Chapitre 9. Structures

> *Autrefois, lorsque les bergers voulaient voir si deux troupeaux de moutons étaient isomorphes, ils recherchaient un isomorphisme explicite.*
>
> —John C. Baez et James Dolan, « Catégorisation »

RouillerLes structures, parfois appelées *structures* , ressemblent aux `struct` types en C et C++, aux classes en Python et aux objets en JavaScript. Une structure assemble plusieurs valeurs de types assortis en une seule valeur afin que vous puissiez les traiter comme une unité. Étant donné une structure, vous pouvez lire et modifier ses composants individuels. Et une structure peut être associée à des méthodes qui fonctionnent sur ses composants.

Rust a trois types de types de structure, *named-field* , *tuple-like* et *unit-like* , qui diffèrent dans la façon dont vous vous référez à leurs composants : une structure de champ nommé donne un nom à chaque composant, tandis qu'une structure de type tuple identifie par l'ordre dans lequel ils apparaissent. Les structures de type unité n'ont aucun composant ; ce 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 leur ajouter des méthodes, comment définir des types de structures 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 structures.

Structures de champ nommé

La définition d'une structure de champ nomméle type ressemble à ceci :

```
/// A rectangle of eight-bit grayscale pixels.
struct GrayscaleMap {
    pixels: Vec<u8>,
    size:(usize, usize)
}
```

Ceci déclare un type `GrayscaleMap` avec deux champs nommés `pixels` et `size`, des types donnés. La convention dans Rust est que tous les types, structures incluses, aient des noms qui mettent en majuscule la première lettre de chaque mot, comme `GrayscaleMap`, une convention appelée *CamelCase* (ou *PascalCase*). Les champs et les méthodes sont en minuscules, les mots étant séparés par des traits de soulignement. Ceci s'appelle *snake_case*.

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 structuréecommence par le nom du type (`GrayscaleMap`) et répertorie le nom et la valeur de chaque champ, le tout entre accolades. Il existe également un raccourci pour remplir les champs à partir de variables locales ou d'arguments portant le même nom :

```
fn new_map(size: (usize, usize), pixels: Vec<u8>) ->GrayscaleMap {
    assert_eq!(pixels.len(), size.0 * size.1);
    GrayscaleMap { pixels, size }
}
```

L'expression de structure `GrayscaleMap { pixels, size }` est l'abréviation de `GrayscaleMap { pixels: pixels, size: size }`. Vous pouvez utiliser `key: value` la syntaxe pour certains champs et la sténographie pour d'autres dans la même expression de structure.

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 structures 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 mo-

dule en préfixant sa définition avec `pub`. Il en va de même pour chacun de ses champs, qui sont également privés par défaut :

```
/// A rectangle of eight-bit grayscale pixels.
pub struct GrayscaleMap {
    pub pixels: Vec<u8>,
    pub size:(usize, usize)
}
```

Même si une structure est déclarée `pub`, ses champs peuvent être privés :

```
/// A rectangle of eight-bit grayscale pixels.
pub struct GrayscaleMap {
    pixels: Vec<u8>,
    size:(usize, usize)
}
```

D'autres modules peuvent utiliser cette structure et toutes les fonctions publiques associées qu'elle pourrait avoir, mais ne peuvent pas accéder aux champs privés par leur nom ou utiliser des expressions de structure pour créer de nouvelles `GrayscaleMap` valeurs. Autrement dit, la création d'une valeur de structure nécessite que tous les champs de la structure soient visibles. C'est pourquoi vous ne pouvez pas écrire une expression de structure pour créer un nouveau `String` ou `Vec`. Ces types standard sont des structures, mais tous leurs champs sont privés. Pour en créer un, vous devez utiliser des fonctions publiques associées à un type telles que `Vec::new()`.

Lors de la création d'une valeur de structure de champ nommé, vous pouvez utiliser une autre structure du même type pour fournir des valeurs pour les champs que vous omettez. Dans une expression struct, si les champs nommés sont suivis de `.. EXPR`, tous les champs non mentionnés tirent leurs valeurs de `EXPR`, qui doit être une autre valeur du même type struct. Supposons que nous ayons une structure représentant un monstre dans un jeu :

```
// 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 programmeurs est *L'apprenti sorcier* : un magicien novice enchante un balai pour qu'il fasse son travail à sa place, mais ne sait pas comment l'arrêter une fois le travail terminé. Couper le balai en deux avec une hache ne produit que deux balais, chacun de la moitié de la taille, mais poursuivant la tâche avec le même dévouement aveugle que l'original :

