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 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`. Sir
    // `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. `GrayscaleMap`

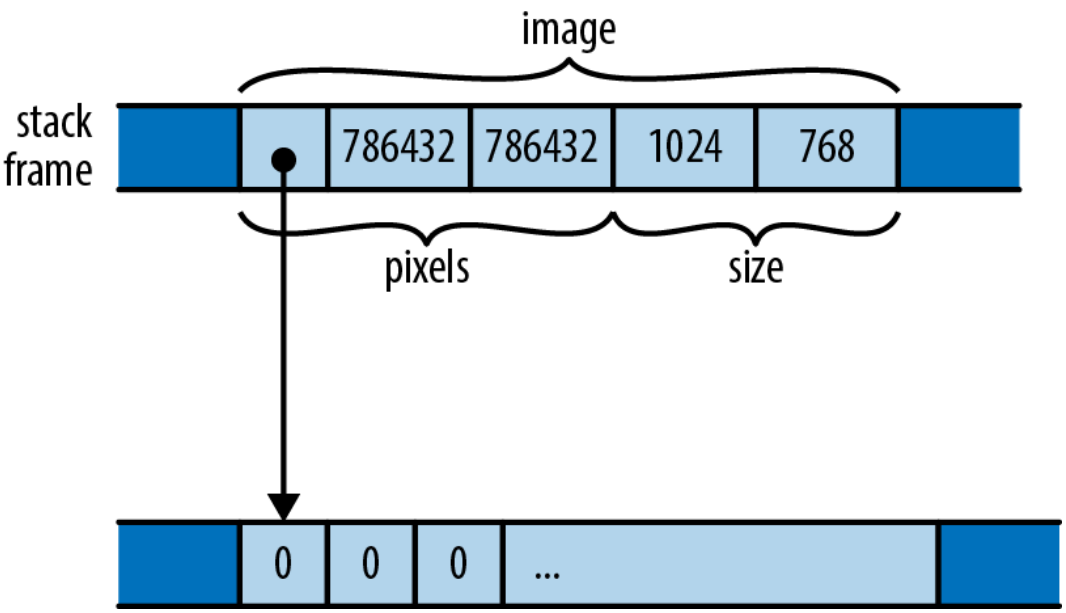

Figure 9-1. Une structure en mémoire `GrayscaleMap`

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 pixel s`

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 t
    /// 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 the
            // 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 . Puisque le type de est évidemment celui nommé en haut du bloc, ou une référence à cela, Rust vous permet d'omettre le type, et d'écrire , , ou comme raccourci pour , , ou . 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 et se réfèrent aux champs de 'comme et . 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 , et la façon dont les méthodes JavaScript utilisent
. `push pop Queue self.older self.younger self self this`

Puisque et ont besoin de modifier le , ils prennent tous les deux . 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 , comme si vous aviez écrit , 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 , 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('⊙');
assert!(!q.is_empty());
```

Ou, si une méthode veut s'approprier , 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 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 . Ce serait une perte de temps. `Queue String char 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.older.is_empty() && self.younger.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 { ... } Queue<T>`

`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 peut 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 usize`

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 Polynomial`

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 itére 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

doivent 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 souhaitiez 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 value`

`cell.get()`

Renvoie une copie de la valeur dans le fichier . `cell`

`cell.set(value)`

Stocke la donnée dans le , en supprimant la valeur précédemment stockée. `value cell`

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 : `Cell SpiderRobot`

```
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, reportez-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`