l'utilisation de fonctionnalités dangereuses :

- Les blocs de Rust établissent la frontière entre le code Rust ordinaire et sûr et le code qui utilise des fonctionnalités dangereuses. `unsafe`
- Vous pouvez marquer les fonctions comme , alertant les appelants de la présence de contrats supplémentaires qu'ils doivent suivre pour éviter un comportement indéfini. `unsafe`
- Les pointeurs bruts et leurs méthodes permettent un accès sans contrainte à la mémoire et vous permettent de créer des structures de données que le système de type Rust interdirait autrement. Alors que les références de Rust sont sûres mais contraintes, les pointeurs bruts, comme tout programmeur C ou C++ le sait, sont un outil puissant et pointu.
- Comprendre la définition d'un comportement indéfini vous aidera à comprendre pourquoi il peut avoir des conséquences beaucoup plus graves que le simple fait d'obtenir des résultats incorrects.
- Les caractéristiques dangereuses, analogues aux fonctions, imposent un contrat que chaque implémentation (plutôt que chaque appelant) doit suivre. `unsafe`

## Dangereux de quoi?

Au début de ce livre, nous avons montré un programme C qui se bloque de manière surprenante car il ne suit pas l'une des règles prescrites par la norme C. Vous pouvez faire la même chose dans Rust:

```
$ cat crash.rs
fn main() {
 let mut a: usize = 0;
 let ptr = &mut a as *mut usize;
 unsafe {
 *ptr.offset(3) = 0x7ffff72f484c;
 }
}
$ cargo build
Compiling unsafe-samples v0.1.0
 Finished debug [unoptimized + debuginfo] target(s) in 0.44s
$../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
```

```
Segmentation fault (core dumped)
```

```
$
```

Ce programme emprunte une référence modifiable à la variable locale, la transforme en pointeur brut de type , puis utilise la méthode pour produire un pointeur trois mots plus loin dans la mémoire. Il se trouve que c'est là que l'adresse de retour de 's est stockée. Le programme écrase l'adresse de retour par une constante, de sorte que le retour de se comporte de manière surprenante. Ce qui rend ce crash possible, c'est l'utilisation incorrecte par le programme de fonctionnalités dangereuses , dans ce cas, la possibilité de déréférencer les pointeurs bruts. a \*mut  
usize offset main main

Une fonctionnalité dangereuse est celle qui impose un *contrat* : des règles que Rust ne peut pas appliquer automatiquement, mais que vous devez néanmoins suivre pour éviter *un comportement indéfini*.

Un contrat va au-delà des contrôles de type et des contrôles de durée de vie habituels, imposant des règles supplémentaires spécifiques à cette caractéristique dangereuse. En règle générale, Rust lui-même ne connaît pas du tout le contrat; c'est juste expliqué dans la documentation de la fonctionnalité. Par exemple, le type de pointeur brut a un contrat vous interdisant de déréférencer un pointeur qui a été avancé au-delà de la fin de son référent d'origine. L'expression de cet exemple rompt ce contrat. Mais, comme le montre la transcription, Rust compile le programme sans se plaindre : ses contrôles de sécurité ne détectent pas cette violation. Lorsque vous utilisez des fonctionnalités dangereuses, vous, en tant que programmeur, assumez la responsabilité de vérifier que votre code respecte leurs contrats. `*ptr.offset(3) = ...`

De nombreuses fonctionnalités ont des règles que vous devez suivre pour les utiliser correctement, mais ces règles ne sont pas des contrats au sens où nous le voulons dire ici, sauf si les conséquences possibles incluent un comportement indéfini. Un comportement indéfini est un comportement que Rust suppose fermement que votre code ne pourrait jamais présenter. Par exemple, Rust suppose que vous n'écraserez pas l'adresse de retour d'un appel de fonction par autre chose. Le code qui passe les contrôles de sécurité habituels de Rust et se conforme aux contrats des fonctionnalités dangereuses qu'il utilise ne peut pas faire une telle chose. Étant donné que le programme viole le contrat de pointeur brut, son comportement n'est pas défini et il déraille.

Si votre code présente un comportement indéfini, vous avez rompu la moitié de votre contrat avec Rust, et Rust refuse d'en prédire les conséquences. Le dragage des messages d'erreur non pertinents des profondeurs des bibliothèques système et le plantage sont une conséquence possible; confier le contrôle de votre ordinateur à un attaquant en est une autre. Les effets peuvent varier d'une version de Rust à l'autre, sans avertissement. Parfois, cependant, un comportement indéfini n'a pas de conséquences visibles. Par exemple, si la fonction ne revient jamais (peut-être qu'elle appelle à mettre fin au programme plus tôt), l'adresse de retour corrompue n'aura probablement pas d'importance. `main std::process::exit`

Vous ne pouvez utiliser que des fonctionnalités dangereuses au sein d'un bloc ou d'une fonction ; nous expliquerons les deux dans les sections qui suivent. Cela rend plus difficile l'utilisation de fonctionnalités dangereuses sans le savoir : en vous forçant à écrire un bloc ou une fonction, Rust s'assure que vous avez reconnu que votre code peut avoir des règles supplémentaires à suivre. `unsafe unsafe unsafe`

## Blocs dangereux

Un bloc ressemble à un bloc Rust ordinaire précédé du mot-clé, à la différence que vous pouvez utiliser des fonctionnalités dangereuses dans le bloc: `unsafe unsafe`

```
unsafe {
 String::from_utf8_unchecked(ascii)
}
```

Sans le mot-clé devant le bloc, Rust s'opposerait à l'utilisation de , qui est une fonction. Avec le bloc autour, vous pouvez utiliser ce code n'importe où. `unsafe from_utf8_unchecked unsafe unsafe`

Comme un bloc Rust ordinaire, la valeur d'un bloc est celle de son expression finale, ou s'il n'en a pas. L'appel à affiché précédemment fournit la valeur du bloc. `unsafe () String::from_utf8_unchecked`

Un bloc déverrouille cinq options supplémentaires pour vous : `unsafe`

- Vous pouvez appeler des fonctions. Chaque fonction doit spécifier son propre contrat, en fonction de son objet. `unsafe unsafe`

- Vous pouvez déréférencer les pointeurs bruts. Le code sécurisé peut passer des pointeurs bruts, les comparer et les créer par conversion à partir de références (ou même d'entiers), mais seul le code non sécurisé peut réellement les utiliser pour accéder à la mémoire. Nous couvrirons les pointeurs bruts en détail et expliquerons comment les utiliser en toute sécurité dans [« Pointeurs bruts »](#).
- Vous pouvez accéder aux champs de `s`, dont le compilateur ne peut pas être sûr qu'ils contiennent des modèles de bits valides pour leurs types respectifs. `union`
- Vous pouvez accéder à des variables modifiables. Comme expliqué dans [« Variables globales »](#), Rust ne peut pas être sûr lorsque les threads utilisent des variables mutables, de sorte que leur contrat vous oblige à vous assurer que tous les accès sont correctement synchronisés. `static static`
- Vous pouvez accéder aux fonctions et variables déclarées via l'interface de fonction étrangère de Rust. Ceux-ci sont considérés même lorsqu'ils sont immuables, car ils sont visibles pour le code écrit dans d'autres langages qui peuvent ne pas respecter les règles de sécurité de Rust. `unsafe`

Restreindre les fonctionnalités dangereuses aux blocs ne vous empêche pas vraiment de faire ce que vous voulez. Il est parfaitement possible de simplement coller un bloc dans votre code et de passer à autre chose. L'avantage de la règle réside principalement dans le fait d'attirer l'attention humaine sur un code dont Rust ne peut garantir la sécurité : `unsafe unsafe`

- Vous n'utiliserez pas accidentellement des fonctionnalités dangereuses et ne découvrirez pas que vous étiez responsable de contrats dont vous ignoriez même l'existence.
- Un bloc attire davantage l'attention des réviseurs. Certains projets ont même une automatisation pour assurer cela, signalant les modifications de code qui affectent les blocs pour une attention particulière. `unsafe unsafe`
- Lorsque vous envisagez d'écrire un bloc, vous pouvez prendre un moment pour vous demander si votre tâche nécessite vraiment de telles mesures. Si c'est pour la performance, avez-vous des mesures pour montrer qu'il s'agit en fait d'un goulot d'étranglement? Peut-être y a-t-il un bon moyen d'accomplir la même chose dans Rust en toute sécurité. `unsafe`

# Exemple : un type de chaîne ASCII efficace

Voici la définition de , un type de chaîne qui garantit que son contenu est toujours valide ASCII. Ce type utilise une fonctionnalité dangereuse pour fournir une conversion à coût nul en : Ascii String

```
mod my_ascii {
 /// An ASCII-encoded string.
 #[derive(Debug, Eq, PartialEq)]
 pub struct Ascii(
 // This must hold only well-formed ASCII text:
 // bytes from `0` to `0x7f`.
 Vec<u8>
);

 impl Ascii {
 /// Create an `Ascii` from the ASCII text in `bytes`. Return an
 /// `NotAsciiError` error if `bytes` contains any non-ASCII
 /// characters.
 pub fn from_bytes(bytes: Vec<u8>) -> Result<Ascii, NotAsciiError> {
 if bytes.iter().any(|&byte| !byte.is_ascii()) {
 return Err(NotAsciiError(bytes));
 }
 Ok(Ascii(bytes))
 }
 }

 // When conversion fails, we give back the vector we couldn't convert.
 // This should implement `std::error::Error`; omitted for brevity.
 #[derive(Debug, Eq, PartialEq)]
 pub struct NotAsciiError(pub Vec<u8>);

 // Safe, efficient conversion, implemented using unsafe code.
 impl From<Ascii> for String {
 fn from(ascii: Ascii) -> String {
 // If this module has no bugs, this is safe, because
 // well-formed ASCII text is also well-formed UTF-8.
 unsafe { String::from_utf8_unchecked(ascii.0) }
 }
 }
 ...
}
```

La clé de ce module est la définition du type. Le type lui-même est marqué , pour le rendre visible à l'extérieur du module. Mais l'élément du type n'est *pas* public, de sorte que seul le module peut construire une valeur ou faire référence à son élément. Cela laisse le code du module en contrôle total sur ce qui peut ou non y apparaître. Tant que les constructeurs et les méthodes publics s'assurent que les valeurs nouvellement créées sont bien formées et le restent tout au long de leur vie, le reste du programme ne peut pas enfreindre cette règle. Et en effet, le constructeur public vérifie soigneusement le vecteur qui lui est donné avant d'accepter de construire un à partir de celui-ci. Par souci de brièveté, nous ne montrons aucune méthode, mais vous pouvez imaginer un ensemble de méthodes de gestion de texte qui garantissent que les valeurs contiennent toujours du texte ASCII approprié, tout comme les méthodes d'a garantissent que son contenu reste bien formé UTF-

```
8. Ascii pub my_ascii Vec<u8> my_ascii Ascii Ascii Ascii::from_bytes Ascii Ascii String
```

Cet arrangement nous permet de mettre en œuvre très efficacement. La fonction dangereuse prend un vecteur octet et en construit un à partir de celui-ci sans vérifier si son contenu est du texte UTF-8 bien formé; le contrat de la fonction tient son appelant responsable de cela. Heureusement, les règles appliquées par le type sont exactement ce dont nous avons besoin pour satisfaire le contrat de . Comme nous l'avons expliqué dans [« UTF-8 »](#), tout bloc de texte ASCII est également utf-8 bien formé, de sorte que le sous-jacent d'un 'est immédiatement prêt à servir de tampon de .. From<Ascii> String String::from\_utf8\_unchecked String Ascii from\_utf8\_unchecked Ascii Vec<u8> String

Avec ces définitions en place, vous pouvez écrire :

```
use my_ascii::Ascii;

let bytes: Vec<u8> = b"ASCII and ye shall receive".to_vec();

// This call entails no allocation or text copies, just a scan.
let ascii: Ascii = Ascii::from_bytes(bytes)
 .unwrap(); // We know these chosen bytes are ok.

// This call is zero-cost: no allocation, copies, or scans.
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");
```

Aucun bloc n'est requis pour utiliser . Nous avons mis en place une interface sécurisée utilisant des opérations dangereuses et nous nous sommes arrangés pour respecter leurs contrats en fonction uniquement du code du module, et non du comportement de ses utilisateurs. `unsafe`

An n'est rien de plus qu'un wrapper autour d'un , caché à l'intérieur d'un module qui applique des règles supplémentaires sur son contenu. Un type de ce type est appelé un *nouveau type*, un modèle commun dans Rust. Le propre type de Rust est défini exactement de la même manière, sauf que son contenu est limité à UTF-8, pas ASCII. En fait, voici la définition de la bibliothèque standard: `Ascii Vec<u8> String`

```
pub struct String {
 vec: Vec<u8>,
}
```

Au niveau de la machine, avec les types de Rust hors de l'image, un nouveau type et son élément ont des représentations identiques en mémoire, de sorte que la construction d'un nouveau type ne nécessite aucune instruction de la machine. Dans , l'expression considère simplement que la représentation de 'a maintenant une valeur. De même, ne nécessite probablement aucune instruction machine lorsqu'il est inséré: le est maintenant considéré comme un

```
.Ascii::from_bytes Ascii(bytes) Vec<u8> Ascii String::from_
utf8_unchecked Vec<u8> String
```

## Fonctions dangereuses

Une définition de fonction ressemble à une définition de fonction ordinaire précédée du mot-clé. Le corps d'une fonction est automatiquement considéré comme un bloc. `unsafe`

Vous ne pouvez appeler des fonctions qu'à l'intérieur des blocs. Cela signifie que le marquage d'une fonction avertit ses appelants que la fonction a un contrat qu'ils doivent remplir pour éviter un comportement indéfini. `unsafe`

Par exemple, voici un nouveau constructeur pour le type que nous avons introduit précédemment qui construit un vecteur à partir d'un octet sans vérifier si son contenu est ASCII valide : `Ascii Ascii`

```

// This must be placed inside the `my_ascii` module.
impl Ascii {
 /// Construct an `Ascii` value from `bytes`, without checking
 /// whether `bytes` actually contains well-formed ASCII.
 ///
 /// This constructor is infallible, and returns an `Ascii` direct]
 /// rather than a `Result<Ascii, NotAsciiError>` as the `from_byt
 /// constructor does.
 ///
 /// # Safety
 ///
 /// The caller must ensure that `bytes` contains only ASCII
 /// characters: bytes no greater than 0x7f. Otherwise, the effect
 /// undefined.
 pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
 Ascii(bytes)
 }
}

```

Vraisemblablement, l'appel de code sait déjà d'une manière ou d'une autre que le vecteur en main ne contient que des caractères ASCII, de sorte que la vérification qui insiste pour être effectuée serait une perte de temps, et l'appelant devrait écrire du code pour gérer des résultats dont il sait qu'ils ne se produiront jamais. permet à un tel appelant d'éviter les vérifications et la gestion des

erreurs. `Ascii::from_bytes_unchecked` `Ascii::from_bytes Err As`  
`cii::from_bytes_unchecked`

Mais plus tôt, nous avons souligné l'importance des constructeurs publics et des méthodes garantissant que les valeurs sont bien formées. Ne manque-t-il pas de s'acquitter de cette responsabilité?

`Ascii Ascii from_bytes_unchecked`

Pas tout à fait : remplit ses obligations en les transmettant à son appelant via son contrat. La présence de ce contrat est ce qui rend correct de marquer cette fonction : malgré le fait que la fonction elle-même n'effectue aucune opération dangereuse, ses appelants doivent suivre des règles que Rust ne peut pas appliquer automatiquement pour éviter un comportement indéfini. `from_bytes_unchecked unsafe`

Pouvez-vous vraiment provoquer un comportement indéfini en rompant le contrat de ? Oui. Vous pouvez construire un UTF-8 mal formé comme

```

suit::Ascii::from_bytes_unchecked String

// Imagine that this vector is the result of some complicated process
// that we expected to produce ASCII. Something went wrong!
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];

let ascii = unsafe {
 // This unsafe function's contract is violated
 // when `bytes` holds non-ASCII bytes.
 Ascii::from_bytes_unchecked(bytes)
};

let bogus: String = ascii.into();

// `bogus` now holds ill-formed UTF-8. Parsing its first character produces
// a `char` that is not a valid Unicode code point. That's undefined
// behavior, so the language doesn't say how this assertion should behave.
assert_eq!(bogus.chars().next().unwrap() as u32, 0xffff);

```

Dans certaines versions de Rust, sur certaines plates-formes, cette affirmation a échoué avec le message d'erreur divertissant suivant :

```

thread 'main' panicked at 'assertion failed: `(left == right)`
 left: `2097151`,
 right: `2097151``', src/main.rs:42:5

```

Ces deux chiffres nous semblent égaux, mais ce n'est pas la faute de Rust; c'est la faute du bloc précédent. Quand nous disons qu'un comportement indéfini conduit à des résultats imprévisibles, c'est le genre de chose que nous voulons dire. `unsafe`

Ceci illustre deux faits critiques sur les bogues et le code dangereux :

- *Les bogues qui se produisent avant le blocage dangereux peuvent briser les contrats.* Le fait qu'un bloc provoque un comportement non défini peut dépendre non seulement du code du bloc lui-même, mais également du code qui fournit les valeurs sur lesquelles il opère. Tout ce sur quoi votre code s'appuie pour satisfaire les contrats est essentiel pour la sécurité. La conversion de à basée sur n'est bien définie que si le reste du module maintient correctement les invariants de
 

```
.unsafe unsafe Ascii String String::from_utf8_unchecked As
ci
```

- *Les conséquences de la rupture d'un contrat peuvent apparaître après que vous ayez quitté le bloc dangereux.* Le comportement indéfini courtisé par le non-respect du contrat d'une fonctionnalité dangereuse ne se produit souvent pas dans le bloc lui-même. La construction d'un faux comme indiqué précédemment peut ne causer de problèmes que beaucoup plus tard dans l'exécution du programme. `unsafe String`

Essentiellement, le vérificateur de type, le vérificateur d'emprunt et d'autres contrôles statiques de Rust inspectent votre programme et tentent de construire la preuve qu'il ne peut pas présenter un comportement indéfini. Lorsque Rust compile votre programme avec succès, cela signifie qu'il a réussi à prouver le son de votre code. Un bloc est une lacune dans cette preuve : « Ce code », dites-vous à Rust, « va bien, croyez-moi. » La véracité de votre affirmation peut dépendre de n'importe quelle partie du programme qui influence ce qui se passe dans le bloc, et les conséquences d'être faux peuvent apparaître n'importe où influencées par le bloc. Écrire le mot-clé revient à rappeler que vous ne bénéficiez pas pleinement des contrôles de sécurité de la langue. `unsafe unsafe unsafe unsafe`

Si vous avez le choix, vous devriez naturellement préférer créer des interfaces sûres, sans contrats. Ceux-ci sont beaucoup plus faciles à utiliser, car les utilisateurs peuvent compter sur les contrôles de sécurité de Rust pour s'assurer que leur code est exempt de comportement non défini. Même si votre implémentation utilise des fonctionnalités dangereuses, il est préférable d'utiliser les types, les durées de vie et le système de modules de Rust pour respecter leurs contrats tout en utilisant uniquement ce que vous pouvez vous garantir, plutôt que de transférer les responsabilités à vos appelants.