```
// Receive the input Broom by value, taking ownership.
fn chop(b: Broom) -> (Broom, Broom) {
    // Initialize `broom1` mostly from `b`, changing only `height`. Since
    // `String` is not `Copy`, `broom1` takes ownership of `b`'s name.
    let mut broom1 = Broom { height:b.height / 2, .. b };

    // Initialize `broom2` mostly from `broom1`. Since `String` is not
    // `Copy`, we must clone `name` explicitly.
    let mut broom2 = Broom { name:broom1.name.clone(), .. broom1 };

    // Give each fragment a distinct name.
    broom1.name.push_str(" I");
    broom2.name.push_str(" II");

    (broom1, broom2)
}
```

Avec cette définition en place, nous pouvons créer un balai, le couper en deux et voir ce que nous obtenons :

```
let hokey = Broom {
    name: "Hokey".to_string(),
    height: 60,
    health: 100,
    position: (100.0, 200.0, 0.0),
    intent: BroomIntent::FetchWater
};

let (hokey1, hokey2) = chop(hokey);
assert_eq!(hokey1.name, "Hokey I");
assert_eq!(hokey1.height, 30);
assert_eq!(hokey1.health, 100);

assert_eq!(hokey2.name, "Hokey II");
```

```
assert_eq!(hokey2.height, 30);
assert_eq!(hokey2.health, 100);
```

Les nouveaux `hokey1` et `hokey2` les balais ont reçu des noms ajustés, la moitié de la hauteur et toute la santé de l'original.

Structures de type tuple

Le deuxième type de structuretype est appelé un *tuple-like struct* , car il ressemble à un tuple :

```
struct Bounds(usize, usize);
```

Vous construisez une valeur de ce type comme vous le feriez pour un tuple, sauf que vous devez inclure le nom de la structure :

```
let image_bounds = Bounds(1024, 768);
```

Les valeurs détenues par une structure de type tuple sont appelées *éléments*, tout comme le sont 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 `Bounds(1024, 768)` ressemble à un appel de fonction, et en fait c'est le cas : la définition du type définit également implicitement une fonction :

```
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 celui à utiliser se résume à des questions de lisibilité, d'ambiguïté et de brièveté. Si vous utilisez l' `.` opérateur pour accéder aux composants d'une valeur, l'identification des champs par leur nom fournit au lecteur plus d'informations et est proba-

blement plus robuste contre les fautes de frappe. Si vous utilisez généralement la correspondance de modèles pour trouver les éléments, les structures de type tuple peuvent bien fonctionner.

Les structures de type tuple sont bonnes pour les *nouveaux types*, structures avec un seul composant que vous définissez pour obtenir une vérification de type plus stricte. Par exemple, si vous travaillez uniquement avec du texte ASCII, vous pouvez définir un nouveau type comme ceci :

```
struct Ascii(Vec<u8>);
```

L'utilisation de ce type pour vos chaînes ASCII est bien meilleure que de simplement passer des `Vec<u8>` tampons et d'expliquer ce qu'ils sont dans les commentaires. Le newtype aide Rust à détecter les erreurs lorsqu'un autre tampon d'octets est passé à une fonction attendant du texte ASCII. Nous donnerons un exemple d'utilisation de newtypes pour des conversions de type efficaces au chapitre 22 .

Structures de type unité

Le troisième type de structureest un peu obscur : il déclare un type struct sans aucun élément :

```
struct Onesuch;
```

Une valeur d'un tel type n'occupe pas de mémoire, tout comme le type d'unité `()` . Rust ne prend pas la peine de stocker en mémoire des valeurs de structure de type unité ou de générer du code pour les exploiter, car il peut dire tout ce dont il a 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 "Fields and Elements" . Alors qu'une expression like `3..5` est un raccourci pour la struct value `Range { start: 3, end: 5 }` , l'expression `..` , une plage omettant les deux points de terminaison, est un raccourci pour la struct value `RangeFull` .

Les structures de type unité peuvent également être utiles lorsque vous travaillez avec des traits, que nous décrirons au chapitre 11 .

Disposition de la structure

En mémoire, les structures de champ nommé et de type tuplesont la même chose : un ensemble de valeurs, de types éventuellement mixtes, agencées d'une manière particulière en mémoire. Par exemple, plus tôt dans le chapitre, nous avons défini cette structure :

```
struct GrayscaleMap {
    pixels: Vec<u8>,
    size:(usize, usize)
}
```

Une `GrayscaleMap` valeur est disposée en mémoire comme illustré à la Figure 9-1 .

