Chapitre 8. Caisses et modules

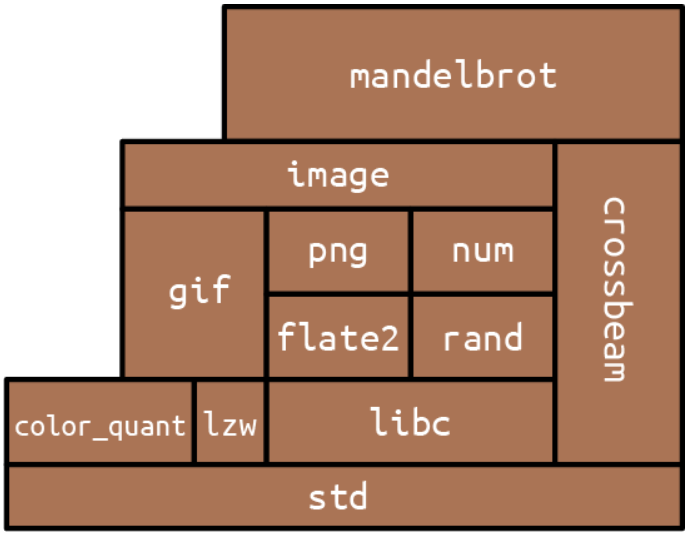
> *C'est une note dans un thème Rust: les programmeurs de systèmes peuvent avoir de belles choses.*
>
> —Robert O'Callahan, « Pensées aléatoires sur rust: crates.io et les IDE »

Supposons que vous écriviez 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-l
        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-traits = "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 Cargo.toml utilisée
`cargo build`	`[profile.dev]`
`cargo build --release`	`[profile.release]`
`cargo test`	`[profile.test]`

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'entourent*, 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. 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) 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 `AminoAci
    ...
```

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 et ont une signification particulière dans les chemins: se réfère au module parent, et se réfère 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 r

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 , 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 plant_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 thread-safe, 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/efer-n.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 et . `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]`

`#[cfg(...)]` option	Activé lorsque
`test`	Les tests sont activés (compilation avec ou). `cargo test rustc --test`
`debug_assertions`	Les assertions de débogage sont activées (généralement dans les builds non optimisées).
`unix`	Compilation pour Unix, y compris macOS.
`windows`	Compilation pour Windows.
`target_pointer_width = "64"`	Cibler une plateforme 64 bits. L'autre valeur possible est `."32"`
`target_arch = "x86_64"`	Cibler x86-64 en particulier. Autres valeurs: `,,, « powerpc »,,."x86" "arm" "aarch64" "powerpc64" "mips"`
`target_os = "macos"`	Compilation pour macOS. Autres valeurs : `,,,,,,,."windows" "ios" "android" "linux" "freebsd" "openbsd" "netbsd" "dragonfly"`

`#[cfg(...)]` **option**	**Activé lorsque**
`feature = "robots"`	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(`*Un*`)`	*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(`*AB*`, )`	*A* et *B sont tous deux* satisfaits (l'équivalent de). `&&`
`any(`*AB*`, )`	*A* ou *B* est satisfait (l'équivalent de). `||`

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 insistants, 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 et , peuvent être attachés à un module entier et s'appliquer à tout ce qu'il contient. D'autres, comme et , 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. `#[cfg]` `#[allow]` `#[test]` `#[inline]`

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 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 , 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] 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 ou
```

(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*.

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

cells	The simulation of biological cells, which is as low-level as we go.
plant_structures	Higher-level biological structures.
simulation	Overall simulation control.
spores	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 et , 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::Le
///
/// [r]: roots::Root
pub fn trace_path(leaf: &leaves::Leaf, root: &roots::Root) -> VascularPat
    ...
}
```

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

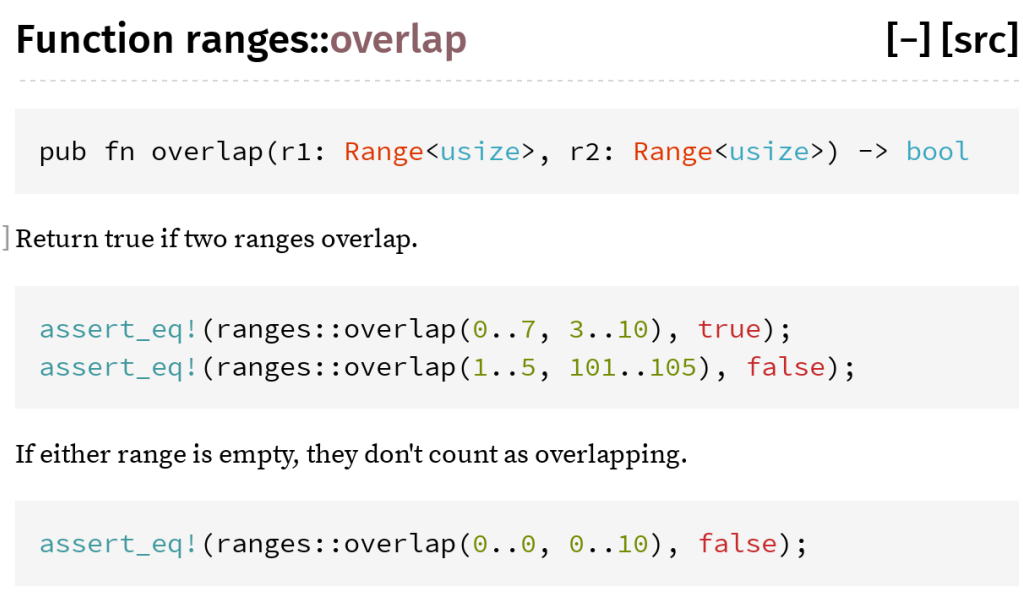
Lorsque vous exécutez des tests dans une caisse de bibliothèque Rust, Rust vérifie que tout le code qui apparaît dans votre documentation s'exécute et fonctionne réellement. Pour ce faire, il prend chaque bloc de code qui apparaît dans un commentaire de document, le compile en tant que caisse exécutable 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 , comme illustré à la figure 8-3. `cargo doc`



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 ou
```

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. 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 à la place de . Les blocs marqués avec n'apparaissent pas dans la sortie de , 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 ou , ou pour le texte brut. 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 forker: 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 sémantique.

- 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
`image = "=0.10.0"`	Utilisez uniquement la version exacte 0.10.0
`image = ">=1.0.5"`	Utilisez 1.0.5 ou *toute version* supérieure (même 2.9, si elle est disponible)
`image = ">1.0.5 <1.1.9"`	Utilisez une version supérieure à 1.0.5, mais inférieure à 1.1.9
`image = "<=2.7.10"`	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, documentatio
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://fernsim.example.com/"
repository = "https://gitlair.com/sporeador/fern_sim"
documentation = "http://fernsim.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 5j0dV54BjlXBpUUbfIj7G9DvNl1vsWW1
```

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 côte à côte 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. `target`

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-commentaire de niveau supérieur de votre caisse. Cette fonctionnalité est proposée 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 ambitieux. Il dispose d'excellents outils et d'une communauté active. Les programmeurs système *peuvent* avoir de belles choses.