Malheureusement, il n'est pas rare de rencontrer des fonctions dangereuses dans la nature dont la documentation ne prend pas la peine d'expliquer leurs contrats. Vous devez déduire les règles vous-même, en fonction de votre expérience et de votre connaissance du comportement du code. Si vous êtes déjà demandé si ce que vous faites avec une API C ou C ++ est OK, alors vous savez à quoi cela ressemble.

## Bloc dangereux ou fonction dangereuse?

Vous vous demandez peut-être s'il faut utiliser un bloc ou simplement marquer toute la fonction comme dangereuse. L'approche que nous recommandons est de prendre d'abord une décision concernant la fonction: `unsafe`

- S'il est possible d'abuser de la fonction d'une manière qui compile correctement mais provoque toujours un comportement indéfini, vous devez la marquer comme dangereuse. Les règles d'utilisation correcte de la fonction sont son contrat; l'existence d'un contrat est ce qui rend la fonction dangereuse.
- Sinon, la fonction est sûre : aucun appel bien tapé ne peut provoquer un comportement indéfini. Il ne doit pas être marqué . `unsafe`

La question de savoir si la fonction utilise des caractéristiques dangereuses dans son corps n'est pas pertinente; ce qui compte, c'est la présence d'un contrat. Auparavant, nous avons montré une fonction dangereuse qui n'utilise aucune fonctionnalité dangereuse et une fonction sécurisée qui utilise des fonctionnalités dangereuses.

Ne marquez pas une fonction sûre simplement parce que vous utilisez des caractéristiques dangereuses dans son corps. Cela rend la fonction plus difficile à utiliser et confond les lecteurs qui s'attendront (correctement) à trouver un contrat expliqué quelque part. Au lieu de cela, utilisez un bloc, même s'il s'agit du corps entier de la fonction. `unsafe unsafe`

## Comportement non défini

Dans l'introduction, nous avons dit que le terme *comportement indéfini* signifie « comportement que Rust suppose fermement que votre code ne pourrait jamais présenter ». C'est une étrange tournure de phrase, d'autant plus que nous savons par notre expérience avec d'autres langues que ces comportements se *produisent* par accident avec une certaine fréquence. Pourquoi ce concept est-il utile pour établir les obligations d'un code dangereux?

Un compilateur est un traducteur d'un langage de programmation à un autre. Le compilateur Rust prend un programme Rust et le traduit en un programme équivalent en langage machine. Mais qu'est-ce que cela signifie de dire que deux programmes dans des langues aussi complètement différentes sont équivalents?

Heureusement, cette question est plus facile pour les programmeurs que pour les linguistes. Nous disons généralement que deux programmes sont équivalents s'ils auront toujours le même comportement visible lorsqu'ils sont exécutés : ils font les mêmes appels système, interagissent avec des bibliothèques étrangères de manière équivalente, etc. C'est un peu comme un test de Turing pour les programmes : si vous ne pouvez pas dire si vous interagissez avec l'original ou la traduction, alors ils sont équivalents.

Considérez maintenant le code suivant :

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

Même en ne sachant rien de la définition de , nous pouvons voir qu'il ne reçoit qu'une référence partagée à , de sorte que l'appel ne peut pas changer la valeur de . Puisque la valeur transmise à sera toujours , Rust peut traduire ce code en langage machine comme si nous avions écrit : very\_trustworthy i i println! 1000

```
very_trustworthy(&10);
println!("{}", 1000);
```

Cette version transformée a le même comportement visible que l'original, et c'est probablement un peu plus rapide. Mais il est logique de ne considérer les performances de cette version que si nous convenons qu'elle a la même signification que l'original. Et si elles étaient définies comme suit ?

```
very_trustworthy

fn very_trustworthy(shared: &i32) {
 unsafe {
 // Turn the shared reference into a mutable pointer.
 // This is undefined behavior.
 let mutable = shared as *const i32 as *mut i32;
 *mutable = 20;
 }
}
```

Ce code enfreint les règles pour les références partagées : il change la valeur de en , même s'il doit être gelé car il est emprunté pour le partage.

En conséquence, la transformation que nous avons faite à l'appelant a maintenant un effet très visible: si Rust transforme le code, le programme imprime; s'il laisse le code seul et utilise la nouvelle valeur de , il imprime . Enfreindre les règles pour les références partagées signifie que les références partagées ne se comporteront pas comme prévu chez leurs appelants. `i 20 i 1000 i 2000 very_trustworthy`

Ce genre de problème se pose avec presque tous les types de transformation que Rust pourrait tenter. Même l'insertion d'une fonction dans son site d'appel suppose, entre autres, que lorsque l'appelé se termine, le flux de contrôle retourne au site d'appel. Mais nous avons ouvert le chapitre avec un exemple de code mal élevé qui viole même cette hypothèse.

Il est fondamentalement impossible pour Rust (ou tout autre langage) d'évaluer si une transformation en un programme préserve sa signification à moins qu'il ne puisse faire confiance aux caractéristiques fondamentales du langage pour se comporter comme prévu. Et qu'ils le fassent ou non peut dépendre non seulement du code à portée de main, mais d'autres parties potentiellement éloignées du programme. Afin de faire quoi que ce soit avec votre code, Rust doit supposer que le reste de votre programme se comporte bien.

Voici donc les règles de Rust pour les programmes bien élevés:

- Le programme ne doit pas lire la mémoire non initialisée.
- Le programme ne doit pas créer de valeurs primitives non valides :
  - Références, zones ou pointeurs qui sont `fn null`
  - `bool` valeurs qui ne sont pas `a` ou `0 1`
  - `enum` valeurs avec des valeurs discriminantes non valides
  - `char` valeurs qui ne sont pas valides, points de code Unicode non substitution
  - `str` valeurs qui ne sont pas bien formées UTF-8
  - Pointeurs de graisse avec des vtables/longueurs de tranche non valides
  - Toute valeur du type « jamais », écrite , pour les fonctions qui ne renvoient pas !
- Les règles de référence expliquées au [chapitre 5](#) doivent être suivies.  
Aucune référence ne peut survivre à son référent; l'accès partagé est un accès en lecture seule ; et l'accès mutable est un accès exclusif.
- Le programme ne doit pas déréférencer les pointeurs null, mal alignés ou pendants.

- Le programme ne doit pas utiliser de pointeur pour accéder à la mémoire en dehors de l'allocation à laquelle le pointeur est associé. Nous expliquerons cette règle en détail dans « [Déréférencement des pointeurs bruts en toute sécurité](#) ».
- Le programme doit être exempt de courses de données. Une course de données se produit lorsque deux threads accèdent au même emplacement de mémoire sans synchronisation et qu'au moins un des accès est une écriture.
- Le programme ne doit pas se dérouler sur un appel effectué à partir d'une autre langue, via l'interface de fonction étrangère, comme expliqué dans « [Déroulement](#) ».
- Le programme doit être conforme aux contrats des fonctions standard de la bibliothèque.

Comme nous n'avons pas encore de modèle complet de la sémantique de Rust pour le code, cette liste évoluera probablement avec le temps, mais celles-ci resteront probablement interdites. `unsafe`

Toute violation de ces règles constitue un comportement indéfini et rend les efforts de Rust pour optimiser votre programme et le traduire en langage machine indignes de confiance. Si vous enfreignez la dernière règle et passez UTF-8 mal formé à , peut-être que 2097151 n'est pas si égal à 2097151 après tout. `String::from_utf8_unchecked`

Le code Rust qui n'utilise pas de fonctionnalités dangereuses est garanti de suivre toutes les règles précédentes, une fois compilé (en supposant que le compilateur n'a pas de bogues; nous y arrivons, mais la courbe n'intersectera jamais l'asymptote). Ce n'est que lorsque vous utilisez des fonctionnalités dangereuses que ces règles deviennent votre responsabilité.

En C et C++, le fait que votre programme compile sans erreurs ni avertissements signifie beaucoup moins; Comme nous l'avons mentionné dans l'introduction de ce livre, même les meilleurs programmes C et C ++ écrits par des projets respectés qui maintiennent leur code à des normes élevées présentent un comportement indéfini dans la pratique.

## Traits dangereux

Un *trait dangereux* est un trait qui a un contrat que Rust ne peut pas vérifier ou appliquer que les implémentateurs doivent faire pour éviter un

comportement indéfini. Pour implémenter un caractère dangereux, vous devez marquer l'implémentation comme dangereuse. C'est à vous de comprendre le contrat du trait et de vous assurer que votre type le satisfait.

Une fonction qui limite ses variables de type avec un trait dangereux est généralement une fonction qui utilise elle-même des entités dangereuses et satisfait leurs contrats uniquement en fonction du contrat du caractère dangereux. Une implémentation incorrecte du trait pourrait entraîner un comportement indéfini d'une telle fonction.

`std::marker::Send` et `std::marker::Sync` sont les exemples classiques de traits dangereux. Ces traits ne définissent aucune méthode, ils sont donc triviaux à mettre en œuvre pour tout type que vous aimez. Mais ils ont des contrats : exige que les implémenteurs soient en sécurité pour passer à un autre thread, et exige qu'ils soient sûrs à partager entre les threads via des références partagées. La mise en œuvre pour un type inapproprié, par exemple, ne serait plus à l'abri des courses de données. `std::marker::Sync Send Sync Send std::sync::Mutex`

À titre d'exemple simple, la bibliothèque standard Rust incluait un trait dangereux, `Zeroable`, pour les types qui peuvent être initialisés en toute sécurité en définissant tous leurs octets à zéro. De toute évidence, la mise à zéro de `a` est très bien, mais la mise à zéro de `a` vous donne une référence nulle, ce qui provoquera un crash si elle est déréférencée. Pour les types qui étaient `Zeroable`, certaines optimisations étaient possibles: vous pouviez initialiser un tableau d'entre eux rapidement avec (l'équivalent de Rust de `memset(0, size)`) ou utiliser des appels de système d'exploitation qui allouent des pages à zéro. (étaient instables et déplacés vers une utilisation interne uniquement dans la caisse dans Rust 1.26, mais c'est un bon exemple simple et réel.) `core::nonzero::Zeroable` `usize &T Zeroable` `std::ptr::write_bytes(0, num)`

`Zeroable` était un trait marqueur typique, dépourvu de méthodes ou de types associés :

```
pub unsafe trait Zeroable {}
```

Les implémentations pour les types appropriés étaient tout aussi simples :

```

unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// and so on for all the integer types

```

Avec ces définitions, on pourrait écrire une fonction qui alloue rapidement un vecteur d'une longueur donnée contenant un type : `Zeroable`

```

use core::nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
 where T: Zeroable
{
 let mut vec = Vec::with_capacity(len);
 unsafe {
 std::ptr::write_bytes(vec.as_mut_ptr(), 0, len);
 vec.set_len(len);
 }
 vec
}

```

Cette fonction commence par créer un vide avec la capacité requise, puis appelle à remplir le tampon inoccupé avec des zéros. (La fonction traite comme un nombre d'éléments, et non comme un nombre d'octets, de sorte que cet appel remplit la totalité de la mémoire tampon.) La méthode d'un vecteur change de longueur sans rien faire au tampon ; ceci n'est pas sûr, car vous devez vous assurer que l'espace tampon nouvellement fermé contient effectivement des valeurs de type correctement initialisées . Mais c'est exactement ce que la liaison établit : un bloc de zéro octet représente une valeur valide. Notre utilisation de était sûre. `Vec write_bytes write_byte len T set_len T T:`  
`Zeroable T set_len`

Ici, nous l'avons mis à profit:

```

let v: Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));

```

De toute évidence, doit être un trait dangereux, car une implémentation qui ne respecte pas son contrat peut conduire à un comportement indéfini: `Zeroable`

```

struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v: Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1; // crashes: dereferences null pointer

```

Rust n'a aucune idée de ce qui est censé signifier, il ne peut donc pas dire quand il est mis en œuvre pour un type inapproprié. Comme pour toute autre fonctionnalité dangereuse, c'est à vous de comprendre et d'adhérer au contrat d'un trait dangereux. `Zeroable`

Notez que le code dangereux ne doit pas dépendre de caractéristiques ordinaires et sûres correctement implémentées. Par exemple, supposons qu'il y ait une implémentation du trait qui renvoie simplement une valeur de hachage aléatoire, sans rapport avec les valeurs hachées. Le trait exige que le hachage des mêmes bits deux fois produise la même valeur de hachage, mais cette implémentation ne répond pas à cette exigence ; c'est tout simplement incorrect. Mais parce que ce n'est pas un trait dangereux, le code dangereux ne doit pas présenter un comportement indéfini lorsqu'il utilise ce hasher. Le type est soigneusement écrit pour respecter les contrats des fonctionnalités dangereuses qu'il utilise, quel que soit le comportement du hasher. Certes, la table ne fonctionnera pas correctement : les recherches échoueront et les entrées apparaîtront et disparaîtront au hasard. Mais la table ne présentera pas de comportement indéfini. `std::hash::Hasher` `std::collections::HashMap`

## Pointeurs bruts

Un *pointeur brut* dans Rust est un pointeur sans contrainte. Vous pouvez utiliser des pointeurs bruts pour former toutes sortes de structures que les types de pointeurs vérifiés de Rust ne peuvent pas, comme des listes doublement liées ou des graphiques arbitraires d'objets. Mais parce que les pointeurs bruts sont si flexibles, Rust ne peut pas dire si vous les utilisez en toute sécurité ou non, vous ne pouvez donc les déréférencer que dans un bloc. `unsafe`

Les pointeurs bruts sont essentiellement équivalents aux pointeurs C ou C++, ils sont donc également utiles pour interagir avec le code écrit dans

ces langages.

Il existe deux types de pointeurs bruts :

- A est un pointeur brut vers a qui permet de modifier son référent. `*mut T T`
- A est un pointeur brut vers a qui ne permet que de lire son référent. `*const T T`

(Il n'y a pas de type simple ; vous devez toujours spécifier l'un ou l'autre.) `*T const mut`

Vous pouvez créer un pointeur brut par conversion à partir d'une référence et le déréférencer avec l'opérateur : \*

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
 *ptr_x += *ptr_y;
}
assert_eq!(x, 30);
```

Contrairement aux boîtes et aux références, les pointeurs bruts peuvent être nuls, comme en C ou en C++ : `NULL nullptr`

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
 match opt {
 None => std::ptr::null(),
 Some(r) => r as *const T
 }
}

assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw::<i32>(None), std::ptr::null());
```

Cet exemple n'a pas de blocs : la création de pointeurs bruts, leur transmission et leur comparaison sont tous sûrs. Seul le déréférencement d'un pointeur brut n'est pas sûr. `unsafe`

Un pointeur brut vers un type non dimensionné est un pointeur gras, tout comme la référence ou le type correspondant. Un pointeur inclut une longueur avec l'adresse, et un objet trait comme un pointeur porte un vtable.

```
Box *const [u8] *mut dyn std::io::Write
```

Bien que Rust déréfère implicitement les types de pointeurs sûrs dans diverses situations, les déréférencements de pointeurs bruts doivent être explicites :

- L'opérateur ne déréfère pas implicitement un pointeur brut ; vous devez écrire ou ... (\*raw).field (\*raw).method(...)
- Les pointeurs bruts n'implémentent pas , de sorte que les coercitions deref ne s'appliquent pas à eux. Deref
- Les opérateurs aiment et comparent les pointeurs bruts en tant qu'adresses : deux pointeurs bruts sont égaux s'ils pointent vers le même emplacement en mémoire. De même, le hachage d'un pointeur brut hache l'adresse vers laquelle il pointe, et non la valeur de son référent. == <
- Les traits de mise en forme tels que suivre les références automatiquement, mais ne gèrent pas du tout les pointeurs bruts. Les exceptions sont et , qui affichent les pointeurs bruts sous forme d'adresses hexadécimales, sans les déréférencer. std::fmt::Display std::fmt::Debug std::fmt::Pointer

Contrairement à l'opérateur en C et C++, Rust ne gère pas les pointeurs bruts, mais vous pouvez effectuer l'arithmétique des pointeurs via leurs méthodes et méthodes, ou les méthodes plus pratiques , , et . Inversement, la méthode donne la distance entre deux pointeurs en octets, bien que nous soyons responsables de nous assurer que le début et la fin sont dans la même région de mémoire (la même, par exemple):

```
+ + offset wrapping_offset add sub wrapping_add wrapping_sub offset_from Vec
```

```
let trucks = vec!["garbage truck", "dump truck", "moonstruck"];
let first: *const &str = &trucks[0];
let last: *const &str = &trucks[2];
assert_eq!(unsafe { last.offset_from(first) }, 2);
assert_eq!(unsafe { first.offset_from(last) }, -2);
```

Aucune conversion explicite n'est nécessaire pour et ; il suffit de spécifier le type. Rust constraint implicitement les références à des pointeurs bruts (mais pas l'inverse, bien sûr). `first last`

L'opérateur permet presque toutes les conversions plausibles à partir de références à des pointeurs bruts ou entre deux types de pointeurs bruts. Cependant, vous devrez peut-être diviser une conversion complexe en une série d'étapes plus simples. Par exemple: as

```
&vec![42_u8] as *const String; // error: invalid conversion
&vec![42_u8] as *const Vec<u8> as *const String; // permitted
```

Notez que cela ne convertira pas les pointeurs bruts en références. De telles conversions seraient dangereuses et devraient rester une opération sûre. Au lieu de cela, vous devez déréférencer le pointeur brut (dans un bloc), puis emprunter la valeur résultante. as as unsafe

Soyez très prudent lorsque vous faites cela: une référence produite de cette manière a une durée de vie sans contrainte: il n'y a pas de limite à la durée de vie, car le pointeur brut ne donne rien à Rust sur lequel fonder une telle décision. Dans [« A Safe Interface to libgit2 »](#) plus loin dans ce chapitre, nous montrons plusieurs exemples de la façon de limiter correctement les durées de vie.