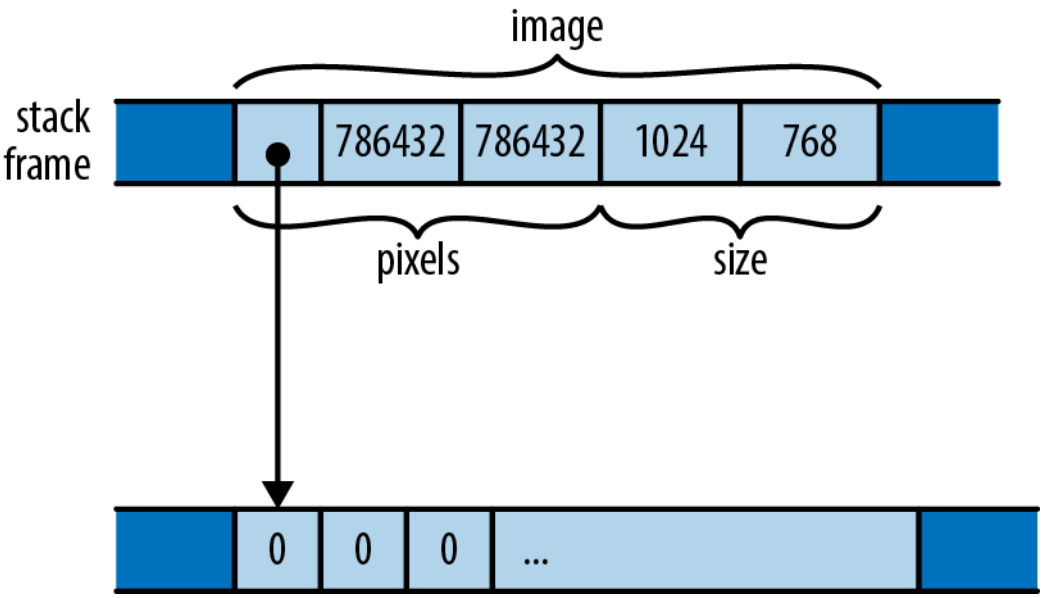

Illustration 9-1. Une `GrayscaleMap` structure en mémoire

Contrairement à C et C++, Rust ne fait pas de promesses spécifiques sur la manière dont il ordonnera les champs ou les éléments d'une structure en mémoire ; ce schéma ne montre qu'un seul agencement 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 les valeurs `pixels` et `size` chacun dans leurs propres blocs alloués par tas et feraient `GrayscaleMap` pointer les champs vers eux, Rust intègre `pixels` et `size` directement dans la `GrayscaleMap` valeur. Seul le tampon alloué au tas appartenant au `pixels` vecteur reste dans son propre bloc.

Vous pouvez demander à Rust de disposer les structures d'une manière compatible avec C et C++, en utilisant l' `#[repr(C)]` attribut. Nous aborderons cela en détail au chapitre 23 .

Définir des méthodes avec impl

À traversle livre que nous appelons des méthodes sur toutes sortes de valeurs. Nous avons poussé des éléments sur des vecteurs avec `v.push(e)` , récupéré leur longueur avec `v.len()` , vérifié les `Result` valeurs pour les erreurs avec `r.expect("msg")` , etc. Vous pouvez également définir des méthodes sur vos propres types de structure. Plutôt que d'apparaître dans la définition de structure, comme en C++ ou Java, les méthodes Rust apparaissent dans un `impl` bloc séparé.

Un `impl` bloc est simplement une collection de `fn` définitions, dont chacune devient une méthode sur le type de structure nommé en haut du bloc. Ici, par exemple, nous définissons un public struct `Queue` , puis lui donnons deux méthodes publiques, `push` et `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 ther
    /// was a character to pop, or `None` if the queue was empty.
    pub fn pop(&mut self) ->Option<char> {
        if self.older.is_empty() {
            if self.younger.is_empty() {
                return None;
            }

            // Bring the elements in younger over to older, and put them i
            // 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 `impl` bloc sont appelées *fonctions associées*, puisqu'ils sont associés à un type spécifique. L'opposé d'une fonction associée est une *fonction libre*, celui qui n'est pas défini comme `impl` élément d'un bloc.

Rust transmet à une méthode la valeur sur laquelle elle est appelée comme premier argument, qui doit avoir le nom spécial `self` . Étant donné `self` que le type de est évidemment celui nommé en haut du `impl` bloc, ou une référence à celui-ci, Rust vous permet d'omettre le type et d'écrire `self` , `&self` ou `&mut self` comme raccourci pour `self: Queue` , `self: &Queue` ou `self: &mut Queue` . Vous pouvez utiliser les formulaires longs si vous le souhaitez, mais presque tout le code Rust utilise le raccourci, comme indiqué précédemment.

Dans notre exemple, les méthodes `push` et font référence 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'identifiants 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 le façon dont les méthodes JavaScript utilisent
`.pop Queue self.older self.younger self self this`

Depuis `push` et `pop` doivent modifier le `Queue` , ils prennent tous les deux `&mut self` . Cependant, lorsque vous appelez une méthode, vous n'avez pas besoin d'emprunter vous-même la référence mutable ; la syntaxe d'appel de méthode ordinaire s'en charge implicitement. Donc, avec ces définitions en place, vous pouvez utiliser `Queue` comme ceci :

```
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);
```

L'écriture simple `q.push(...)` emprunte une référence mutable à `q`, comme si vous aviez écrit `(&mut q).push(...)`, puisque c'est ce que la `push` méthode `self` exige.

Si une méthode n'a pas besoin de modifier son `self`, vous pouvez la définir pour prendre une référence partagée à la place. Par exemple:

```
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 `self`, elle peut prendre `self` par valeur :

```
impl Queue {
    pub fn split(self) ->(Vec<char>, Vec<char>) {
        (self.older, self.younger)
    }
}
```

L'appel de cette `split` méthode ressemble aux autres appels de méthode :

```
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 `split` prend sa `self` par valeur, cela *déplace* le `Queue` hors de `q`, laissant `q` non initialisé. Étant donné `split` que `self` possède maintenant la file d'attente, il est capable d'en retirer les vecteurs individuels et de les renvoyer à l'appelant.

Parfois, prendre `self` par valeur comme celle-ci, ou même par référence, ne suffit pas, donc Rust vous permet également de passer `self` par des types de pointeurs intelligents.

Se faire passer pour une boîte, un Rc ou un arc

`self` L' argument d'une méthodepeut également être un `Box<Self>`, `Rc<Self>`, ou `Arc<Self>`. Une telle méthodene peut être appelée que sur une valeur du type de pointeur donné. L'appel de la méthode lui transmet la propriété du pointeur.

Vous n'aurez généralement pas besoin de le faire. Une méthode qui attend `self` par référence fonctionne correctement lorsqu'elle est appelée sur l'un de ces types de pointeurs :

```
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 aux champs, Rust emprunte automatiquement une référence à partir de types de pointeurs tels que `Box`, `Rc`, et `Arc`, donc `&self` et `&mut self` sont presque toujours la bonne chose dans une signature de méthode, avec occasionnellement `self`.

Mais s'il arrive qu'une méthode nécessite la propriété d'un pointeur sur `Self`, et que ses appelants disposent d'un tel pointeur, Rust vous laissera le passer comme `self` argument de la méthode. Pour ce faire, vous devez épeler le type de `self`, comme s'il s'agissait d'un paramètre ordinaire :

```
impl Node {
    fn append_to(self: Rc<Self>, parent:&mut Node) {
        parent.children.push(self);
    }
}
```

Fonctions associées au type

Un `impl` bloc pour un type donné peut aussi définir des fonctionsqui ne prennent pas `self` du tout comme argument. Ce sont toujours des fonctions associées, puisqu'elles sont dans un `impl` bloc, mais ce ne sont pas des méthodes, puisqu'elles ne prennent pas d' `self` argument. Pour les distinguer des méthodes, nous les appelons *fonctions associées au type* .

Ils sont souvent utilisés pour fournir des fonctions constructeur, comme ceci :

```
impl Queue {
    pub fn new() -> Queue {
        Queue { older: Vec:: new(), younger: Vec::new() }
    }
}
```

Pour utiliser cette fonction, nous nous y référons comme `Queue::new` : le nom du type, un double deux-points, puis le nom de la fonction. Maintenant, notre exemple de code devient un peu plus svelte :