De nombreux types ont et méthodes qui renvoient un pointeur brut à leur contenu. Par exemple, les tranches de tableau et les chaînes renvoient des pointeurs vers leurs premiers éléments, et certains itérateurs renvoient un pointeur vers l'élément suivant qu'ils produiront. Posséder des types de pointeurs tels que , , et avoir et des fonctions qui convertissent vers et à partir de pointeurs bruts. Certains contrats de ces méthodes imposent des exigences surprenantes, alors vérifiez leur documentation avant de les

utiliser. `as_ptr as_mut_ptr Box Rc Arc into_raw from_raw`

Vous pouvez également construire des pointeurs bruts par conversion à partir d'entiers, bien que les seuls entiers auxquels vous pouvez faire confiance pour cela soient généralement ceux que vous avez obtenus d'un pointeur en premier lieu. [« Exemple: RefWithFlag » utilise des pointeurs bruts de cette façon.](#)

Contrairement aux références, les pointeurs bruts ne sont ni ni . Par conséquent, tout type qui inclut des pointeurs bruts n'implémente pas ces traits par défaut. Il n'y a rien d'intrinsèquement dangereux à envoyer ou à partager des pointeurs bruts entre les threads ; après tout, où qu'ils aillent, vous avez toujours besoin d'un bloc pour les déréférencer. Mais étant donné les rôles que jouent généralement les pointeurs bruts, les concepteurs de langage ont considéré ce comportement comme la valeur par défaut la plus utile. Nous avons déjà discuté de la façon de mettre en œuvre et vous-même dans « [Traits](#)

[dangereux ».](#) Send Sync unsafe Send Sync

## Déréférencement des pointeurs bruts en toute sécurité

Voici quelques directives de bon sens pour utiliser les pointeurs bruts en toute sécurité :

- Le déréférencement de pointeurs nuls ou de pointeurs pendants est un comportement indéfini, tout comme la mémoire non initialisée ou les valeurs qui sont sorties de la portée.
- Le déréférencement des pointeurs qui ne sont pas correctement alignés pour leur type de référent n'est pas un comportement défini.
- Vous ne pouvez emprunter des valeurs à partir d'un pointeur brut déréférencé que si cela respecte les règles de sécurité des références expliquées au [chapitre 5](#) : aucune référence ne peut survivre à son référent, l'accès partagé est un accès en lecture seule et l'accès mutable est un accès exclusif. (Cette règle est facile à enfreindre par accident, car les pointeurs bruts sont souvent utilisés pour créer des structures de données avec un partage ou une propriété non standard.)
- Vous ne pouvez utiliser le référent d'un pointeur brut que s'il s'agit d'une valeur bien formée de son type. Par exemple, vous devez vous assurer que le déréférencement d'un produit produit un point de code Unicode approprié et non aurigué. `*const char`
- Vous pouvez utiliser les méthodes et sur les pointeurs bruts uniquement pour pointer vers des octets dans la variable ou le bloc de mémoire alloué au tas auquel le pointeur d'origine faisait référence, ou vers le premier octet au-delà d'une telle région. `offset wrapping_offset`  
Si vous effectuez de l'arithmétique de pointeur en convertissant le pointeur en entier, en effectuant de l'arithmétique sur l'entier, puis en

le reconvertisant en pointeur, le résultat doit être un pointeur que les règles de la méthode vous auraient permis de produire. `offset`

- Si vous affectez au référent d'un pointeur brut, vous ne devez pas violer les invariants de tout type dont le référent fait partie. Par exemple, si vous avez un pointage vers un octet de `a`, vous ne pouvez stocker que des valeurs dans ce qui laisse le maintien UTF-8 bien formé. `*mut`  
`u8 String u8 String`

La règle d'emprunt mise à part, ce sont essentiellement les mêmes règles que vous devez suivre lorsque vous utilisez des pointeurs en C ou C++.

La raison de ne pas violer les invariants des types doit être claire. De nombreux types standard de Rust utilisent du code dangereux dans leur mise en œuvre, mais fournissent toujours des interfaces sûres en supposant que les contrôles de sécurité, le système de modules et les règles de visibilité de Rust seront respectés. L'utilisation de pointeurs bruts pour contourner ces mesures de protection peut conduire à un comportement indéfini.

Le contrat complet et exact pour les pointeurs bruts n'est pas facile à énoncer et peut changer à mesure que le langage évolue. Mais les principes décrits ici devraient vous garder en territoire sûr.

## Exemple : RefWithFlag

Voici un exemple de la façon de prendre un classique<sup>1</sup> hack au niveau des bits rendu possible par des pointeurs bruts et l'envelopper comme un type Rust complètement sûr. Ce module définit un type, , qui contient à la fois `a` et `a`, comme le tuple et qui parvient tout de même à n'occuper qu'un seul mot machine au lieu de deux. Ce type de technique est régulièrement utilisé dans les garbage collectors et les machines virtuelles, où certains types, par exemple le type représentant un objet, sont si nombreux que l'ajout d'un seul mot à chaque valeur augmenterait considérablement l'utilisation de la mémoire : `RefWithFlag<'a,`

`T> &'a T bool (&'a T, bool)`

```
mod ref_with_flag {
 use std::marker::PhantomData;
 use std::mem::align_of;

 /// A `&T` and a `bool`, wrapped up in a single word.
 /// The type `T` must require at least two-byte alignment.
```

```

/// If you're the kind of programmer who's never met a pointer who
/// 20-bit you didn't want to steal, well, now you can do it safely
/// ("But it's not nearly as exciting this way...")
pub struct RefWithFlag<'a, T> {
 ptr_and_bit: usize,
 behaves_like: PhantomData<&'a T> // occupies no space
}

impl<'a, T: 'a> RefWithFlag<'a, T> {
 pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
 assert!(align_of::<T>() % 2 == 0);
 RefWithFlag {
 ptr_and_bit: ptr as *const T as usize | flag as usize,
 behaves_like: PhantomData
 }
 }

 pub fn get_ref(&self) -> &'a T {
 unsafe {
 let ptr = (self.ptr_and_bit & !1) as *const T;
 &*ptr
 }
 }

 pub fn get_flag(&self) -> bool {
 self.ptr_and_bit & 1 != 0
 }
}

```

Ce code tire parti du fait que de nombreux types doivent être placés à des adresses paires en mémoire: puisque le bit le moins significatif d'une adresse paire est toujours nul, nous pouvons y stocker autre chose et ensuite reconstruire de manière fiable l'adresse d'origine simplement en masquant le bit inférieur. Tous les types ne sont pas admissibles; par exemple, les types et peuvent être placés à n'importe quelle adresse. Mais nous pouvons vérifier l'alignement du type sur la construction et refuser les types qui ne fonctionneront pas. u8 (bool, [i8; 2])

Vous pouvez utiliser comme ceci: RefWithFlag

```
use ref_with_flag::RefWithFlag;
```

```

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);

```

Le constructeur prend une référence et une valeur, affirme que le type de la référence est approprié, puis convertit la référence en pointeur brut, puis en . Le type est défini comme étant suffisamment grand pour contenir un pointeur sur n'importe quel processeur pour lequel nous compilons, donc la conversion d'un pointeur brut en a et inversement est bien définie. Une fois que nous avons un , nous savons qu'il doit être pair, nous pouvons donc utiliser l'opérateur bitwise-or pour le combiner avec le , que nous avons converti en un entier 0 ou

1. RefWithFlag::new bool usize usize usize usize | bool

La méthode extrait le composant d'un fichier . C'est simple: il suffit de masquer le bit inférieur et de vérifier s'il n'est pas nul.

`get_flag bool RefWithFlag`

La méthode extrait la référence d'un fichier . Tout d'abord, il masque le bit inférieur de 's et le convertit en pointeur brut. L'opérateur ne convertira pas les pointeurs bruts en références, mais nous pouvons déréférencer le pointeur brut (dans un bloc, naturellement) et l'emprunter. Emprunter le référent d'un pointeur brut vous donne une référence avec une durée de vie illimitée : Rust accordera la référence quelle que soit la durée de vie qui ferait vérifier le code qui l'entoure, s'il y en a une. Habituellement, cependant, il y a une durée de vie spécifique qui est plus précise et qui détecterait donc plus d'erreurs. Dans ce cas, puisque le type de retour de 'est , Rust voit que la durée de vie de la référence est la même que le paramètre de durée de vie de ', ce qui est exactement ce que nous voulons: c'est la durée de vie de la référence avec laquelle nous avons commencé.

`get_ref RefWithFlag usize as unsafe get_ref & 'a`  
`T RefWithFlag 'a`

En mémoire, a ressemble à un : puisqu'il s'agit d'un type de taille nulle, le champ ne prend aucune place dans la structure. Mais il est nécessaire pour Rust de savoir comment traiter les durées de vie dans le code qui utilise . Imaginez à quoi ressemblerait le type sans le champ

`: RefWithFlag usize PhantomData behaves_like PhantomData Ref`  
`WithFlag behaves_like`

```
// This won't compile.
pub struct RefWithFlag<'a, T: 'a> {
 ptr_and_bit: usize
}
```

Au [chapitre 5](#), nous avons souligné que toute structure contenant des références ne doit pas survivre aux valeurs qu'elles empruntent, de peur que les références ne deviennent des indications pendantes. La structure doit respecter les restrictions qui s'appliquent à ses champs. Cela s'applique certainement à : dans l'exemple de code que nous venons de regarder, ne doit pas survivre, puisqu'il renvoie une référence à celui-ci. Mais notre type réduit ne contient aucune référence et n'utilise jamais son paramètre de durée de vie. C'est juste un fichier . Comment Rust devrait-il savoir que des restrictions s'appliquent à la durée de vie de ? L'inclusion d'un champ indique à Rust de traiter *comme s'il* contenait un , sans affecter réellement la représentation de la

```
structure. RefWithFlag flagged vec flagged.get_ref() RefWithFlag<'a,
T> &'a T
```

Bien que Rust ne sache pas vraiment ce qui se passe (c'est ce qui rend dangereux), il fera de son mieux pour vous aider avec cela. Si vous omettez le champ, Rust se plaindra que les paramètres et sont inutilisés et suggère d'utiliser un . RefWithFlag behaves\_like 'a T PhantomData

RefWithFlag utilise les mêmes tactiques que le type que nous avons présenté précédemment pour éviter un comportement indéfini dans son bloc. Le type lui-même est , mais ses champs ne le sont pas, ce qui signifie que seul le code du module peut créer ou regarder à l'intérieur d'une valeur. Vous n'avez pas besoin d'inspecter beaucoup de code pour avoir l'assurance que le champ est bien

```
construit. Ascii unsafe pub ref_with_flag RefWithFlag ptr_and_
bit
```

## Pointeurs Nullable

Un pointeur brut nul dans Rust est une adresse zéro, tout comme en C et C++. Pour tout type , la fonction std::ptr::null<T> renvoie un pointeur null et renvoie un pointeur NULL. T \*const T std::ptr::null\_mut<T> \*mut T

Il existe plusieurs façons de vérifier si un pointeur brut est nul. La plus simple est la méthode, mais la méthode peut être plus pratique : elle prend un pointeur et renvoie un , transformant un pointeur null en un . De même, la méthode convertit les pointeurs en valeurs.

```
is_null as_ref *const T Option<&'a
T> None as_mut *mut T Option<&'a mut T>
```

## Tailles de type et alignements

Une valeur de tout type occupe un nombre constant d'octets en mémoire et doit être placée à une adresse qui est un multiple d'une certaine valeur *d'alignement*, déterminée par l'architecture de la machine. Par exemple, un tuple occupe huit octets, et la plupart des processeurs préfèrent qu'il soit placé à une adresse qui est un multiple de quatre. `Sized (i32, i32)`

L'appel renvoie la taille d'une valeur de type , en octets, et renvoie l'alignement requis. Par exemple:

```
std::mem::size_of::<T>()
T std::mem::align_of::<T>()
```

```
assert_eq!(std::mem::size_of::<i64>(), 8);
assert_eq!(std::mem::align_of::<(i32, i32)>(), 4);
```

L'alignement de tout type est toujours une puissance de deux.

La taille d'un type est toujours arrondie à un multiple de son alignement, même s'il pourrait techniquement tenir dans moins d'espace. Par exemple, même si un tuple comme ne nécessite que cinq octets, est , parce que est . Cela garantit que si vous avez un tableau, la taille du type d'élément reflète toujours l'espacement entre un élément et le suivant. `(f32, u8) size_of::<(f32, u8)>() 8 align_of::<(f32, u8)>() 4`

Pour les types non dimensionnés, la taille et l'alignement dépendent de la valeur à portée de main. Étant donné une référence à une valeur non dimensionnée, les fonctions et renvoient la taille et l'alignement de la valeur. Ces fonctions peuvent fonctionner sur des références à des types à la fois et à des types non dimensionnés

```
: std::mem::size_of_val std::mem::align_of_val Sized
```

```
// Fat pointers to slices carry their referent's length.
let slice: &[i32] = &[1, 3, 9, 27, 81];
```

```

assert_eq!(std::mem::size_of_val(slice), 20);

let text: &str = "alligator";
assert_eq!(std::mem::size_of_val(text), 9);

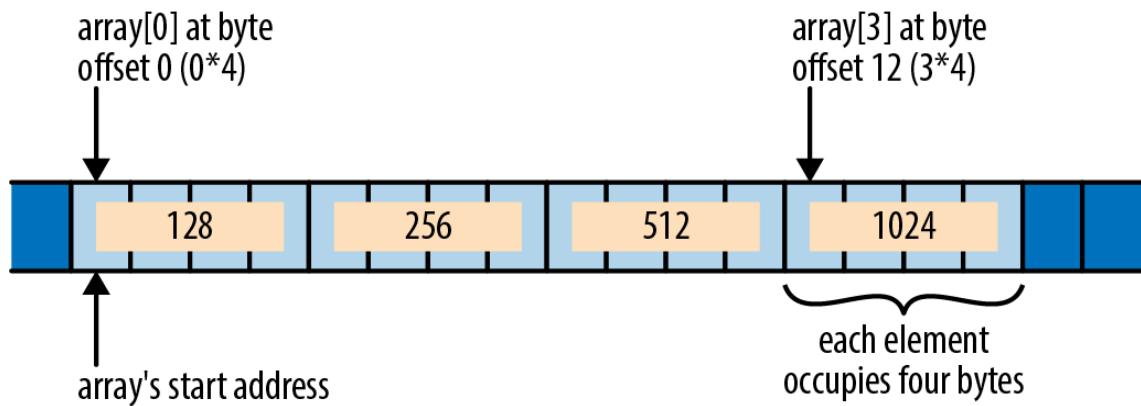
use std::fmt::Display;
let unremarkable: &dyn Display = &193_u8;
let remarkable: &dyn Display = &0.0072973525664;

// These return the size/alignment of the value the
// trait object points to, not those of the trait object
// itself. This information comes from the vtable the
// trait object refers to.
assert_eq!(std::mem::size_of_val(unremarkable), 1);
assert_eq!(std::mem::align_of_val(remarkable), 8);

```

## Arithmétique du pointeur

Rust présente les éléments d'un tableau, d'une tranche ou d'un vecteur sous la forme d'un seul bloc de mémoire contigu, comme illustré à [la figure 22-1](#). Les éléments sont régulièrement espacés, de sorte que si chaque élément occupe des octets, le  $i$ ème élément commence par le  $i$ ème octet.  $\text{size} = i \cdot \text{size}$



Graphique 22-1. Une baie en mémoire

Une bonne conséquence de ceci est que si vous avez deux pointeurs bruts vers des éléments d'un tableau, la comparaison des pointeurs donne les mêmes résultats que la comparaison des indices des éléments: si  $i < j$ , alors un pointeur brut vers le  $i$ th élément est inférieur à un pointeur brut vers le  $j$ ème élément. Cela rend les pointeurs bruts utiles en tant que limites sur les traversées de tableau. En fait, l'itérateur simple de la bibliothèque standard sur une tranche a été défini à l'origine comme suit :  $i < j \iff i \leq j$

```

struct Iter<'a, T> {
 ptr: *const T,
 end: *const T,
 ...
}

```

Le champ pointe vers l'élément suivant que l'itération doit produire, et le champ sert de limite : lorsque , l'itération est terminée. `ptr == end`

Une autre conséquence intéressante de la disposition du tableau: si est un pointeur brut vers le ème élément d'un tableau, alors est un pointeur brut vers le ème élément. Sa définition est équivalente à ceci

```
:element_ptr *const T *mut T i element_ptr.offset(o) (i +
o)
```

```

fn offset<T>(ptr: *const T, count: isize) -> *const T
 where T: Sized
{
 let bytes_per_element = std::mem::size_of::<T>() as isize;
 let byte_offset = count * bytes_per_element;
 (ptr as isize).checked_add(byte_offset).unwrap() as *const T
}

```

La fonction renvoie la taille du type en octets. Étant donné qu'il est, par définition, assez grand pour contenir une adresse, vous pouvez convertir le pointeur de base en , faire de l'arithmétique sur cette valeur, puis reconvertir le résultat en pointeur. `std::mem::size_of::`

```
<T> T isize isize
```

Il est bon de produire un pointeur vers le premier octet après la fin d'un tableau. Vous ne pouvez pas déréférencer un tel pointeur, mais il peut être utile de représenter la limite d'une boucle ou pour les contrôles de limites.

Toutefois, il s'agit d'un comportement indéfini à utiliser pour produire un pointeur au-delà de ce point ou avant le début du tableau, même si vous ne le déréférez jamais. Dans un souci d'optimisation, Rust aimeraient supposer que quand est positif et que quand est négatif. Cette hypothèse semble sûre, mais elle peut ne pas tenir si l'arithmétique dans déborde une valeur. Si est contraint de rester dans le même tableau que , aucun débordement ne peut se produire : après tout, le tableau lui-même ne dé-

passe pas les limites de l'espace d'adressage. (Pour sécuriser les pointeurs vers le premier octet après la fin, Rust ne place jamais de valeurs à l'extrême supérieure de l'espace d'adressage.) `offset ptr.offset(i) > ptr i ptr.offset(i) < ptr i offset isize i ptr`

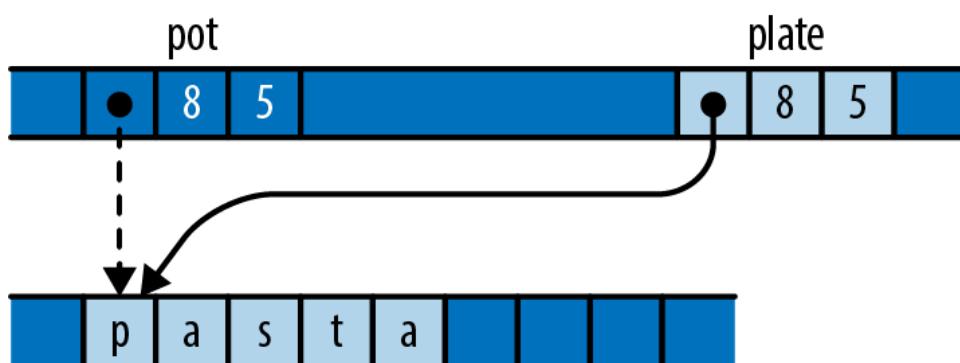
Si vous devez décaler les pointeurs au-delà des limites du tableau auquel ils sont associés, vous pouvez utiliser la méthode. Ceci est équivalent à , mais Rust ne fait aucune hypothèse sur l'ordre relatif de et lui-même. Bien sûr, vous ne pouvez toujours pas déréférencer ces pointeurs à moins qu'ils ne tombent dans le tableau. `wrapping_offset offset ptr.wrapping_offset(i) ptr`

## Entrer et sortir de la mémoire

Si vous implémentez un type qui gère sa propre mémoire, vous devrez suivre quelles parties de votre mémoire contiennent des valeurs actives et lesquelles ne sont pas initialisées, tout comme Rust le fait avec les variables locales. Considérez ce code :

```
let pot = "pasta".to_string();
let plate = pot;
```

Une fois ce code exécuté, la situation ressemble à [la figure 22-2](#).