```
let mut q = Queue::new();

q.push('*');
...
```

Il est conventionnel dans Rust que les fonctions constructeur soient nommées `new` ; nous avons déjà vu `Vec::new` , `Box::new` , `HashMap::new` , et d'autres. Mais il n'y a rien de spécial dans le nom `new` . Ce n'est pas un mot-clé, et les types ont souvent d'autres fonctions associées qui servent de constructeurs, comme `Vec::with_capacity` .

Bien que vous puissiez avoir plusieurs blocs distincts `impl` pour un seul type, ils doivent tous se trouver dans la même caisse qui définit ce type. Cependant, Rust vous permet d'attacher vos propres méthodes à d'autres types ; nous expliquerons comment au chapitre 11 .

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 par-

courir des centaines de lignes de définitions de fonctions membres pour vous assurer que vous n'avez manqué aucun des membres de données de la classe ; à Rust, ils sont tous au même endroit.

- Bien que l'on puisse imaginer intégrer des méthodes dans la syntaxe des structures de champ nommé, ce n'est pas si simple pour les structures de type tuple et de type unité. L'extraction de méthodes dans un `impl` 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 des `enum` types et des types primitifs comme `i32` . (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* .)
- La même `impl` syntaxe sert également parfaitement à implémenter des traits, que nous aborderons au chapitre 11 .

Const associés

Une autre caractéristiquedes langages comme C# et Java que Rust adopte dans son système de types est l'idée de valeurs associées à un type, plutôt qu'une instance spécifique de ce type. Dans Rust, ceux-ci sont connus sous le nom de *consts associés* .

Comme son nom l'indique, les constantes associées 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 `Vector2` . Tout comme les fonctions associées, elles sont accessibles en nommant le type auquel elles sont associées, suivi de leur nom :

```
let scaled = Vector2::UNIT.scaled_by(2.0);
```

Un const associé ne doit pas non plus être du même type que le type auquel il est associé ; nous pourrions utiliser cette fonctionnalité pour ajouter des identifiants ou des noms aux types. Par exemple, s'il y avait plusieurs types similaires à ceux `Vector2` qui devaient être écrits dans un fichier puis chargés en mémoire plus tard, un const associé pourrait être utilisé pour ajouter des noms ou des identifiants numériques qui pourraient être écrits à côté des données pour identifier son type :

```
impl Vector2 {
    const NAME: &'static str = "Vector2";
    const ID:u32 = 18;
}
```

Structures génériques

Notre plus tôtLa définition de `Queue` n'est pas satisfaisante : il est écrit pour stocker des caractères, mais il n'y a rien dans sa structure ou ses méthodes qui soit spécifique aux caractères. Si nous devions définir une autre structure contenant, par exemple, `String` des valeurs, le code pourrait être identique, sauf qu'il `char` serait remplacé par `String` . Ce serait une perte de temps.

Heureusement, les structures Rust peuvent être *génériques* , ce qui signifie que leur définition est un modèle dans lequel vous pouvez insérer les types de votre choix. Par exemple, voici une définition pour `Queue` qui peut contenir des valeurs de n'importe quel type :

```
pub struct Queue<T> {
    older: Vec<T>,
    younger:Vec<T>
}
```

Vous pouvez lire le `<T>` dans `Queue<T>` comme "pour tout type d'élément `T` ...". Donc, cette définition se lit comme suit : "Pour tout type `T` , a `Queue<T>` est deux champs de type `Vec<T>` ." Par exemple, dans `Queue<String>` , `T` is `String` , so `older` et `younger` have type `Vec<String>` . Dans `Queue<char>` , `T` est `char` , et nous obtenons une structure identique à la `char` définition spécifique avec laquelle nous

avons commencé. En fait, `Vec` elle-même est une structure générique, définie exactement de cette manière.

Dans les définitions de structure génériques, les noms de type utilisés `<` entre crochets `>` sont appelés *paramètres de type*. Un `impl` bloc pour une structure générique ressemble à ceci :

```
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 `impl<T> Queue<T>` comme quelque chose comme « pour tout type `T` , voici quelques fonctions associées disponibles sur `Queue<T>` ». Ensuite, vous pouvez utiliser le paramètre de type `T` comme type dans les définitions de fonctions associées.

La syntaxe peut sembler un peu redondante, mais `impl<T>` il est clair que le `impl` bloc couvre n'importe quel type `T` , ce qui le distingue d'un `impl` bloc écrit pour un type spécifique de `Queue` , comme celui-ci :

```
impl Queue<f64> {
    fn sum(&self) ->f64 {
        ...
    }
}
```

Cet `impl` en-tête de bloc se lit comme suit : "Voici quelques fonctions associées spécifiquement pour `Queue<f64>` ." Cela donne `Queue<f64>` une `sum` méthode, disponible sur aucun autre type de `Queue` .

Nous avons utilisé le raccourci de Rust pour les `self` paramètres dans le code précédent ; écrire `Queue<T>` partout devient une bouchée et une distraction. Comme autre raccourci, chaque `impl` bloc, générique ou non,

définit le paramètre de type spécial `Self` (notez le `CamelCase` nom) comme étant le type auquel nous ajoutons des méthodes. Dans le code précédent, `Self` serait `Queue<T>`, nous pouvons donc abréger `Queue::new` un peu plus la définition de :

```
pub fn new() -> Self {
    Queue { older: Vec:: new(), younger: Vec::new() }
}
```

Vous avez peut-être remarqué que, dans le corps de `new`, nous n'avions pas besoin d'écrire le paramètre de type dans l'expression de construction ; simplement écrire `Queue { ... }` suffisait. 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, `Queue<T>` —Rust nous fournit le paramètre. Cependant, vous devrez toujours fournir des paramètres de type dans les signatures de fonction et les définitions de type. Rust ne les déduit pas ; à la place, il utilise ces types explicites comme base à partir de laquelle il déduit les types dans les corps de fonction.

`Self` peut également être utilisé de cette manière; nous aurions pu écrire `Self { ... }` à la place. C'est à vous de décider ce que vous trouvez le plus facile à comprendre.

Pour les appels de fonction associés, vous pouvez fournir le paramètre de type explicitement en utilisant la `::<>` notation (turbofish) :

```
let mut q = Queue:: <char>::new();
```

Mais en pratique, vous pouvez généralement laisser Rust le découvrir 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 `Vec`, un autre type de structure générique, tout au long du livre.

Il n'y a pas que les structures qui peuvent être génériques. Les énumérations peuvent également prendre des paramètres de type, avec une syntaxe très similaire. Nous montrerons cela en détail dans « Enums » .

Structures génériques avec paramètres de durée de vie

Comme nous l'avons vu dans « Structures contenant des références » , si un type de structure contient des références, vous devez nommer les durées de vie de ces références. Par exemple, voici une structure qui peut contenir des références aux éléments les plus grands et les plus petits d'une tranche :

```
struct Extrema<'elt> {
    greatest: &'elt i32,
    least:&'elt i32
}
```

Plus tôt, nous vous avons invité à penser à une déclaration comme `struct Queue<T>` signifiant que, étant donné n'importe quel type spécifique `T` , vous pouvez faire un `Queue<T>` contenant ce type. De même, vous pouvez penser `struct Extrema<'elt>` que, compte tenu de toute durée de vie spécifique `'elt` , vous pouvez créer un `Extrema<'elt>` qui contient des références avec cette durée de vie.

Voici une fonction pour analyser une tranche et renvoyer une `Extrema` valeur dont les champs font référence à ses éléments :

```
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 `find_extrema` emprunte des éléments de `slice` , qui a life `'s` , la `Extrema` structure que nous retournons utilise également `'s` comme durée de vie de ses références. Rust déduit toujours les para-

mètres de durée de vie des appels, donc les appels `find_extrema` n'ont pas besoin de les mentionner :

```
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 comme argument, Rust nous permet d'omettre les durées de vie lorsqu'il y a un candidat évident. Nous aurions aussi pu écrire `find_extrema` la signature de comme ceci, sans changement de sens :

```
fn find_extrema(slice: &[i32]) ->Extrema {
    ...
}
```

Certes, nous *aurions* pu vouloir dire `Extrema<'static>`, mais c'est assez inhabituel. Rust fournit un raccourci pour le cas courant.

Structures génériques avec 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 ceci :

```
/// 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, `Polynomial<3>` est un polynôme quadratique, par exemple. La `<const N: usize>` clause indique que le `Polynomial` type attend une `usize` valeur comme paramètre générique, qu'il utilise pour décider du nombre de coefficients à stocker.

Contrairement à `Vec`, qui a des champs contenant sa longueur et sa capacité et stocke ses éléments dans le tas, `Polynomial` 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 `Polynomial`s ne peut pas croître de manière dynamique.)

Nous pouvons utiliser le paramètre `N` dans les fonctions associées au type :

```
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 `new` fonction accepte un tableau de longueur `N` et prend ses éléments comme coefficients d'une nouvelle `Polynomial` valeur. La `eval` méthode itère sur la plage `0..N` pour trouver la valeur du polynôme à un point donné `x`.

Comme pour les paramètres de type et de durée de vie, Rust peut souvent déduire les bonnes valeurs pour les 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 `Polynomial::new` un tableau avec six éléments, Rust sait que nous devons construire un `Polynomial<6>`. La `eval` méthode sait combien d'itérations la `for` boucle doit exécuter simplement en consultant son `Self` type. Comme la longueur est connue au moment de la compilation, le compilateur remplacera probablement entièrement la boucle par du code linéaire.