Graphique 22-2. Déplacement d'une chaîne d'une variable locale à une autre

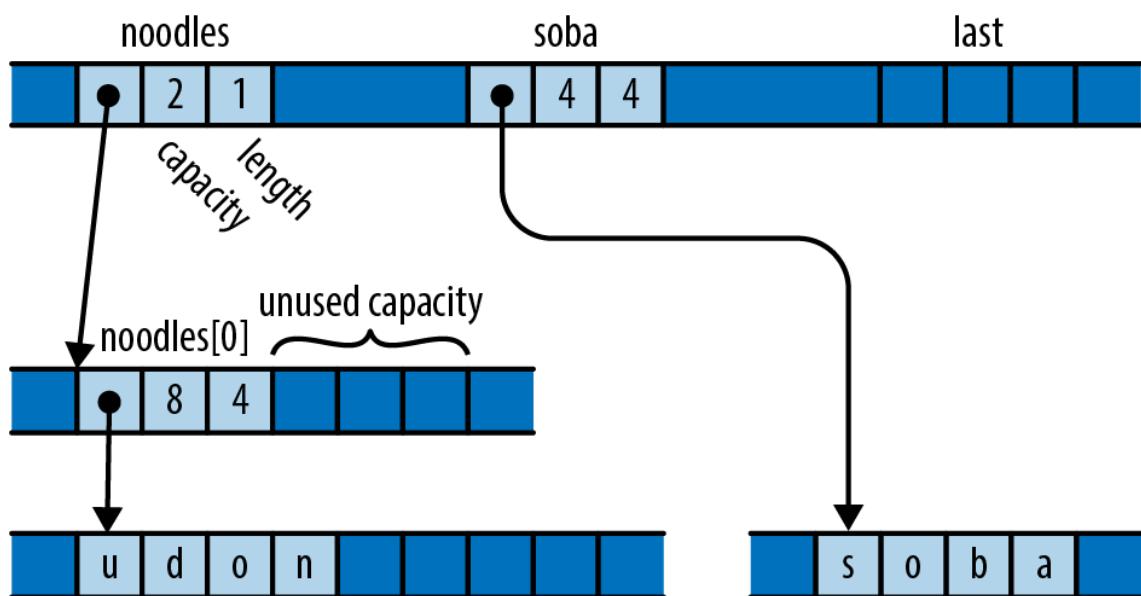
Après l'affectation, n'est pas initialisé et est le propriétaire de la chaîne. `pot plate`

Au niveau de la machine, il n'est pas spécifié ce qu'un déplacement fait à la source, mais dans la pratique, il ne fait généralement rien du tout. L'affectation laisse probablement toujours un pointeur, une capacité et une longueur pour la chaîne. Naturellement, il serait désastreux de traiter cela comme une valeur réelle, et Rust s'assure que vous ne le faites pas. `pot`

Les mêmes considérations s'appliquent aux structures de données qui gèrent leur propre mémoire. Supposons que vous exécutez ce code :

```
let mut noodles = vec!["udon".to_string()];
let soba = "soba".to_string();
let last;
```

En mémoire, l'état ressemble à [la figure 22-3](#).

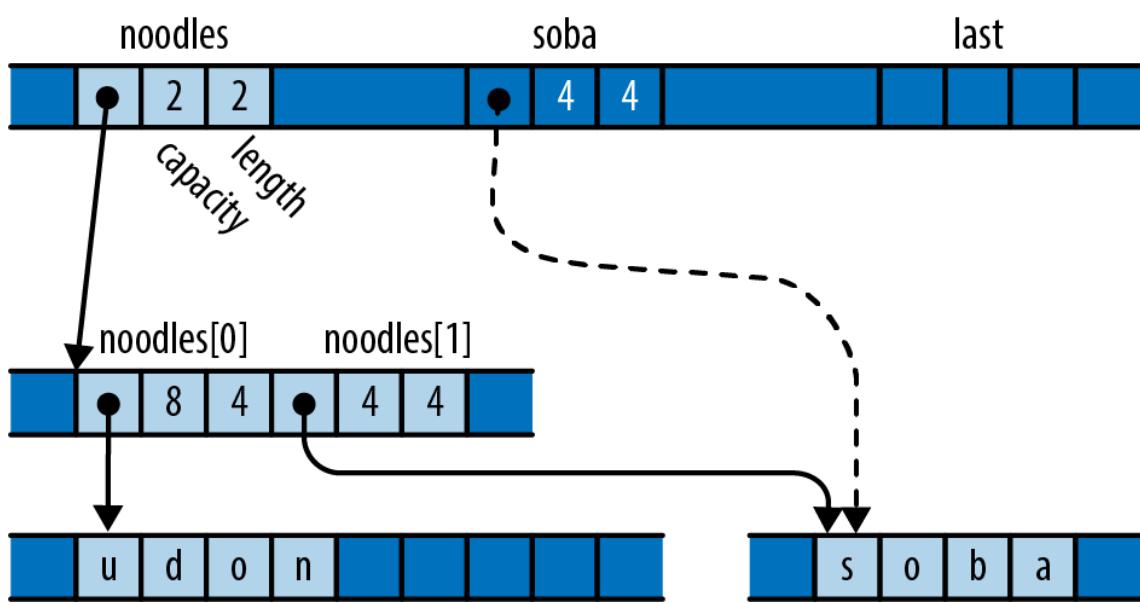


Graphique 22-3. Un vecteur avec une capacité inutilisée non initialisée

Le vecteur a la capacité de réserver pour contenir un élément de plus, mais son contenu est indésirable, probablement quelle que soit la mémoire détenue précédemment. Supposons que vous exécutez ensuite ce code :

```
noodles.push(soba);
```

Pousser la chaîne sur le vecteur transforme cette mémoire non initialisée en un nouvel élément, comme illustré à [la figure 22-4](#).



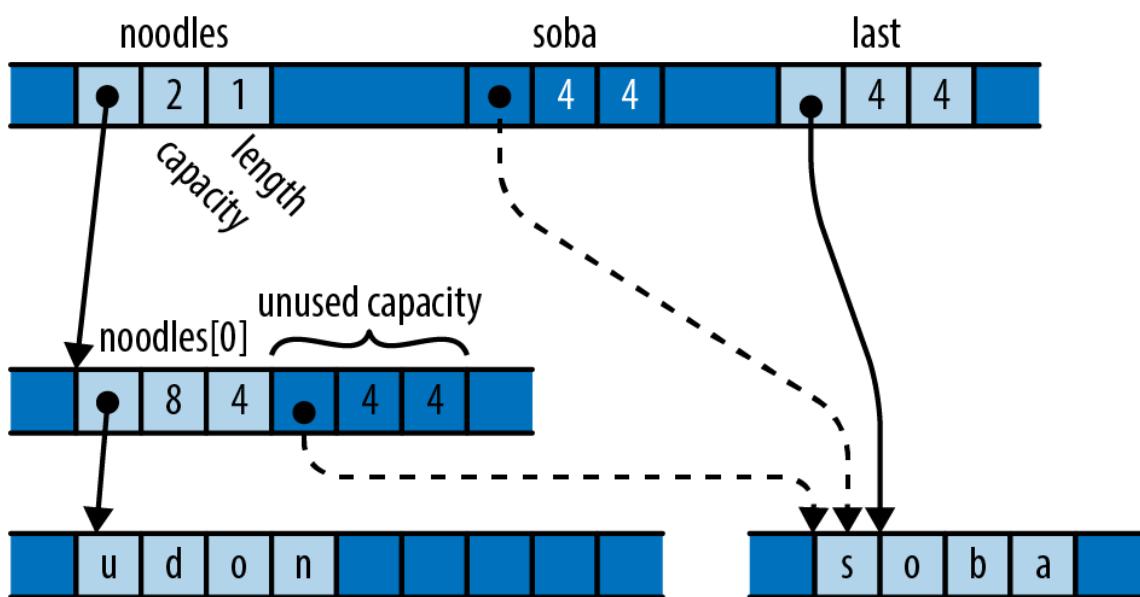
Graphique 22-4. Après avoir poussé la valeur de 's' sur le vecteur soba

Le vecteur a initialisé son espace vide pour posséder la chaîne et incrémenté sa longueur pour la marquer comme un nouvel élément actif. Le vecteur est maintenant le propriétaire de la chaîne ; vous pouvez vous référer à son deuxième élément, et la suppression du vecteur libérerait les deux chaînes. Et n'est plus initialisé. soba

Enfin, considérons ce qui se passe lorsque nous faisons apparaître une valeur à partir du vecteur :

```
last = noodles.pop().unwrap();
```

En mémoire, les choses ressemblent maintenant à [la figure 22-5](#).



Graphique 22-5. Après avoir fait apparaître un élément du vecteur dans `last`

La variable `a` a pris possession de la chaîne. Le vecteur a décrémenté sa longueur pour indiquer que l'espace qui contenait la chaîne n'est plus ini-

tialisé. last

Tout comme avec et précédemment, les trois de , et l'espace libre du vecteur contiennent probablement des motifs de bits identiques. Mais seul est considéré comme possédant la valeur. Traiter l'un ou l'autre des deux autres endroits comme vivant serait une erreur. pot pasta soba last last

La véritable définition d'une valeur initialisée est celle qui est *traitée comme vivante*. L'écriture sur les octets d'une valeur est généralement une partie nécessaire de l'initialisation, mais uniquement parce que cela prépare la valeur à être traitée comme active. Un déplacement et une copie ont tous deux le même effet sur la mémoire ; la différence entre les deux est que, après un déplacement, la source n'est plus traitée comme vivante, alors qu'après une copie, la source et la destination sont en direct.

Rust suit les variables locales qui sont en ligne au moment de la compilation et vous empêche d'utiliser des variables dont les valeurs ont été déplacées ailleurs. Des types comme , , , et ainsi de suite suivent leurs tampons dynamiquement. Si vous implémentez un type qui gère sa propre mémoire, vous devrez faire de même. Vec HashMap Box

Rust fournit deux opérations essentielles pour la mise en œuvre de tels types:

`std::ptr::read(src)`

Déplace une valeur hors de l'emplacement pointé vers, transférant la propriété à l'appelant. L'argument doit être un pointeur brut, où est un type de taille. Après avoir appelé cette fonction, le contenu de n'est pas affecté, mais sauf si c'est , vous devez vous assurer que votre programme les traite comme de la mémoire non initialisée. src src \*const T T \*src T Copy

C'est l'opération derrière . L'effacement d'une valeur appelle à déplacer la valeur hors de la mémoire tampon, puis décrémente la longueur pour marquer cet espace comme capacité non initialisée. Vec::pop read

`std::ptr::write(dest, value)`

Se déplace dans l'emplacement pointé vers, qui doit être de la mémoire non initialisée avant l'appel. Le référent est maintenant pro-

priétaire de la valeur. Ici, doit être un pointeur brut et une valeur, où est un type de taille. `value dest dest *mut T value T T`

C'est l'opération derrière . Pousser une valeur appelle à déplacer la valeur dans l'espace disponible suivant, puis incrémenter la longueur pour marquer cet espace comme un élément valide. `vec::push write`

Les deux sont des fonctions libres, pas des méthodes sur les types de pointeurs bruts.

Notez que vous ne pouvez pas faire ces choses avec l'un des types de pointeurs de sécurité de Rust. Ils exigent tous que leurs référents soient initialisés à tout moment, de sorte que la transformation de la mémoire non initialisée en valeur, ou vice versa, est hors de leur portée. Les pointeurs bruts correspondent à la facture.

La bibliothèque standard fournit également des fonctions permettant de déplacer des tableaux de valeurs d'un bloc de mémoire à un autre :

`std::ptr::copy(src, dst, count)`

Déplace le tableau de valeurs en mémoire à partir de la mémoire à , comme si vous aviez écrit une boucle de et appelle pour les déplacer une à la fois. La mémoire de destination doit être non initialisée avant l'appel, puis la mémoire source n'est pas initialisée. Les arguments et doivent être des pointeurs bruts, et doivent être un . `count src dst read write src dest *const T *mut T count usize`

`ptr. (dst, compte) copy_to`

Une version plus pratique de cela déplace le tableau de valeurs en mémoire à partir de , plutôt que de prendre son point de départ comme argument. `copy count ptr dst`

`std::ptr::copy_nonoverlapping(src, dst, count)`

Comme l'appel correspondant à , sauf que son contrat exige en outre que les blocs de mémoire source et de destination ne se chevauchent pas. Cela peut être légèrement plus rapide que d'appeler . `copy copy`

`ptr.copy_to_nonoverlapping(dst, count)`

Une version plus pratique de , comme . `copy_nonoverlapping copy_to`

Il existe deux autres familles et fonctions, également dans le module: `read` `write` `std::ptr`

#### `read_unaligned`, `write_unaligned`

Ces fonctions sont comme et , sauf que le pointeur n'a pas besoin d'être aligné comme normalement requis pour le type de référent. Ces fonctions peuvent être plus lentes que la plaine et les fonctions. `read` `write` `read` `write`

#### `read_volatile`, `write_volatile`

Ces fonctions sont l'équivalent de lectures et d'écritures volatiles en C ou C++.

## Exemple : GapBuffer

Voici un exemple qui met à profit les fonctions de pointeur brutes que nous venons de décrire.

Supposons que vous écrivez un éditeur de texte et que vous recherchez un type pour représenter le texte. Vous pouvez choisir et utiliser les méthodes et pour insérer et supprimer des caractères au fur et à mesure que l'utilisateur tape. Mais s'ils modifient du texte au début d'un fichier volumineux, ces méthodes peuvent être coûteuses : l'insertion d'un nouveau caractère implique de déplacer tout le reste de la chaîne vers la droite dans la mémoire, et la suppression déplace tout cela vers la gauche. Vous aimeriez que ces opérations courantes soient moins chères. `String insert remove`

L'éditeur de texte Emacs utilise une structure de données simple appelée *tampon d'écart* qui peut insérer et supprimer des caractères en temps constant. Alors qu'un garde toute sa capacité inutilisée à la fin du texte, ce qui fait et bon marché, un tampon d'écart maintient sa capacité inutilisée au milieu du texte, au moment où l'édition a lieu. Cette capacité inutilisée s'appelle *l'écart*. Insérer ou supprimer des éléments à l'écart est bon marché: il vous suffit de réduire ou d'agrandir l'espace selon vos besoins. Vous pouvez déplacer l'espace vers n'importe quel emplacement de votre choix en déplaçant le texte d'un côté de l'espace à l'autre. Lorsque l'espace est vide, vous migrez vers un tampon plus grand. `String push pop`

Bien que l'insertion et la suppression dans un tampon d'écart soient rapides, changer la position à laquelle elles ont lieu implique de déplacer l'écart vers la nouvelle position. Le déplacement des éléments nécessite un temps proportionnel à la distance déplacée. Heureusement, l'activité d'édition typique consiste à apporter un tas de modifications dans un

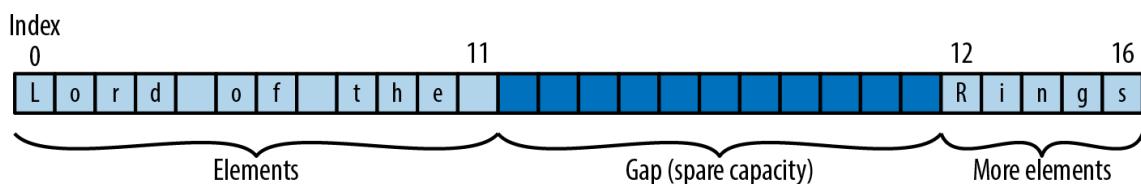
quartier de la mémoire tampon avant de partir et de jouer avec du texte ailleurs.

Dans cette section, nous allons implémenter un tampon d'espace dans Rust. Pour éviter d'être distrait par UTF-8, nous allons faire en sorte que nos valeurs de stockage de tampon directement, mais les principes de fonctionnement seraient les mêmes si nous stockions le texte sous une autre forme. `char`

Tout d'abord, nous allons montrer un tampon d'écart en action. Ce code crée un , y insère du texte, puis déplace le point d'insertion pour qu'il s'assoie juste avant le dernier mot : `GapBuffer`

```
let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

Après avoir exécuté ce code, la mémoire tampon ressemble à ce qui est illustré à [la figure 22-6](#).

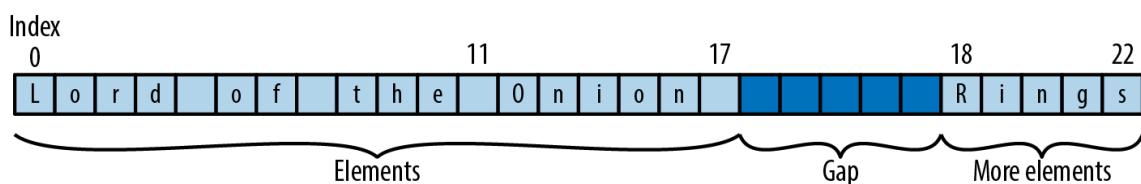


Graphique 22-6. Un tampon d'espace contenant du texte

L'insertion consiste à combler le vide avec un nouveau texte. Ce code ajoute un mot et ruine le film :

```
buf.insert_iter("Onion ".chars());
```

Il en résulte l'état illustré à [la figure 22-7](#).



Graphique 22-7. Un tampon d'espace contenant un peu plus de texte

Voici notre type: `GapBuffer`

```
use std;
use std::ops::Range;
```

```

pub struct GapBuffer<T> {
 // Storage for elements. This has the capacity we need, but its length
 // always remains zero. GapBuffer puts its elements and the gap in
 // `Vec`'s "unused" capacity.
 storage: Vec<T>,
 // Range of uninitialized elements in the middle of `storage`.
 // Elements before and after this range are always initialized.
 gap: Range<usize>
}

```

GapBuffer utilise son champ d'une manière étrange. storage <sup>2</sup> Il ne stocke jamais réellement d'éléments dans le vecteur , ou pas tout à fait. Il appelle simplement à obtenir un bloc de mémoire suffisamment grand pour contenir des valeurs, obtient des pointeurs bruts vers cette mémoire via le vecteur et les méthodes, puis utilise le tampon directement à ses propres fins. La longueur du vecteur reste toujours nulle. Lorsque le est abandonné, le n'essaie pas de libérer ses éléments, car il ne sait pas qu'il en a, mais il libère le bloc de mémoire. C'est ce que veut; il a sa propre implémentation qui sait où se trouvent les éléments vivants et les laisse tomber

correctement. `Vec::with_capacity(n) n as_ptr as_mut_ptr Vec V`  
`ec GapBuffer Drop`

GapBuffer les méthodes les plus simples sont ce à quoi vous vous attendez:

```

impl<T> GapBuffer<T> {
 pub fn new() -> GapBuffer<T> {
 GapBuffer { storage: Vec::new(), gap: 0..0 }
 }

 /// Return the number of elements this GapBuffer could hold without
 /// reallocation.
 pub fn capacity(&self) -> usize {
 self.storage.capacity()
 }

 /// Return the number of elements this GapBuffer currently holds.
 pub fn len(&self) -> usize {
 self.capacity() - self.gap.len()
 }
}

```

```

 /// Return the current insertion position.
 pub fn position(&self) -> usize {
 self.gap.start
 }

 ...
}

}

```

Il nettoie de nombreuses fonctions suivantes pour avoir une méthode utilitaire qui renvoie un pointeur brut vers l'élément tampon à un index donné. Ceci étant Rust, nous finissons par avoir besoin d'une méthode pour les pointeurs et d'une pour . Contrairement aux méthodes précédentes, celles-ci ne sont pas publiques. Suite de ce bloc : `:mut const impl`

```

 /// Return a pointer to the `index`th element of the underlying storage
 /// regardless of the gap.
 ///
 /// Safety: `index` must be a valid index into `self.storage`.
 unsafe fn space(&self, index: usize) -> *const T {
 self.storage.as_ptr().offset(index as isize)
 }