Un `const` paramètre générique peut être n'importe quel type entier, `char`, ou `bool`. Les nombres à virgule flottante, les énumérations et autres types ne sont pas autorisés.

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 `const` valeurs. Par exemple, un type contenant un tableau de références pourrait être déclaré comme ceci :

```
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 le moment. Par exemple, il aurait été plus agréable de définir `Polynomial` comme ceci :

```
/// 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.e.
```

Bien que ce soit bien de le dire `[f64; N]`, un type comme `[f64; N + 1]` est apparemment trop risqué pour Rust. Mais Rust impose cette restriction pour le moment pour éviter de se confronter à des problèmes comme celui-ci :

```
struct Ketchup<const N: usize> {
    tomayto: [i32; N & !31],
    tomahto:[i32; N - (N % 32)],
}
```

En fait, `N & !31` et `N - (N % 32)` sont égaux pour toutes les valeurs de `N`, donc `tomayto` et `tomahto` 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 manipulation de bits dont il aurait besoin pour être en mesure de reconnaître ce fait risque d'introduire des cas d'angle déroutants dans un aspect du langage qui est déjà assez compliqué. Bien sûr, des expressions simples comme `N + 1` sont beaucoup plus sages, et des travaux sont en cours pour apprendre à Rust à les gérer en douceur.

Étant donné que le problème ici concerne le comportement du vérificateur de type, cette restriction ne s'applique qu'aux paramètres constants apparaissant dans les types, comme la longueur d'un tableau. Dans une expression ordinaire, vous pouvez utiliser `N` comme bon vous semble : `N + 1` et `N & !31` sont parfaitement acceptables .

Si la valeur que vous souhaitez fournir pour un `const` paramètre générique n'est pas simplement un littéral ou un identifiant unique, vous devez l'entourer d'accolades, comme dans `Polynomial<{5 + 1}>`. Cette règle permet à Rust de signaler les erreurs de syntaxe avec plus de précision.

Dérivation de traits communs pour les types de structure

Structurespeut être très simple à écrire :

```
struct Point {
    x: f64,
    y:f64
}
```

Cependant, si vous deviez commencer à utiliser ce `Point` type, vous remarqueriez rapidement que c'est un peu pénible. Comme écrit, `Point` n'est pas copiable ou clonable. Vous ne pouvez pas l'imprimer

avec `println!("{:?}", point);` et il ne prend pas en charge les opérateurs `==` et `!=`.

Chacune de ces fonctionnalités a un nom dans Rust— `Copy`, `Clone`, `Debug` et `PartialEq`. On les appelle *traits*. Au 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 souhaitez une sorte de comportement personnalisé. Rust peut les implémenter automatiquement pour vous, avec une précision mécanique. Ajoutez simplement un `#[derive]` attribut à la structure :

```
#[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. Nous pouvons demander à Rust de dériver `PartialEq` car `Point` ses deux champs sont tous les deux de type `f64`, qui implémente déjà `PartialEq`.

Rust peut également dériver `PartialOrd`, ce qui ajouterait la prise en charge des 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 prendre en charge ces opérateurs pour les `Point` valeurs. Des cas comme celui-ci sont l'une des raisons pour lesquelles Rust nous fait écrire l' `#[derive]` attribut plutôt que de dériver automatiquement tous les traits possibles. Une autre raison est que l'implémentation d'un trait est automatiquement une fonctionnalité publique, donc la copiabilité, la clonage, etc. font toutes partie de l'API publique de votre structure et doivent être choisies délibérément.

Nous décrirons en détail les traits standard de Rust et expliquerons lesquels sont `#[derive]` capables au chapitre 13.