 /// Return a mutable pointer to the `index`th element of the underlying storage,
 /// regardless of the gap.
 ///
 /// Safety: `index` must be a valid index into `self.storage`.
 unsafe fn space_mut(&mut self, index: usize) -> *mut T {
 self.storage.as_mut_ptr().offset(index as isize)
 }
}

```

Pour trouver l'élément à un indice donné, vous devez déterminer si l'indice tombe avant ou après l'écart et ajuster de manière appropriée :

```

 /// Return the offset in the buffer of the `index`th element, taking
 /// the gap into account. This does not check whether index is in range
 /// but it never returns an index in the gap.
 fn index_to_raw(&self, index: usize) -> usize {
 if index < self.gap.start {
 index
 } else {
 index + self.gap.len()
 }
 }
}

```

```
 /// Return a reference to the `index`-th element,
 /// or `None` if `index` is out of bounds.

 pub fn get(&self, index: usize) -> Option<&T> {
 let raw = self.index_to_raw(index);
 if raw < self.capacity() {
 unsafe {
 // We just checked `raw` against self.capacity(),
 // and index_to_raw skips the gap, so this is safe.
 Some(&*self.space(raw))
 }
 } else {
 None
 }
 }
```

Lorsque nous commençons à effectuer des insertions et des suppressions dans une autre partie de la mémoire tampon, nous devons déplacer l'espace vers le nouvel emplacement. Déplacer l'espace vers la droite implique de déplacer les éléments vers la gauche, et vice versa, tout comme la bulle dans un niveau d'esprit se déplace dans une direction lorsque le fluide s'écoule dans l'autre:

```

 distance);
 }

 self.gap = pos .. pos + gap.len();
}
}

```

Cette fonction utilise la méthode pour déplacer les éléments ; exige que la destination ne soit pas initialisée et laisse la source non initialisée. Les plages source et de destination peuvent se chevaucher, mais gèrent correctement ce cas. Étant donné que l'écart est une mémoire non initialisée avant l'appel et que la fonction ajuste la position de l'écart pour couvrir l'espace libéré par la copie, le contrat de la fonction est satisfait.

`std::ptr::copy`

L'insertion et le retrait des éléments sont relativement simples. L'insertion prend plus d'un espace de l'espace pour le nouvel élément, tandis que la suppression déplace une valeur et agrandit l'espace pour couvrir l'espace qu'il occupait auparavant:

```

/// Insert `elt` at the current insertion position,
/// and leave the insertion position after it.
pub fn insert(&mut self, elt: T) {
 if self.gap.len() == 0 {
 self.enlarge_gap();
 }

 unsafe {
 let index = self.gap.start;
 std::ptr::write(self.space_mut(index), elt);
 }
 self.gap.start += 1;
}

/// Insert the elements produced by `iter` at the current insertion
/// position, and leave the insertion position after them.
pub fn insert_iter<I>(&mut self, iterable: I)
 where I: IntoIterator<Item=T>
{
 for item in iterable {
 self.insert(item)
 }
}

```

```
 /// Remove the element just after the insertion position
 /// and return it, or return `None` if the insertion position
 /// is at the end of the GapBuffer.
 pub fn remove(&mut self) -> Option<T> {
 if self.gap.end == self.capacity() {
 return None;
 }

 let element = unsafe {
 std::ptr::read(self.space(self.gap.end))
 };
 self.gap.end += 1;
 Some(element)
 }
}
```

Semblable à la façon dont les utilisations pour push et pour , utilise pour et pour . Et tout comme doit ajuster sa longueur pour maintenir la frontière entre les éléments initialisés et la capacité de réserve, ajuste son écart. Vec std::ptr::write std::ptr::read pop GapBuffer write insert read remove Vec GapBuffer

Lorsque l'écart a été comblé, la méthode doit augmenter le tampon pour acquérir plus d'espace libre. La méthode (la dernière du bloc) gère ceci :

```
:insert enlarge gap impl
```

```

 self.gap.start);

 // Move the elements that fall after the gap.
 let new_gap_end = new.as_mut_ptr().offset(new_gap.end as isize)
 std::ptr::copy_nonoverlapping(self.space(self.gap.end),
 new_gap_end,
 after_gap);
 }

 // This frees the old Vec, but drops no elements,
 // because the Vec's length is zero.
 self.storage = new;
 self.gap = new_gap;
}

```

Alors que doit utiliser pour déplacer des éléments d'avant en arrière dans l'espace, peut utiliser , car il déplace des éléments vers un tampon entièrement

nouveau.set\_position copy enlarge\_gap copy\_nonoverlapping

Le déplacement du nouveau vecteur dans laisse tomber l'ancien vecteur. Comme sa longueur est nulle, l'ancien vecteur croit qu'il n'a pas d'éléments à laisser tomber et libère simplement son tampon. Soigneusement, laisse sa source non initialisée, de sorte que l'ancien vecteur est correct dans cette croyance: tous les éléments sont maintenant la propriété du nouveau vecteur. self.storage copy\_nonoverlapping

Enfin, nous devons nous assurer que la chute d'un laisse tomber tous ses éléments: GapBuffer

```

impl<T> Drop for GapBuffer<T> {
 fn drop(&mut self) {
 unsafe {
 for i in 0 .. self.gap.start {
 std::ptr::drop_in_place(self.space_mut(i));
 }
 for i in self.gap.end .. self.capacity() {
 std::ptr::drop_in_place(self.space_mut(i));
 }
 }
 }
}

```

Les éléments se trouvent avant et après l'écart, nous itérons donc sur chaque région et utilisons la fonction pour laisser tomber chacune d'elles. La fonction est un utilitaire qui se comporte comme , mais ne prend pas la peine de déplacer la valeur vers son appelant (et fonctionne donc sur les types non dimensionnés). Et tout comme dans , au moment où le vecteur est abandonné, son tampon n'est vraiment pas initialisé.

```
std::ptr::drop_in_place drop_in_place drop(std::ptr
::read(ptr)) enlarge_gap self.storage
```

Comme les autres types que nous avons montrés dans ce chapitre, s'assure que ses propres invariants sont suffisants pour s'assurer que le contrat de chaque fonctionnalité dangereuse qu'il utilise est suivi, de sorte qu'aucune de ses méthodes publiques n'a besoin d'être marquée comme dangereuse. implémente une interface sécurisée pour une fonctionnalité qui ne peut pas être écrite efficacement dans du code sécurisé.

```
GapBuffer GapBuffer
```

## Sécurité de panique dans le code dangereux

Dans Rust, les paniques ne peuvent généralement pas provoquer un comportement indéfini; la macro n'est pas une fonctionnalité dangereuse. Mais lorsque vous décidez de travailler avec un code dangereux, la sécurité panique fait partie de votre travail.

```
panic!
```

Considérez la méthode de la section précédente:

```
GapBuffer::remove
```

```
pub fn remove(&mut self) -> Option<T> {
 if self.gap.end == self.capacity() {
 return None;
 }

 let element = unsafe {
 std::ptr::read(self.space(self.gap.end))
 };
 self.gap.end += 1;
 Some(element)
}
```

Appel à déplacer l'élément immédiatement après l'espace hors de la mémoire tampon, laissant derrière lui un espace non initialisé. À ce stade, le est dans un état incohérent: nous avons brisé l'invariant que tous les éléments en dehors de l'écart doivent être initialisés. Heureusement, la dé-

laration suivante agrandit l'écart pour couvrir cet espace, de sorte qu'au moment où nous revenons, l'invariant tient à nouveau. `read` `GapBuffer`

Mais réfléchissez à ce qui se passerait si, après l'appel à `mais` avant `l'ajustement à`, ce code essayait d'utiliser une fonctionnalité qui pourrait paniquer, par exemple, l'indexation d'une tranche. Quitter brusquement la méthode n'importe où entre ces deux actions laisserait le avec un élément non initialisé en dehors de l'espace. Le prochain appel à `pourrait` essayer à nouveau; même en laissant tomber le serait essayer de le laisser tomber. Les deux sont un comportement non défini, car ils accèdent à la mémoire non

initialisée. `read` `self.gap.end` `GapBuffer` `remove` `read` `GapBuffer`

Il est presque inévitable que les méthodes d'un type détendent momentanément les invariants du type pendant qu'ils font leur travail, puis remettent tout en place avant leur retour. Une méthode de panique au milieu pourrait couper court à ce processus de nettoyage, laissant le type dans un état incohérent.

Si le type utilise uniquement du code sécurisé, cette incohérence peut entraîner un mauvais comportement du type, mais elle ne peut pas introduire un comportement indéfini. Mais le code utilisant des fonctionnalités non sécurisées compte généralement sur ses invariants pour répondre aux contrats de ces fonctionnalités. Les invariants brisés conduisent à des contrats rompus, ce qui conduit à un comportement indéfini.

Lorsque vous travaillez avec des fonctionnalités dangereuses, vous devez prendre des précautions particulières pour identifier ces régions sensibles du code où les invariants sont temporairement détendus, et vous assurer qu'ils ne font rien qui pourrait paniquer.

## Réinterprétation de la mémoire avec les unions

Rust fournit de nombreuses abstractions utiles, mais en fin de compte, le logiciel que vous écrivez ne fait que pousser des octets. Les unions sont l'une des fonctionnalités les plus puissantes de Rust pour manipuler ces octets et choisir comment ils sont interprétés. Par exemple, toute collection de 32 bits (4 octets) peut être interprétée comme un entier ou comme un nombre à virgule flottante. L'une ou l'autre interprétation est valide,

bien que l'interprétation de données destinées à l'un comme à l'autre entraînera probablement des absurdités.

Une union représentant une collection d'octets pouvant être interprétée comme un nombre entier ou un nombre à virgule flottante s'écrirait comme suit :

```
union FloatOrInt {
 f: f32,
 i: i32,
}
```

Il s'agit d'une union avec deux champs, et . Ils peuvent être affectés à tout comme les champs d'une structure, mais lors de la construction d'une union, contrairement à une structure, vous devez en choisir exactement une. Là où les champs d'une struct se réfèrent à différentes positions en mémoire, les champs d'une union se réfèrent à différentes interprétations de la même séquence de bits. Affecter à un champ différent signifie simplement écraser tout ou partie de ces bits, conformément à un type approprié. Ici, fait référence à une seule plage de mémoire 32 bits, qui stocke d'abord codée sous la forme d'un simple entier, puis sous la forme d'un nombre à virgule flottante IEEE 754. Dès qu'elle est écrite, la valeur précédemment écrite dans le est écrasée : f i one 1 1.0 f FloatOrInt

```
let mut one = FloatOrInt { i: 1 };
assert_eq!(unsafe { one.i }, 0x00_00_00_01);
one.f = 1.0;
assert_eq!(unsafe { one.i }, 0x3F_80_00_00);
```

Pour la même raison, la taille d'une union est déterminée par son plus grand champ. Par exemple, cette union a une taille de 64 bits, même si elle n'est qu'un : SmallOrLarge::s bool

```
union SmallOrLarge {
 s: bool,
 l: u64
}
```

Bien que la construction d'une union ou l'affectation à ses champs soit totalement sûre, la lecture à partir de n'importe quel champ d'une union est toujours dangereuse :

```
let u = SmallOrLarge { l: 1337 };
println!("{}", unsafe {u.l}); // prints 1337
```

En effet, contrairement aux enums, les syndicats n'ont pas d'étiquette. Le compilateur n'ajoute aucun bit supplémentaire pour distinguer les variantes. Il n'y a aucun moyen de dire au moment de l'exécution si a est destiné à être interprété comme un ou un , à moins que le programme n'ait un contexte supplémentaire. `SmallOrLarge u64 bool`

Il n'y a pas non plus de garantie intégrée que le modèle de bits d'un champ donné est valide. Par exemple, écrire dans le champ d'une valeur écrasera son champ, créant un modèle de bits qui ne signifie certainement rien d'utile et qui n'est probablement pas valide. Par conséquent, bien que l'écriture dans des champs d'union soit sûre, chaque lecture nécessite . La lecture de n'est autorisée que lorsque les bits du champ forment un fichier valide ; sinon, il s'agit d'un comportement non défini. `SmallOrLarge l s bool unsafe u.s s bool`

Avec ces restrictions à l'esprit, les unions peuvent être un moyen utile de réinterpréter temporairement certaines données, en particulier lors de calculs sur la représentation des valeurs plutôt que sur les valeurs elles-mêmes. Par exemple, le type mentionné précédemment peut facilement être utilisé pour imprimer les bits individuels d'un nombre à virgule flottante, même s'il n'implémente pas le formateur

```
:FloatOrInt f32 Binary
```

```
let float = FloatOrInt { f: 31337.0 };
// prints 10001101111010011010010000000000
println!("{:b}", unsafe { float.i });
```

Bien que ces exemples simples fonctionneront presque certainement comme prévu sur n'importe quelle version du compilateur, il n'y a aucune garantie qu'un champ commence à un endroit spécifique à moins qu'un attribut ne soit ajouté à la définition indiquant au compilateur comment disposer les données en mémoire. L'ajout de l'attribut garantit que tous les champs commencent au décalage 0, plutôt que là où le compilateur le souhaite. Avec cette garantie en place, le comportement d'écrasement peut être utilisé pour extraire des bits individuels, comme le bit de signe d'un entier : `union #[repr(C)]`

```

#[repr(C)]
union SignExtractor {
 value: i64,
 bytes: [u8; 8]
}

fn sign(int: i64) -> bool {
 let se = SignExtractor { value: int};
 println!("{:b} ({:?}", unsafe { se.value }, unsafe { se.bytes });
 unsafe { se.bytes[7] >= 0b10000000 }
}

assert_eq!(sign(-1), true);
assert_eq!(sign(1), false);
assert_eq!(sign(i64::MAX), false);
assert_eq!(sign(i64::MIN), true);

```

Ici, le bit de signe est le bit le plus significatif de l'octet le plus significatif. Étant donné que les processeurs x86 sont peu endiens, l'ordre de ces octets est inversé ; l'octet le plus significatif n'est pas , mais . Normalement, ce n'est pas quelque chose que le code Rust doit gérer, mais parce que ce code fonctionne directement avec la représentation en mémoire du , ces détails de bas niveau deviennent importants.

Parce que les syndicats ne peuvent pas dire comment supprimer leur contenu, tous leurs champs doivent être . Cependant, si vous devez simplement stocker un dans un syndicat, il existe une solution de contournement; consultez la documentation standard de la bibliothèque pour . Copy String std::mem::ManuallyDrop

## Syndicats correspondants

La correspondance sur une union rust est similaire à la correspondance sur une structure, sauf que chaque motif doit spécifier exactement un champ :

```

unsafe {
 match u {
 SmallOrLarge { s: true } => { println!("boolean true"); }
 SmallOrLarge { l: 2 } => { println!("integer 2"); }
 _ => { println!("something else"); }
 }
}

```

```
 }
}
```

Un bras qui correspond à une variante d'union sans spécifier de valeur réussira toujours. Le code suivant provoquera un comportement non défini si le dernier champ écrit de était : `match u u.i`

```
// Undefined behavior!
unsafe {
 match u {
 FloatOrInt { f } => { println!("float {}", f) },
 // warning: unreachable pattern
 FloatOrInt { i } => { println!("int {}", i) }
 }
}
```

## Syndicats emprunteurs

Emprunter un champ d'un syndicat emprunte l'ensemble du syndicat. Cela signifie que, conformément aux règles d'emprunt normales, emprunter un champ comme mutable empêche tout emprunt supplémentaire sur celui-ci ou sur d'autres champs, et emprunter un champ comme immuable signifie qu'il ne peut y avoir aucun emprunt mutable sur aucun champ.

Comme nous le verrons dans le chapitre suivant, Rust vous aide à créer des interfaces sécurisées non seulement pour votre propre code dangereux, mais également pour le code écrit dans d'autres langages. Dangereux est, comme son nom l'indique, lourd, mais utilisé avec soin, il peut vous permettre de construire un code hautement performant qui conserve les garanties dont jouissent les programmeurs Rust.

<sup>1</sup> Eh bien, c'est un classique d'où nous venons.

<sup>2</sup> Il existe de meilleures façons de gérer cela en utilisant le type de la caisse interne du compilateur, mais cette caisse est toujours instable. `RawVec alloc`



# Chapitre 23. Fonctions étrangères

*Cyberespace. Complexité impensable. Les lignes de lumière variaient dans le non-espace de l'esprit, les amas et les constellations de données. Comme les lumières de la ville, reculant...*

—William Gibson, *neuromancien*

Tragiquement, tous les programmes dans le monde ne sont pas écrits en Rust. Il existe de nombreuses bibliothèques et interfaces critiques implémentées dans d'autres langages que nous aimerais pouvoir utiliser dans nos programmes Rust. *L'interface de fonction étrangère* (FFI) de Rust permet au code Rust d'appeler des fonctions écrites en C et, dans certains cas, en C++. Étant donné que la plupart des systèmes d'exploitation offrent des interfaces C, l'interface de fonction étrangère de Rust permet un accès immédiat à toutes sortes d'installations de bas niveau.

Dans ce chapitre, nous allons écrire un programme lié à , une bibliothèque C pour travailler avec le système de contrôle de version Git. Tout d'abord, nous allons montrer ce que c'est que d'utiliser les fonctions C directement à partir de Rust, en utilisant les fonctionnalités dangereuses démontrées dans le chapitre précédent. Ensuite, nous montrerons comment construire une interface sûre pour , en s'inspirant de la caisse open source, qui fait exactement cela. `libgit2 libgit2 git2-rs`

Nous supposerons que vous êtes familier avec C et la mécanique de compilation et de liaison de programmes C. L'utilisation de C++ est similaire. Nous supposerons également que vous êtes un peu familier avec le système de contrôle de version Git.

Il existe des caisses Rust pour communiquer avec de nombreux autres langages, y compris Python, JavaScript, Lua et Java. Nous n'avons pas de place pour les couvrir ici, mais en fin de compte, toutes ces interfaces sont construites à l'aide de l'interface de fonction étrangère C, donc ce chapitre devrait vous donner une longueur d'avance, quelle que soit la langue avec laquelle vous devez travailler.

## Recherche de représentations de données communes

Le dénominateur commun de Rust et C est le langage machine, donc pour anticiper à quoi ressemblent les valeurs Rust du code C, ou vice versa,

vous devez prendre en compte leurs représentations au niveau de la machine. Tout au long du livre, nous avons tenu à montrer comment les valeurs sont réellement représentées en mémoire, vous avez donc probablement remarqué que les mondes de données de C et Rust ont beaucoup en commun: un Rust et un C sont identiques, par exemple, et les structs sont fondamentalement la même idée dans les deux langues. Pour établir une correspondance entre les types Rust et C, nous allons commencer par les primitives, puis nous nous frayer un chemin jusqu'à des types plus compliqués.

```
u8 size_t
```

Compte tenu de son utilisation principale en tant que langage de programmation de systèmes, C a toujours été étonnamment lâche sur les représentations de ses types: `an` est généralement long de 32 bits, mais pourrait être plus long, ou aussi court que 16 bits; `a` C peut être signé ou non signé; et ainsi de suite. Pour faire face à cette variabilité, le module de Rust définit un ensemble de types rust qui sont garantis d'avoir la même représentation que certains types C ([tableau 23-1](#)). Ceux-ci couvrent les types d'entiers et de caractères

```
int char std::os::raw
```

| Type C                  | Correspondant std::os::raw type |
|-------------------------|---------------------------------|
| short                   | c_short                         |
| int                     | c_int                           |
| long                    | c_long                          |
| long long               | c_longlong                      |
| unsigned short          | c_ushort                        |
| unsigned , unsigned int | c_uint                          |
| unsigned long           | c_ulong                         |
| unsigned long long      | c_ulonglong                     |
| char                    | c_char                          |
| signed char             | c_schar                         |
| unsigned char           | c_uchar                         |
| float                   | c_float                         |
| double                  | c_double                        |
| void * , const void *   | *mut c_void , *const c_void     |

Quelques remarques sur [le tableau 23-1](#) :

- À l'exception de , tous les types rust ici sont des alias pour certains types rust primitifs : , par exemple, est soit ou . `c_void` `c_char` `i8` `u8`
- Un Rust est équivalent à un C ou C++. `bool` `bool`
- Le type 32 bits de Rust n'est pas l'analogue de , dont la largeur et l'encodage varient d'une implémentation à l'autre. Le type de C est plus proche, mais son encodage n'est toujours pas garanti d'être `Unicode.char` `wchar_t` `char32_t`
- Les primitives et les types de Rust ont les mêmes représentations que les C et. `usize` `isize` `size_t` `ptrdiff_t`

- Les pointeurs C et C++ et les références C++ correspondent aux types de pointeurs bruts de Rust, et `. *mut T *const T`
- Techniquement, la norme C permet aux implémentations d'utiliser des représentations pour lesquelles Rust n'a pas de type correspondant : entiers 36 bits, représentations de signe et de magnitude pour les valeurs signées, etc. En pratique, sur chaque plate-forme sur laquelle Rust a été porté, chaque type d'entier C commun a une correspondance dans Rust.

Pour définir des types rust struct compatibles avec les structS C, vous pouvez utiliser l'attribut. Placer au-dessus d'une définition struct demande à Rust de disposer les champs de la struct en mémoire de la même manière qu'un compilateur C présenterait le type C struct analogue. Par exemple, le fichier d'en-tête `git2/errors.h` de devient définit la structure C suivante pour fournir des détails sur une erreur précédemment signalée

```
:#[repr(C)] #[repr(C)] libgit2
```

```
typedef struct {
 char *message;
 int klass;
} git_error;
```

Vous pouvez définir un type Rust avec une représentation identique comme suit :

```
use std::os::raw::{c_char, c_int};

#[repr(C)]
pub struct git_error {
 pub message: *const c_char,
 pub klass: c_int
}
```

L'attribut affecte uniquement la disposition de la structure elle-même, pas les représentations de ses champs individuels, donc pour correspondre à la structure C, chaque champ doit également utiliser le type C: pour , pour , et ainsi de suite. `#[repr(C)] *const c_char char * c_int int`

Dans ce cas particulier, l'attribut ne modifie probablement pas la disposition de . Il n'y a vraiment pas beaucoup de façons intéressantes de disposer un pointeur et un entier. Mais alors que C et C++ garantissent que les membres d'une structure apparaissent en mémoire dans l'ordre dans lequel ils sont déclarés, chacun à une adresse distincte, Rust réorganise

les champs pour minimiser la taille globale de la structure, et les types de taille zéro ne prennent pas de place. L'attribut indique à Rust de suivre les règles de C pour le type donné. `#[repr(C)] git_error #[repr(C)]`

Vous pouvez également utiliser pour contrôler la représentation des énumérations de style C : `#[repr(C)]`

```
#[repr(C)]
#[allow(non_camel_case_types)]
enum git_error_code {
 GIT_OK = 0,
 GIT_ERROR = -1,
 GIT_ENOTFOUND = -3,
 GIT_EEXISTS = -4,
 ...
}
```

Normalement, Rust joue à toutes sortes de jeux lorsqu'il choisit comment représenter les enums. Par exemple, nous avons mentionné l'astuce que Rust utilise pour stocker en un seul mot (si est dimensionné). Sans , Rust utiliserait un seul octet pour représenter l'enum ; avec , Rust utilise une valeur de la taille d'un C , tout comme C. Option<&T> T #

```
[repr(C)] git_error_code #[repr(C)] int
```

Vous pouvez également demander à Rust de donner à un enum la même représentation qu'un type entier. Commencer la définition précédente par vous donnerait un type 16 bits avec la même représentation que l'énumération C++ suivante : `#[repr(i16)]`

```
#include <stdint.h>

enum git_error_code: int16_t {
 GIT_OK = 0,
 GIT_ERROR = -1,
 GIT_ENOTFOUND = -3,
 GIT_EEXISTS = -4,
 ...
};
```

Comme nous l'avons mentionné précédemment, cela s'applique également aux syndicats. Les champs d'unions commencent toujours au premier bit de la mémoire de l'union, c'est-à-dire l'index 0. `#[repr(C)] #[repr(C)]`

Supposons que vous ayez une structure C qui utilise une union pour contenir des données et une valeur de balise pour indiquer quel champ de

l'union doit être utilisé, semblable à un enum de Rust.

```
enum tag {
 FLOAT = 0,
 INT = 1,
};

union number {
 float f;
 short i;
};

struct tagged_number {
 tag t;
 number n;
};
```

Le code Rust peut interagir avec cette structure en s'appliquant aux types enum, structure et union, et en utilisant une instruction qui sélectionne un champ d'union dans une structure plus grande en fonction de la balise :#[repr(C)] match

```
#[repr(C)]
enum Tag {
 Float = 0,
 Int = 1
}

#[repr(C)]
union FloatOrInt {
 f: f32,
 i: i32,
}

#[repr(C)]
struct Value {
 tag: Tag,
 union: FloatOrInt
}

fn is_zero(v: Value) -> bool {
 use self::Tag::*;

 unsafe {
 match v {
 Value { tag: Int, union: FloatOrInt { i: 0 } } => true,
 Value { tag: Float, union: FloatOrInt { f: num } } => (num ==
 _ => false
 }
 }
}
```

```
}
```

Même les structures complexes peuvent être facilement utilisées à travers la frontière FFI en utilisant ce type de technique.

Passer des cordes entre Rust et C est un peu plus difficile. C représente une chaîne sous forme de pointeur vers un tableau de caractères, terminé par un caractère nul. Rust, d'autre part, stocke explicitement la longueur d'une chaîne, soit en tant que champ de `a`, soit en tant que deuxième mot d'une référence grasse. Les chaînes rust ne sont pas terminées par une valeur `NULL` ; en fait, ils peuvent inclure des caractères nuls dans leur contenu, comme tout autre caractère. `String &str`

Cela signifie que vous ne pouvez pas emprunter une chaîne Rust en tant que chaîne C : si vous passez un pointeur de code C dans une chaîne Rust, il pourrait confondre un caractère null incorporé avec la fin de la chaîne ou s'exécuter à la fin à la recherche d'un null de terminaison qui n'est pas là. En allant dans l'autre sens, vous pourrez peut-être emprunter une chaîne C comme un Rust , tant que son contenu est bien formé UTF-8. `&str`

Cette situation oblige effectivement Rust à traiter les chaînes C comme des types entièrement distincts de `et` . Dans le module, les types `et` représentent des tableaux d'octets détenus et empruntés à terminaison `NULL`. Par rapport à `et` , les méthodes sur `et` sont assez limitées, limitées à la construction et à la conversion à d'autres types. Nous montrerons ces types en action dans la section

suivante. `String &str std::ffi CString CStr String str CString CStr`

## Déclaration de fonctions et de variables étrangères

Un bloc déclare des fonctions ou des variables définies dans une autre bibliothèque avec laquelle l'exécutable Rust final sera lié. Par exemple, sur la plupart des plates-formes, chaque programme Rust est lié à la bibliothèque C standard, de sorte que nous pouvons informer Rust de la fonction de la bibliothèque C comme ceci: `extern strlen`

```
use std::os::raw::c_char;
```

```
extern {
```

```
fn strlen(s: *const c_char) -> usize;
}
```

Cela donne à Rust le nom et le type de la fonction, tout en laissant la définition être liée plus tard.

Rust suppose que les fonctions déclarées à l'intérieur des blocs utilisent des conventions C pour passer des arguments et accepter des valeurs de retour. Ils sont définis comme des fonctions. Ce sont les bons choix pour : il s'agit bien d'une fonction C, et sa spécification en C nécessite que vous lui passiez un pointeur valide vers une chaîne correctement terminée, qui est un contrat que Rust ne peut pas appliquer. (Presque toutes les fonctions qui prennent un pointeur brut doivent être : rust sûr peut construire des pointeurs bruts à partir d'entiers arbitraires, et le déréférencement d'un tel pointeur serait un comportement indéfini.) **extern unsafe strlen unsafe**

Avec ce bloc, nous pouvons appeler comme n'importe quelle autre fonction Rust, bien que son type le donne en tant que touriste: **extern strlen**

```
use std::ffi::CString;

let rust_str = "I'll be back";
let null_terminated = CString::new(rust_str).unwrap();
unsafe {
 assert_eq!(strlen(null_terminated.as_ptr()), 12);
}
```

La fonction génère une chaîne C terminée par une valeur NULL. Il vérifie d'abord son argument pour les caractères nuls incorporés, car ceux-ci ne peuvent pas être représentés dans une chaîne C, et renvoie une erreur s'il en trouve (d'où la nécessité du résultat). Sinon, il ajoute un octet nul à la fin et renvoie une propriété des caractères résultants. **CString::new unwrap CString**

Le coût de dépend du type de passe-t-il. Il accepte tout ce qui implémente . La transmission d'un implique une allocation et une copie, car la conversion en construit une copie allouée au tas de la chaîne pour que le vecteur soit propriétaire. Mais passer une valeur by consomme simplement la chaîne et prend en charge son tampon, donc à moins que l'ajout du caractère null ne force le tampon à être redimensionné, la conversion ne nécessite aucune copie du texte ou allocation. **CString::new Into<Vec<u8>> &str Vec<u8> String**

`CString` déréférence à , dont la méthode renvoie un pointage au début de la chaîne. C'est le type qui s'attend. Dans l'exemple, exécute la chaîne, recherche le caractère null qui s'y trouve et renvoie la longueur, sous la forme d'un nombre d'octets.

```
CStr as_ptr *const
c_char strlen strlen CString::new
```

Vous pouvez également déclarer des variables globales dans des blocs. Les systèmes POSIX ont une variable globale nommée qui contient les valeurs des variables d'environnement du processus. En C, il est déclaré

```
:extern environ
```

```
extern char **environ;
```

Dans Rust, vous diriez :

```
use std::ffi::CStr;
use std::os::raw::c_char;

extern {
 static environ: *mut *mut c_char;
}
```

Pour imprimer le premier élément de l'environnement, vous pouvez écrire :

```
unsafe {
 if !environ.is_null() && !(*environ).is_null() {
 let var = CStr::from_ptr(*environ);
 println!("first environment variable: {}",
 var.to_string_lossy())
 }
}
```

Après s'être assuré qu'il a un premier élément, le code appelle à construire un qui l'emprunte. La méthode renvoie un : si la chaîne C contient UTF-8 bien formé, l'emprunte son contenu sous la forme d'un , sans inclure l'octet null de fin. Sinon, fait une copie du texte dans le tas, remplace les séquences UTF-8 mal formées par le caractère de remplacement Unicode officiel, et construit une propriété à partir de cela. Quoi qu'il en soit, le résultat implémente , de sorte que vous pouvez l'imprimer avec le paramètre

```
format.environ CStr::from_ptr CStr to_string_lossy Cow<str> C
ow &str to_string_lossy ⚡ Cow Display {}
```

# Utilisation des fonctions des bibliothèques

Pour utiliser les fonctions fournies par une bibliothèque particulière, vous pouvez placer un attribut au-dessus du bloc qui nomme la bibliothèque avec laquelle Rust doit lier l'exécutable. Par exemple, voici un programme qui appelle les méthodes d'initialisation et d'arrêt de 'git', mais ne fait rien d'autre:

```
#[link] extern libgit2
```

```
use std::os::raw::c_int;

#[link(name = "git2")]
extern {
 pub fn git_libgit2_init() -> c_int;
 pub fn git_libgit2_shutdown() -> c_int;
}

fn main() {
 unsafe {
 git_libgit2_init();
 git_libgit2_shutdown();
 }
}
```

Le bloc déclare les fonctions externes comme précédemment. L'attribut laisse une note dans la caisse à l'effet que, lorsque Rust crée l'exécutable final ou la bibliothèque partagée, il doit être lié à la bibliothèque. Rust utilise l'éditeur de liens système pour créer des exécutables; sous Unix, cela passe l'argument sur la ligne de commande de l'éditeur de liens ; sous Windows, il passe `.extern #[link(name = "git2")] git2 -lgit2 git2.LIB`

`#[link]` les attributs fonctionnent également dans les caisses de bibliothèque. Lorsque vous créez un programme qui dépend d'autres caisses, Cargo rassemble les notes de lien de l'ensemble du graphique de dépendance et les inclut toutes dans le lien final.

Dans cet exemple, si vous souhaitez suivre sur votre propre machine, vous devrez construire pour vous-même. Nous avons utilisé [libgit2](#) version 0.25.1. Pour compiler, vous devrez installer l'outil de génération CMake et le langage Python; nous avons utilisé [CMake](#) version 3.8.0 et [Python](#) version 2.7.13. `libgit2 libgit2`

Les instructions complètes pour la construction sont disponibles sur son site Web, mais elles sont assez simples pour que nous leur montrions l'essentiel ici. Sous Linux, supposons que vous avez déjà décompressé la source de la bibliothèque dans le répertoire `/home/jimb/libgit2-0.25.1`

`:libgit2`

```
$ cd /home/jimb/libgit2-0.25.1
$ mkdir build
$ cd build
$ cmake ..
$ cmake --build .
```

Sous Linux, cela produit une bibliothèque partagée `/home/jimb/libgit2-0.25.1/build/libgit2.so.0.25.1` avec le nid habituel de liens symboliques pointant vers elle, dont une nommée `libgit2.so`. Sous macOS, les résultats sont similaires, mais la bibliothèque est nommée `libgit2.dylib`.

Sur Windows, les choses sont également simples. Supposons que vous avez décompressé la source dans le répertoire `C:\Users\JimB\libgit2-0.25.1`. Dans une invite de commandes Visual Studio :

```
> cd C:\Users\JimB\libgit2-0.25.1
> mkdir build
> cd build
> cmake -A x64 ..
> cmake --build .
```

Ce sont les mêmes commandes que celles utilisées sous Linux, sauf que vous devez demander une version 64 bits lorsque vous exécutez CMake la première fois pour correspondre à votre compilateur Rust. (Si vous avez installé la chaîne d'outils Rust 32 bits, vous devez omettre l'indicateur à la première commande.) Cela produit une bibliothèque d'importation `git2.LIB` et une bibliothèque de liens `dynamiques git2.DLL`, tous deux dans le répertoire `C:\Users\JimB\libgit2-0.25.1\build\Debug`. (Les instructions restantes sont affichées pour Unix, sauf lorsque Windows est sensiblement différent.) `-A x64 cmake`

Créez le programme Rust dans un répertoire séparé :

```
$ cd /home/jimb
$ cargo new --bin git-toy
 Created binary (application) `git-toy` package
```

Prenez le code montré plus haut et mettez-le dans `src/main.rs`. Naturellement, si vous essayez de construire cela, Rust n'a aucune idée de l'endroit

où trouver le vous avez construit: libgit2

```
$ cd git-toy
$ cargo run
 Compiling git-toy v0.1.0 (/home/jimb/git-toy)
error: linking with `cc` failed: exit status: 1
|
= note: /usr/bin/ld: error: cannot find -lgit2
 src/main.rs:11: error: undefined reference to 'git_libgit2_init'
 src/main.rs:12: error: undefined reference to 'git_libgit2_shutdown'
 collect2: error: ld returned 1 exit status

error: could not compile `git-toy` due to previous error
```

Vous pouvez indiquer à Rust où rechercher des bibliothèques en écrivant un *script de build*, du code Rust que Cargo compile et exécute au moment de la build. Les scripts de build peuvent faire toutes sortes de choses : générer du code dynamiquement, compiler du code C à inclure dans la caisse, etc. Dans ce cas, tout ce dont vous avez besoin est d'ajouter un chemin de recherche de bibliothèque à la commande link de l'exécutable. Lorsque Cargo exécute le script de build, il analyse la sortie du script de build pour obtenir des informations de ce type, de sorte que le script de build doit simplement imprimer la bonne magie sur sa sortie standard.

Pour créer votre script de build, ajoutez un fichier nommé *build.rs* dans le même répertoire que le fichier *Cargo.toml*, avec le contenu suivant :

```
fn main() {
 println!(r#"cargo:rustc-link-search=native=/home/jimb/libgit2-0.25.1/libgit2"#)
}
```

C'est la bonne voie pour Linux; sous Windows, vous devez modifier le chemin d'accès suivant le texte en . (Nous coupons quelques coins pour garder cet exemple simple; dans une application réelle, vous devriez éviter d'utiliser des chemins absous dans votre script de génération. Nous citons la documentation qui montre comment le faire correctement à la fin de cette section.) native= C:\Users\JimB\libgit2-0.25.1\build\Debug

Maintenant, vous pouvez presque exécuter le programme. Sur macOS, cela peut fonctionner immédiatement; sur un système Linux, vous verrez probablement quelque chose comme ceci:

```
$ cargo run
 Compiling git-toy v0.1.0 (/tmp/rustbook-transcript-tests/git-toy)
 Finished dev [unoptimized + debuginfo] target(s)
 Running `target/debug/git-toy`
target/debug/git-toy: error while loading shared libraries:
libgit2.so.25: cannot open shared object file: No such file or directory
```

Cela signifie que, bien que Cargo ait réussi à lier l'exécutable à la bibliothèque, il ne sait pas où trouver la bibliothèque partagée au moment de l'exécution. Windows signale cet échec en faisant apparaître une boîte de dialogue. Sous Linux, vous devez définir la variable d'environnement

: LD\_LIBRARY\_PATH

```
$ export LD_LIBRARY_PATH=/home/jimb/libgit2-0.25.1/build:$LD_LIBRARY_PATH
$ cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
 Running `target/debug/git-toy`
```

Sur macOS, vous devrez peut-être définir à la place. DYLD\_LIBRARY\_PATH

Sous Windows, vous devez définir la variable d'environnement : PATH

```
> set PATH=C:\Users\JimB\libgit2-0.25.1\build\Debug;%PATH%
> cargo run
 Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
 Running `target/debug/git-toy`
```

Naturellement, dans une application déployée, vous voudriez éviter d'avoir à définir des variables d'environnement juste pour trouver le code de votre bibliothèque. Une alternative consiste à lier statiquement la bibliothèque C à votre caisse. Cela copie les fichiers objet de la bibliothèque dans le fichier *.rlib* de la caisse, à côté des fichiers objet et des métadonnées du code Rust de la caisse. L'ensemble de la collection participe ensuite au lien final.

C'est une convention Cargo qu'une caisse qui donne accès à une bibliothèque C doit être nommée , où est le nom de la bibliothèque C. Une caisse ne doit contenir rien d'autre que la bibliothèque liée statiquement et les modules Rust contenant des blocs et des définitions de type. Les interfaces de niveau supérieur appartiennent alors à des caisses qui dépendent de la caisse. Cela permet à plusieurs caisses en amont de dépendre de la même caisse, en supposant qu'il existe une seule version de la caisse

qui répond aux besoins de chacun. LIB-sys LIB -sys extern -sys -  
sys -sys

Pour plus de détails sur la prise en charge par Cargo des scripts de build et des liens avec les bibliothèques système, [consultez la documentation cargo en ligne](#). Il montre comment éviter les chemins absous dans les scripts de construction, contrôler les indicateurs de compilation, utiliser des outils tels que , etc. La caisse fournit également de bons exemples à imiter; son script de build gère certaines situations complexes. pkg-config git2-rs

## Une interface brute vers libgit2

Comprendre comment utiliser correctement se décompose en deux questions: libgit2

- Que faut-il pour utiliser les fonctions dans Rust? libgit2
- Comment pouvons-nous construire une interface Rust sûre autour d'eux?

Nous répondrons à ces questions une à la fois. Dans cette section, nous allons écrire un programme qui est essentiellement un bloc géant rempli de code Rust non idiomatique, reflétant le choc des systèmes de type et des conventions inhérent au mélange des langages. Nous appellerons cela l'interface *brute*. Le code sera désordonné, mais il indiquera clairement toutes les étapes qui doivent se produire pour que le code Rust puisse utiliser .unsafe libgit2

Ensuite, dans la section suivante, nous allons construire une interface sécurisée qui met les types de Rust à utiliser en appliquant les règles imposées à ses utilisateurs. Heureusement, il s'agit d'une bibliothèque C exceptionnellement bien conçue, de sorte que les questions que les exigences de sécurité de Rust nous obligent à poser ont toutes de très bonnes réponses, et nous pouvons construire une interface Rust idiomatique sans fonctions. libgit2 libgit2 libgit2 unsafe

Le programme que nous allons écrire est très simple : il prend un chemin comme argument de ligne de commande, y ouvre le référentiel Git et imprime la validation de tête. Mais cela suffit à illustrer les stratégies clés pour construire des interfaces Rust sûres et idiomatiques.

Pour l'interface brute, le programme finira par avoir besoin d'une collection un peu plus grande de fonctions et de types que ce que nous utilisons auparavant, il est donc logique de déplacer le bloc dans son propre

module. Nous allons créer un fichier nommé `raw.rs` dans `git-toy/src` dont le contenu est le suivant:

```
libgit2 extern
```

```
#![allow(non_camel_case_types)]

use std::os::raw::{c_int, c_char, c_uchar};

#[link(name = "git2")]
extern {
 pub fn git_libgit2_init() -> c_int;
 pub fn git_libgit2_shutdown() -> c_int;
 pub fn giterr_last() -> *const git_error;

 pub fn git_repository_open(out: *mut *mut git_repository,
 path: *const c_char) -> c_int;
 pub fn git_repository_free(repo: *mut git_repository);

 pub fn git_reference_name_to_id(out: *mut git_oid,
 repo: *mut git_repository,
 reference: *const c_char) -> c_int;

 pub fn git_commit_lookup(out: *mut *mut git_commit,
 repo: *mut git_repository,
 id: *const git_oid) -> c_int;

 pub fn git_commit_author(commit: *const git_commit) -> *const git_sig;
 pub fn git_commit_message(commit: *const git_commit) -> *const c_char;
 pub fn git_commit_free(commit: *mut git_commit);
}

#[repr(C)] pub struct git_repository { _private: [u8; 0] }
#[repr(C)] pub struct git_commit { _private: [u8; 0] }

#[repr(C)]
pub struct git_error {
 pub message: *const c_char,
 pub klass: c_int
}

pub const GIT_OID_RAWSZ: usize = 20;

#[repr(C)]
pub struct git_oid {
 pub id: [c_uchar; GIT_OID_RAWSZ]
}

pub type git_time_t = i64;
```

```

#[repr(C)]
pub struct git_time {
 pub time: git_time_t,
 pub offset: c_int
}

#[repr(C)]
pub struct git_signature {
 pub name: *const c_char,
 pub email: *const c_char,
 pub when: git_time
}

```

Chaque élément ici est modelé sur une déclaration des propres fichiers d'en-tête de . Par exemple, *libgit2-0.25.1/include/git2/repository.h* inclut cette déclaration : libgit2

```
extern int git_repository_open(git_repository **out, const char *path);
```

Cette fonction tente d'ouvrir le référentiel Git à l'adresse . Si tout se passe bien, il crée un objet et stocke un pointeur vers celui-ci à l'emplacement indiqué par . La déclaration de rouille équivalente est la suivante

```
:path git_repository out
```

```
pub fn git_repository_open(out: *mut *mut git_repository,
 path: *const c_char) -> c_int;
```

Les fichiers d'en-tête publics définissent le type comme un typedef pour un type struct incomplet : libgit2 git\_repository

```
typedef struct git_repository git_repository;
```

Étant donné que les détails de ce type sont privés pour la bibliothèque, les en-têtes publics ne définissent jamais, ce qui garantit que les utilisateurs de la bibliothèque ne peuvent jamais créer eux-mêmes une instance de ce type. Un analogue possible à un type de structure incomplet dans Rust est le suivant: struct git\_repository

```
#[repr(C)] pub struct git_repository { _private: [u8; 0] }
```

Il s'agit d'un type struct contenant un tableau sans éléments. Puisque le champ n'est pas , les valeurs de ce type ne peuvent pas être construites en dehors de ce module, ce qui est parfait comme reflet d'un type C qui ne

devrait jamais construire, et qui est manipulé uniquement par des pointeurs bruts. `_private pub libgit2`

Écrire de gros blocs à la main peut être une corvée. Si vous créez une interface Rust vers une bibliothèque C complexe, vous pouvez essayer d'utiliser la caisse, qui dispose de fonctions que vous pouvez utiliser à partir de votre script de build pour analyser les fichiers d'en-tête C et générer automatiquement les déclarations Rust correspondantes. Nous n'avons pas d'espace pour montrer en action ici, mais [la page de bindgen sur crates.io](#) comprend des liens vers sa documentation. `extern bindgen bindgen`

Ensuite, nous allons *réécrire complètement* `main.rs`. Tout d'abord, nous devons déclarer le module: `raw`

```
mod raw;
```

Selon les conventions de , les fonctions faillibles renvoient un code entier positif ou nul en cas de succès et négatif en cas d'échec. Si une erreur se produit, la fonction renvoie un pointeur vers une structure fournissant plus de détails sur ce qui s'est mal passé. possède cette structure, nous n'avons donc pas besoin de la libérer nous-mêmes, mais elle pourrait être écrasée par le prochain appel de bibliothèque que nous faisons. Une interface Rust appropriée utiliserait , mais dans la version brute, nous voulons utiliser les fonctions telles quelles, nous devrons donc lancer notre propre fonction pour gérer les erreurs:

```
libgit2 giterr_last git_error libgit2 Result libgit2

use std::ffi::CStr;
use std::os::raw::c_int;

fn check(activity: &'static str, status: c_int) -> c_int {
 if status < 0 {
 unsafe {
 let error = &*raw::giterr_last();
 println!("error while {}: {} ({})", activity,
 CStr::from_ptr(error.message).to_string_lossy(),
 error(klass));
 std::process::exit(1);
 }
 }
 status
}
```

Nous utiliserons cette fonction pour vérifier les résultats d'appels comme celui-ci : libgit2

```
check("initializing library", raw::git_libgit2_init());
```

Cela utilise les mêmes méthodes utilisées précédemment : construire le à partir d'une chaîne C et le transformer en quelque chose que Rust peut imprimer. `CStr from_ptr CStr to_string_lossy`

Ensuite, nous avons besoin d'une fonction pour imprimer un commit :

```
unsafe fn show_commit(commit: *const raw::git_commit) {
 let author = raw::git_commit_author(commit);

 let name = CStr::from_ptr((*author).name).to_string_lossy();
 let email = CStr::from_ptr((*author).email).to_string_lossy();
 println!("{} <{}>\n", name, email);

 let message = raw::git_commit_message(commit);
 println!("{}", CStr::from_ptr(message).to_string_lossy());
}
```

Donné un pointeur vers un `git_commit`, appelle et pour récupérer les informations dont il a besoin. Ces deux fonctions suivent une convention que la documentation explique comme suit

```
:git_commit show_commit git_commit_author git_commit_message libgit2
```

*Si une fonction renvoie un objet en tant que valeur de retour, cette fonction est un getter et la durée de vie de l'objet est liée à l'objet parent.*

En termes de rouille, et sont empruntés à : n'a pas besoin de les libérer lui-même, mais il ne doit pas les conserver après sa libération. Étant donné que cette API utilise des pointeurs bruts, Rust ne vérifiera pas leur durée de vie pour nous : si nous créons accidentellement des pointeurs pendents, nous ne le découvrirons probablement pas avant que le programme ne plante. `author message commit show_commit commit`

Le code précédent suppose que ces champs contiennent du texte UTF-8, ce qui n'est pas toujours correct. Git autorise également d'autres encodages. Interpréter correctement ces cordes impliquerait probablement d'utiliser la caisse. Par souci de brièveté, nous passerons sous silence ces questions ici. `encoding`

La fonction de notre programme se lit comme suit : `main`

```

use std::ffi::CString;
use std::mem;
use std::ptr;
use std::os::raw::c_char;

fn main() {
 let path = std::env::args().skip(1).next()
 .expect("usage: git-toy PATH");
 let path = CString::new(path)
 .expect("path contains null characters");

 unsafe {
 check("initializing library", raw::git_libgit2_init());

 let mut repo = ptr::null_mut();
 check("opening repository",
 raw::git_repository_open(&mut repo, path.as_ptr()));

 let c_name = b"HEAD\0".as_ptr() as *const c_char;
 let oid = {
 let mut oid = mem::MaybeUninit::uninit();
 check("looking up HEAD",
 raw::git_reference_name_to_id(oid.as_mut_ptr(), repo,
 oid.assume_init())
);
 };

 let mut commit = ptr::null_mut();
 check("looking up commit",
 raw::git_commit_lookup(&mut commit, repo, &oid));

 show_commit(commit);

 raw::git_commit_free(commit);

 raw::git_repository_free(repo);

 check("shutting down library", raw::git_libgit2_shutdown());
 }
}

```

Cela commence par du code pour gérer l'argument path et initialiser la bibliothèque, ce que nous avons déjà vu. Le premier nouveau code est le suivant :

```

let mut repo = ptr::null_mut();
check("opening repository",
 raw::git_repository_open(&mut repo, path.as_ptr()));

```

L'appel à tente d'ouvrir le référentiel Git au chemin d'accès donné. S'il réussit, il lui alloue un nouvel objet et définit pour pointer vers cela. Rust constraint implicitement les références à des pointeurs bruts, donc passer ici fournit l'appel

```
attendu.git_repository_open git_repository repo &mut
repo *mut *mut git_repository
```

Cela montre une autre convention en cours d'utilisation (à partir de la documentation): libgit2 libgit2

*Les objets qui sont renvoyés via le premier argument en tant que pointeur à pointeur appartiennent à l'appelant et il est responsable de les libérer.*

En termes de rouille, des fonctions telles que transmettre la propriété de la nouvelle valeur à l'appelant. `git_repository_open`

Ensuite, considérez le code qui recherche le hachage d'objet de la validation d'en-tête actuelle du référentiel :

```
let oid = {
 let mut oid = mem::MaybeUninit::uninit();
 check("looking up HEAD",
 raw::git_reference_name_to_id(oid.as_mut_ptr(), repo, c_name));
 oid.assume_init()
};
```

Le type stocke un identificateur d'objet, c'est-à-dire un code de hachage 160 bits que Git utilise en interne (et tout au long de son interface utilisateur agréable) pour identifier les validations, les versions individuelles des fichiers, etc. Cet appel permet de rechercher l'identificateur d'objet de la validation en cours. `git_oid git_reference_name_to_id "HEAD"`

En C, il est parfaitement normal d'initialiser une variable en lui passant un pointeur vers une fonction qui remplit sa valeur ; c'est ainsi qu'on s'attend à traiter son premier argument. Mais Rust ne nous laissera pas emprunter une référence à une variable non initialisée. Nous pourrions initialiser avec des zéros, mais c'est un gaspillage: toute valeur qui y est stockée sera simplement écrasée. `git_reference_name_to_id oid`

Il est possible de demander à Rust de nous donner une mémoire non initialisée, mais parce que la lecture de la mémoire non initialisée à tout moment est un comportement instantané indéfini, Rust fournit une abstraction, , pour faciliter son utilisation. dit au compilateur de mettre de côté suffisamment de mémoire pour votre type, mais de ne pas le toucher

jusqu'à ce que vous disiez qu'il est sûr de le faire. Bien que cette mémoire appartienne au , le compilateur évitera également certaines optimisations qui pourraient autrement provoquer un comportement indéfini, même sans accès explicite à la mémoire non initialisée de votre code.

```
MaybeUninit MaybeUninit<T> T MaybeUninit
```

MaybeUninit fournit une méthode , qui produit un pointage vers la mémoire potentiellement non initialisée qu'il enveloppe. En passant ce pointeur à une fonction étrangère qui initialise la mémoire, puis en appelant la méthode non sécurisée sur le pour produire un comportement entièrement initialisé , vous pouvez éviter un comportement indéfini sans la surcharge supplémentaire qui provient de l'initialisation et de la suppression immédiate d'une valeur. est dangereux car l'appeler sur un sans être certain que la mémoire est réellement initialisée provoquera immédiatement un comportement indéfini.

```
as_mut_ptr() *mut
T assume_init MaybeUninit T assume_init MaybeUninit
```

Dans ce cas, il est sûr car initialise la mémoire appartenant au . Nous pourrions également utiliser pour les variables et, mais comme il ne s'agit que de mots simples, nous allons de l'avant et les initialisons à null:

```
git_reference_name_to_id MaybeUninit MaybeUninit repo c
ommit
```

```
let mut commit = ptr::null_mut();
check("looking up commit",
 raw::git_commit_lookup(&mut commit, repo, &oid));
```

Cela prend l'identificateur d'objet de la validation et recherche la validation réelle, en stockant un pointeur sur la réussite.

```
git_commit commit
```

Le reste de la fonction doit être explicite. Il appelle la fonction définie précédemment, libère les objets commit et repository et arrête la bibliothèque.

```
main show_commit
```

Maintenant, nous pouvons essayer le programme sur n'importe quel dépôt Git prêt à portée de main:

```
$ cargo run /home/jimb/rbattle
Finished dev [unoptimized + debuginfo] target(s) in 0.0 secs
Running `target/debug/git-toy /home/jimb/rbattle`
Jim Blandy <jimb@red-bean.com>
```

Animate goop a bit.

# Une interface sécurisée vers libgit2

L'interface brute est un exemple parfait d'une fonctionnalité dangereuse: elle peut certainement être utilisée correctement (comme nous le faisons ici, pour autant que nous le sachions), mais Rust ne peut pas appliquer les règles que vous devez suivre. La conception d'une API sécurisée pour une bibliothèque comme celle-ci consiste à identifier toutes ces règles, puis à trouver des moyens de transformer toute violation de celles-ci en une erreur de type ou de vérification d'emprunt. libgit2

Voici donc les règles de fonctionnement des fonctionnalités utilisées par le programme : libgit2

- Vous devez appeler avant d'utiliser une autre fonction de bibliothèque. Vous ne devez utiliser aucune fonction de bibliothèque après avoir appelé `.git_libgit2_init git_libgit2_shutdown`
- Toutes les valeurs transmises aux fonctions doivent être entièrement initialisées, à l'exception des paramètres de sortie. libgit2
- Lorsqu'un appel échoue, les paramètres de sortie transmis pour contenir les résultats de l'appel ne sont pas initialisés et vous ne devez pas utiliser leurs valeurs.
- Un objet fait référence à l'objet dont il est dérivé, de sorte que le premier ne doit pas survivre au second. (Ce n'est pas précisé dans la documentation; nous l'avons déduit de la présence de certaines fonctions dans l'interface, puis vérifié en lisant le code source.) `git_commit git_repository libgit2`
- De même, a est toujours emprunté à un donné, et le premier ne doit pas survivre au second. (La documentation couvre ce cas.) `git_signature git_commit`
- Le message associé à une validation ainsi que le nom et l'adresse e-mail de l'auteur sont tous empruntés à la validation et ne doivent pas être utilisés après la libération de la validation.
- Une fois qu'un objet a été libéré, il ne doit plus jamais être utilisé. libgit2

Il s'avère que vous pouvez créer une interface Rust qui applique toutes ces règles, soit via le système de type Rust, soit en gérant les détails en interne. libgit2

Avant de commencer, restructurons un peu le projet. Nous aimerais avoir un module qui exporte l'interface sécurisée, dont l'interface brute du programme précédent est un sous-module privé. git

L'arborescence source entière ressemblera à ceci :

```
git-toy/
└── Cargo.toml
└── build.rs
└── src/
 ├── main.rs
 └── git/
 ├── mod.rs
 └── raw.rs
```

En suivant les règles que nous avons expliquées dans [« Modules dans des fichiers séparés »](#), la source du module apparaît dans `git/mod.rs`, et la source de son sous-module va dans `git/raw.rs`. `git git::raw`

Encore une fois, nous allons *réécrire complètement* `main.rs`. Il doit commencer par une déclaration du module: `git`

```
mod git;
```

Ensuite, nous devrons créer le sous-répertoire `git` et y déplacer `raw.rs`:

```
$ cd /home/jimb/git-toy
$ mkdir src/git
$ mv src/raw.rs src/git/raw.rs
```

Le module doit déclarer son sous-module. Le fichier `src/git/mod.rs` doit indiquer: `git raw`

```
mod raw;
```

Comme ce n'est pas le cas, ce sous-module n'est pas visible par le programme principal. `pub`

Dans un peu, nous devrons utiliser certaines fonctions de la caisse, nous devons donc ajouter une dépendance dans `Cargo.toml`. Le fichier complet se lit maintenant comme suit : `libc`

```
[package]
name = "git-toy"
version = "0.1.0"
authors = ["You <you@example.com>"]
edition = "2021"

[dependencies]
libc = "0.2"
```

Maintenant que nous avons restructuré nos modules, considérons la gestion des erreurs. Même la fonction d'initialisation de Même peut renvoyer un code d'erreur, nous devrons donc le régler avant de pouvoir commencer. Une interface Rust idiomatique a besoin de son propre type qui capture le code d'échec ainsi que le message d'erreur et la classe de . Un type d'erreur approprié doit implémenter les traits habituels , et . Ensuite, il a besoin de son propre type qui utilise ce type. Voici les définitions nécessaires dans *src/git/mod.rs*

```
:libgit2 Error libgit2 giterr_last Error Debug Display Result Error

use std::error;
use std::fmt;
use std::result;

#[derive(Debug)]
pub struct Error {
 code: i32,
 message: String,
 class: i32
}

impl fmt::Display for Error {
 fn fmt(&self, f: &mut fmt::Formatter) -> result::Result<(), fmt::Error> {
 // Displaying an `Error` simply displays the message from libgit2
 self.message(fmt(f))
 }
}

impl error::Error for Error { }

pub type Result<T> = result::Result<T, Error>;
```

Pour vérifier le résultat des appels de bibliothèque bruts, le module a besoin d'une fonction qui transforme un code de retour en un

```
:libgit2 Result

use std::os::raw::c_int;
use std::ffi::CStr;

fn check(code: c_int) -> Result<c_int> {
 if code >= 0 {
 return Ok(code);
 }

 unsafe {
```

```

let error = raw::giterr_last();

// libgit2 ensures that (*error).message is always non-null and r
// terminated, so this call is safe.
let message = CStr::from_ptr((*error).message)
 .to_string_lossy()
 .into_owned();

Err(Error {
 code: code as i32,
 message,
 class: (*error).klass as i32
})
}
}
}

```

La principale différence entre cela et la fonction de la version brute est que cela construit une valeur au lieu d'imprimer un message d'erreur et de quitter immédiatement. `check Error`

Nous sommes maintenant prêts à nous attaquer à l'initialisation. L'interface sécurisée fournira un type qui représente un référentiel Git ouvert, avec des méthodes pour résoudre les références, rechercher des validations, etc. En continuant dans `git/mod.rs`, voici la définition de

`:libgit2 Repository Repository`

```

/// A Git repository.
pub struct Repository {
 // This must always be a pointer to a live `git_repository` structure
 // No other `Repository` may point to it.
 raw: *mut raw::git_repository
}

```

Le champ de `A` n'est pas public. Étant donné que seul le code de ce module peut accéder au pointeur, l'obtention correcte de ce module devrait garantir que le pointeur est toujours utilisé correctement. `Repository raw raw::git_repository`

Si la seule façon de créer un est d'ouvrir avec succès un nouveau référentiel Git, cela garantira que chacun pointe vers un objet distinct

`:Repository Repository git_repository`

```

use std::path::Path;
use std::ptr;

impl Repository {

```

```

pub fn open<P: AsRef<Path>>(path: P) -> Result<Repository> {
 ensure_initialized();

 let path = path_to_cstring(path.as_ref())?;
 let mut repo = ptr::null_mut();
 unsafe {
 check(raw::git_repository_open(&mut repo, path.as_ptr()))?;
 }
 Ok(Repository { raw: repo })
}

```

Étant donné que la seule façon de faire quoi que ce soit avec l'interface sécurisée est de commencer par une valeur, et commence par un appel à , nous pouvons être sûrs que cela sera appelé avant toute fonction. Sa définition est la suivante

```
:Repository Repository::open ensure_initialized ensure_init
ialized libgit2
```

```

fn ensure_initialized() {
 static ONCE: std::sync::Once = std::sync::Once::new();
 ONCE.call_once(|| {
 unsafe {
 check(raw::git_libgit2_init())
 .expect("initializing libgit2 failed");
 assert_eq!(libc::atexit(shutdown), 0);
 }
 });
}

extern fn shutdown() {
 unsafe {
 if let Err(e) = check(raw::git_libgit2_shutdown()) {
 eprintln!("shutting down libgit2 failed: {}", e);
 std::process::abort();
 }
 }
}

```

Le type permet d'exécuter le code d'initialisation de manière sécurisée. Seul le premier thread à appeler exécute la fermeture donnée. Tous les appels suivants, par ce thread ou tout autre, se bloquent jusqu'à ce que le premier soit terminé, puis reviennent immédiatement, sans exécuter à nouveau la fermeture. Une fois la fermeture terminée, l'appel est bon marché, ne nécessitant rien de plus qu'une charge atomique d'un dra-

peau stocké dans

```
.std::sync::Once ONCE.call_once ONCE.call_once ONCE
```

Dans le code précédent, la fermeture de l'initialisation appelle et vérifie le résultat. Il s'agit un peu et utilise simplement pour s'assurer que l'initialisation a réussi, au lieu d'essayer de propager les erreurs à l'appelant.

```
git_libgit2_init expect
```

Pour s'assurer que le programme appelle , la fermeture d'initialisation utilise la fonction de la bibliothèque C, qui prend un pointeur vers une fonction à appeler avant la fermeture du processus. Les fermetures de rouille ne peuvent pas servir de pointeurs de fonction C : une fermeture est une valeur d'un type anonyme portant les valeurs de toutes les variables qu'elle capture ou auxquelles elle fait référence ; un pointeur de fonction C n'est qu'un pointeur. Cependant, les types Rust fonctionnent bien, tant que vous les déclarez afin que Rust sache utiliser les conventions d'appel C. La fonction locale correspond à la facture et garantit une fermeture

```
correcte. git_libgit2_shutdown atexit fn extern shutdown libgit2
```

Dans « [Unwinding](#) », nous avons mentionné qu'il est un comportement indéfini pour une panique de traverser les frontières linguistiques. L'appel de à est une telle limite, il est donc essentiel de ne pas paniquer. C'est pourquoi vous ne pouvez pas simplement utiliser pour gérer les erreurs signalées par . Au lieu de cela, il doit signaler l'erreur et mettre fin au processus lui-même. POSIX interdit d'appeler dans un gestionnaire, donc appelle pour arrêter le programme

```
brusquement. atexit shutdown shutdown shutdown .expect raw:: git_libgit2_shutdown exit atexit shutdown std::process::abort
```

Il peut être possible de prendre des dispositions pour appeler plus tôt, par exemple lorsque la dernière valeur est supprimée. Mais quelle que soit la façon dont nous organisons les choses, l'appel doit être la responsabilité de l'API sûre. Au moment où il est appelé, tous les objets existants deviennent dangereux à utiliser, de sorte qu'une API sécurisée ne doit pas exposer cette fonction

```
directement. git_libgit2_shutdown Repository git_libgit2_shutdown libgit2
```

Le pointeur brut d'un doit toujours pointer vers un objet actif. Cela implique que la seule façon de fermer un référentiel est de supprimer la valeur qui le possède :

```
Repository git_repository Repository
```

```

impl Drop for Repository {
 fn drop(&mut self) {
 unsafe {
 raw::git_repository_free(self.raw);
 }
 }
}

```

En appelant uniquement lorsque le seul pointeur vers le est sur le point de disparaître, le type garantit également que le pointeur ne sera jamais utilisé après sa

libération. `git_repository_free` `raw::git_repository` `Repository`

La méthode utilise une fonction privée appelée , qui a deux définitions, l'une pour les systèmes de type Unix et l'autre pour Windows

`:Repository::open path_to_cstring`

```

use std::ffi::CString;

#[cfg(unix)]
fn path_to_cstring(path: &Path) -> Result<CString> {
 // The `as_bytes` method exists only on Unix-like systems.
 use std::os::unix::ffi::OsStrExt;

 Ok(CString::new(path.as_os_str().as_bytes())?)
}

#[cfg(windows)]
fn path_to_cstring(path: &Path) -> Result<CString> {
 // Try to convert to UTF-8. If this fails, libgit2 can't handle the path.
 // anyway.
 match path.to_str() {
 Some(s) => Ok(CString::new(s)?),
 None => {
 let message = format!("Couldn't convert path '{}' to UTF-8",
 path.display());
 Err(message.into())
 }
 }
}

```

L'interface rend ce code un peu délicat. Sur toutes les plates-formes, accepte les chemins d'accès en tant que chaînes C terminées par une valeur NULL. Sous Windows, suppose que ces chaînes C contiennent un UTF-8 bien formé et les convertit en interne en chemins d'accès 16 bits requis par Windows. Cela fonctionne généralement, mais ce n'est pas idéal. Win-

dows autorise les noms de fichiers qui ne sont pas bien formés Unicode et ne peuvent donc pas être représentés en UTF-8. Si vous avez un tel fichier, il est impossible de passer son nom à

```
.libgit2 libgit2 libgit2 libgit2
```

Dans Rust, la représentation correcte d'un chemin de système de fichiers est un , soigneusement conçu pour gérer tout chemin pouvant apparaître sur Windows ou POSIX. Cela signifie qu'il existe des valeurs sur Windows auxquelles on ne peut pas passer , car elles ne sont pas bien formées UTF-8. Donc, bien que le comportement de 'soit loin d'être idéal, c'est en fait le meilleur que nous puissions faire étant donné l'interface de

```
.. std::path::Path Path libgit2 path_to_cstring libgit2
```

Les deux définitions qui viennent d'être présentées reposent sur des conversions de notre type : l'opérateur tente de telles conversions, et la version Windows appelle explicitement . Ces conversions ne sont pas remarquables :  
`path_to_cstring Error ? .into()`

```
impl From<String> for Error {
 fn from(message: String) -> Error {
 Error { code: -1, message, class: 0 }
 }
}

// NulError is what `CString::new` returns if a string
// has embedded zero bytes.
impl From<std::ffi::NulError> for Error {
 fn from(e: std::ffi::NulError) -> Error {
 Error { code: -1, message: e.to_string(), class: 0 }
 }
}
```

Ensuite, voyons comment résoudre une référence Git à un identificateur d'objet. Étant donné qu'un identificateur d'objet n'est qu'une valeur de hachage de 20 octets, il est parfaitement correct de l'exposer dans l'API sécurisée :

```
/// The identifier of some sort of object stored in the Git object
/// database: a commit, tree, blob, tag, etc. This is a wide hash of the
/// object's contents.
pub struct Oid {
 pub raw: raw::git_oid
}
```

Nous allons ajouter une méthode pour effectuer la recherche

: Repository

```
use std::mem;
use std::os::raw::c_char;

impl Repository {
 pub fn reference_name_to_id(&self, name: &str) -> Result<Oid> {
 let name = CString::new(name)?;
 unsafe {
 let oid = {
 let mut oid = mem::MaybeUninit::uninit();
 check(raw::git_reference_name_to_id(
 oid.as_mut_ptr(), self.raw,
 name.as_ptr() as *const c_char))?;
 oid.assume_init()
 };
 Ok(Oid { raw: oid })
 }
 }
}
```

Bien qu'elle ne soit pas initialisée lorsque la recherche échoue, cette fonction garantit que son appelant ne peut jamais voir la valeur non initialisée simplement en suivant l'idiome de Rust : soit l'appelant obtient une valeur correctement initialisée, soit il obtient un

.oid Result Ok Oid Err

Ensuite, le module a besoin d'un moyen de récupérer les commits du référentiel. Nous allons définir un type comme suit : Commit

```
use std::marker::PhantomData;

pub struct Commit<'repo> {
 // This must always be a pointer to a usable `git_commit` structure.
 raw: *mut raw::git_commit,
 _marker: PhantomData<&'repo Repository>
}
```

Comme nous l'avons mentionné précédemment, un objet ne doit jamais survivre à l'objet dont il a été récupéré. Les durées de vie de Rust permettent au code de capturer cette règle avec précision. `git_commit` `git_repository`

L'exemple plus haut dans ce chapitre utilisait un champ pour indiquer à Rust de traiter un type comme s'il contenait une référence avec une

durée de vie donnée, même si le type ne contenait apparemment aucune référence de ce type. Le type doit faire quelque chose de similaire. Dans ce cas, le type du champ est , indiquant que Rust doit traiter comme s'il détenait une référence à vie à certains

```
.RefWithFlag PhantomData Commit _marker PhantomData<&'repo
Repository> Commit<'repo> 'repo Repository
```

La méthode de recherche d'une validation est la suivante :

```
impl Repository {
 pub fn find_commit(&self, oid: &Oid) -> Result<Commit> {
 let mut commit = ptr::null_mut();
 unsafe {
 check(raw::git_commit_lookup(&mut commit, self.raw, &oid.raw));
 }
 Ok(Commit { raw: commit, _marker: PhantomData })
 }
}
```

Comment cela relie-t-il la durée de vie de la '? La signature de omet les durées de vie des références impliquées selon les règles décrites dans [« Omis des paramètres de durée de vie »](#). Si nous devions écrire les durées de vie, la signature complète se lirait comme suit: Commit Repository find\_commit

```
fn find_commit<'repo, 'id>(&'repo self, oid: &'id Oid)
-> Result<Commit<'repo>>
```

C'est exactement ce que nous voulons: Rust traite le retourné comme s'il empruntait quelque chose à , qui est le .Commit self Repository

Lorsque a est lâché, il doit libérer son : Commit raw::git\_commit

```
impl<'repo> Drop for Commit<'repo> {
 fn drop(&mut self) {
 unsafe {
 raw::git_commit_free(self.raw);
 }
 }
}
```

À partir d'un , vous pouvez emprunter un (un nom et une adresse e-mail) et le texte du message de validation : Commit Signature

```
impl<'repo> Commit<'repo> {
 pub fn author(&self) -> Signature {
```

```

unsafe {
 Signature {
 raw: raw::git_commit_author(self.raw),
 _marker: PhantomData
 }
}

pub fn message(&self) -> Option<&str> {
 unsafe {
 let message = raw::git_commit_message(self.raw);
 char_ptr_to_str(self, message)
 }
}
}

```

Voici le type : `Signature`

```

pub struct Signature<'text> {
 raw: *const raw::git_signature,
 _marker: PhantomData<&'text str>
}

```

Un objet emprunte toujours son texte à d'autres endroits; en particulier, les signatures retournées en empruntant leur texte au . Donc, notre type de sécurité comprend un pour dire à Rust de se comporter comme s'il contenait un avec une durée de vie de . Tout comme auparavant, relie correctement cette vie du il revient à celle du sans que nous ayons besoin d'écrire une chose. La méthode fait de même avec la conservation du message de

```

validation.git_signature git_commit_author git_commit Signature PhantomData<&'text str> &str 'text Commit::author 'text Signature Commit Commit::message Option<&str>

```

A inclut des méthodes pour récupérer le nom et l'adresse e-mail de l'auteur : `Signature`

```

impl<'text> Signature<'text> {
 /// Return the author's name as a `&str`,
 /// or `None` if it is not well-formed UTF-8.
 pub fn name(&self) -> Option<&str> {
 unsafe {
 char_ptr_to_str(self, (*self.raw).name)
 }
 }
}

```

```

 /// Return the author's email as a `&str`,
 /// or `None` if it is not well-formed UTF-8.
 pub fn email(&self) -> Option<&str> {
 unsafe {
 char_ptr_to_str(self, (*self.raw).email)
 }
 }
}

```

Les méthodes précédentes dépendent d'une fonction utilitaire privée

:char\_ptr\_to\_str

```

/// Try to borrow a `&str` from `ptr`, given that `ptr` may be null or
/// refer to ill-formed UTF-8. Give the result a lifetime as if it were
/// borrowed from `_owner`.
///
/// Safety: if `ptr` is non-null, it must point to a null-terminated C
/// string that is safe to access for at least as long as the lifetime of
/// `_owner`.
unsafe fn char_ptr_to_str<'T>(_owner: &T, ptr: *const c_char) -> Option<&str>
{
 if ptr.is_null() {
 return None;
 } else {
 CStr::from_ptr(ptr).to_str().ok()
 }
}

```

La valeur du paramètre n'est jamais utilisée, mais sa durée de vie l'est.

Rendre explicites les durées de vie dans la signature de cette fonction nous donne : \_owner

```

fn char_ptr_to_str<'o, T: 'o>(_owner: &'o T, ptr: *const c_char)
-> Option<&'o str>

```

La fonction renvoie un dont la durée de vie est complètement illimitée, puisqu'elle a été empruntée à un pointeur brut déréférencé. Les durées de vie illimitées sont presque toujours inexactes, il est donc bon de les contraindre dès que possible. L'inclusion du paramètre entraîne l'attribution par Rust de sa durée de vie au type de la valeur renvoyée, afin que les appelants puissent recevoir une référence délimitée plus précise. `CStr::from_ptr &CStr _owner`

Il n'est pas clair dans la documentation si un 's et des pointeurs peuvent être nuls, bien que la documentation soit assez bonne. Vos auteurs ont fouillé dans le code source pendant un certain temps sans pouvoir se per-

suader d'une manière ou d'une autre et ont finalement décidé qu'il valait mieux se préparer à des pointeurs nuls au cas où. Dans Rust, ce genre de question est répondu immédiatement par le type: si c'est , vous pouvez compter sur la chaîne pour être là; si c'est , c'est

```
 facultatif.libgit2 git_signature email author libgit2 char_ptr
 _to_str &str Option<&str>
```

Enfin, nous avons fourni des interfaces sécurisées pour toutes les fonctionnalités dont nous avons besoin. La nouvelle fonction dans *src / main.rs* est un peu allégée et ressemble à un vrai code Rust: *main*

```
fn main() {
 let path = std::env::args_os().skip(1).next()
 .expect("usage: git-toy PATH");

 let repo = git::Repository::open(&path)
 .expect("opening repository");

 let commit_oid = repo.reference_name_to_id("HEAD")
 .expect("looking up 'HEAD' reference");

 let commit = repo.find_commit(&commit_oid)
 .expect("looking up commit");

 let author = commit.author();
 println!("{} <{}>\n",
 author.name().unwrap_or("(none)"),
 author.email().unwrap_or("none"));

 println!("{}", commit.message().unwrap_or("(none)"));
}
```

Dans ce chapitre, nous sommes passés d'interfaces simplistes qui ne fournissent pas beaucoup de garanties de sécurité à une API sécurisée enveloppant une API intrinsèquement dangereuse en faisant en sorte que toute violation du contrat de cette dernière soit une erreur de type Rust. Le résultat est une interface que Rust peut s'assurer que vous utilisez correctement. Pour la plupart, les règles que nous avons fait appliquer par Rust sont le genre de règles que les programmeurs C et C ++ finissent par s'imposer de toute façon. Ce qui rend Rust beaucoup plus strict que C et C ++, ce n'est pas que les règles sont si étrangères, mais que cette application est mécanique et complète.

## Conclusion

Rust n'est pas un langage simple. Son but est de couvrir deux mondes très différents. C'est un langage de programmation moderne, sûr par conception, avec des commodités comme des fermetures et des itérateurs, mais il vise à vous donner le contrôle des capacités brutes de la machine sur laquelle il fonctionne, avec une surcharge d'exécution minimale.

Les contours de la langue sont déterminés par ces objectifs. Rust parvient à combler la majeure partie de l'écart avec un code sûr. Son vérificateur d'emprunt et ses abstractions à coût nul vous rapprochent le plus possible du métal nu sans risquer un comportement indéfini. Lorsque cela ne suffit pas ou lorsque vous souhaitez tirer parti du code C existant, le code dangereux et l'interface de la fonction étrangère sont prêts. Mais encore une fois, la langue ne vous offre pas seulement ces fonctionnalités dangereuses et vous souhaite bonne chance. L'objectif est toujours d'utiliser des fonctionnalités dangereuses pour créer des API sécurisées. C'est ce que nous avons fait avec `.` . C'est aussi ce que l'équipe Rust a fait avec `, ,`, les autres collections, canaux, et plus encore : la bibliothèque standard est pleine d'abstractions sécurisées, implémentées avec du code dangereux dans les coulisses. `libgit2 Box Vec`

Un langage avec les ambitions de Rust n'était peut-être pas destiné à être le plus simple des outils. Mais Rust est sûr, rapide, simultané et efficace. Utilisez-le pour construire des systèmes volumineux, rapides, sécurisés et robustes qui tirent parti de toute la puissance du matériel sur lequel ils s'exécutent. Utilisez-le pour améliorer le logiciel.

[Soutien](#) [Se déconnecter](#)

©2022 O'REILLY MEDIA, INC. [CONDITIONS D'UTILISATION](#) [POLITIQUE DE CONFIDENTIALITÉ](#)