Mutabilité intérieure

Mutabilitéc'est comme n'importe quoi d'autre : en excès, ça cause des problèmes, mais on en veut souvent juste un peu. Par exemple, supposons que votre système de contrôle de robot araignée ait une structure centrale, `SpiderRobot`, qui contient des paramètres et des poignées d'E/S. Il est configuré au démarrage du robot et les valeurs ne changent jamais :

```
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 `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 sites Web, la prédation, le contrôle du flux de venin, etc. ont également toutes un `Rc<SpiderRobot>` pointeur intelligent. Rappel qui `Rc` signifie comptage de références, et une valeur dans une `Rc` boîte est toujours partagée et donc toujours immuable.

Supposons maintenant que vous souhaitiez ajouter un peu de journalisation à la `SpiderRobot` structure, en utilisant le `File` type standard. Il y a un problème : `File` a doit être `mut`. Toutes les méthodes pour y écrire nécessitent une `mut` référence.

Ce genre de situation revient assez souvent. Ce dont nous avons besoin, c'est d'un peu de données modifiables (a `File`) à l'intérieur d'une valeur autrement immuable (la `SpiderRobot` structure). 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 : `Cell<T>` et `RefCell<T>`, tous deux dans le `std::cell` module.

A `Cell<T>` est une structurequi contient une seule valeur privée de type `T`. La seule particularité de a `Cell` est que vous pouvez obtenir et définir le champ même si vous n'avez pas `mut` accès à lui- `Cell` même :

`Cell::new(value)`

Crée un nouveau `Cell`, en y déplaçant le donné `value`.

`cell.get()`

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

`cell.set(value)`

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

Cette méthode prend `self` comme non `mut` référence :

```
fn set(&self, value:T)    // note: not `&mut self`
```

Ceci est, bien sûr, inhabituel pour les méthodes nommées `set`. À l'heure actuelle, Rust nous a appris à nous attendre à ce que nous ayons besoin d' `mut` un accès si nous voulons apporter des modifications aux données. Mais du même coup, ce détail inhabituel est tout l'intérêt de l' `Cell` art. Ils sont simplement un moyen sûr de contourner les règles d'immuabilité, ni plus, ni moins.

Les cellules ont également quelques autres méthodes, que vous pouvez lire dans la documentation.

A `Cell` serait pratique si vous ajoutiez un simple compteur à votre fichier `SpiderRobot`. Vous pourriez écrire :

```
use std:: cell::Cell;

pub struct SpiderRobot {
    ...
    hardware_error_count:Cell<u32>,
    ...
}
```

Ensuite, même les non- `mut` méthodes de `SpiderRobot` peuvent accéder à cela `u32`, en utilisant les méthodes `.get()` et : `.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 simple, mais cela ne résout pas notre problème de journalisation. `Cell` ne vous permet *pas* d'appeler des `mut` méthodes sur une valeur partagée. La `.get()` méthoderenvoie une copie de la valeur dans la cellule, donc cela ne fonctionne que si `T` implémente le `Copy` trait . Pour la journalisation, nous avons besoin d'un mutable `File` , et `File` n'est pas copiable.

Le bon outil dans ce casest un `RefCell` . Comme `Cell<T>` , `RefCell<T>` est un type générique qui contient une seule valeur de type `T` . Contrairement à `Cell` , prend en `RefCell` charge les références d'emprunt à sa `T` valeur :

`RefCell::new(value)`

Créeun nouveau `RefCell` , emménageant `value` dedans.

`ref_cell.borrow()`

Retoura `Ref<T>` , qui est essentiellement juste une référence partagée à la valeur stockée dans `ref_cell` .

Cette méthode panique si la valeur est déjà empruntée de manière mutable ; voir les détails à suivre.

`ref_cell.borrow_mut()`

Retoura `RefMut<T>` , essentiellement une référence mutable à la valeur dans `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()`

Travaillercomme `borrow()` et `borrow_mut()` , mais renvoie un `Result` . Au lieu de paniquer si la valeur est déjà empruntée de manière mutable, ils renvoient une `Err` valeur.

Encore une fois, `RefCell` a quelques autres méthodes, que vous pouvez trouver dans la documentation .

Les deux `borrow` méthodes ne paniquent que si vous essayez d'enfreindre la règle de Rust selon laquelle `mut` les références sont des références exclusives. Par exemple, cela ferait paniquer :

```
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 mettre ces deux emprunts dans des blocs séparés. De cette façon, `r` serait abandonné avant d'essayer d'emprunter `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 de compilation. `RefCell` applique la même règle à l'aide de contrôles d'exécution. Donc, si vous enfreignez les règles, vous obtenez une panique (ou un `Err` , pour `try_borrow` et `try_borrow_mut`).

Nous sommes maintenant prêts `RefCell` à travailler dans notre `SpiderRobot` type :

```
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 `file` est de type `RefMut<File>`. Il peut être utilisé comme une référence mutable à un fichier `File`. Pour plus de détails sur l'écriture dans des fichiers, voir Chapitre 18.

Les cellules sont faciles à utiliser. Devoir appeler `.get()` et `.set()` ou `.borrow()` et `.borrow_mut()` est un peu gênant, mais c'est juste le prix à payer 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 variantes thread-safe de la mutabilité intérieure au chapitre 19, lorsque nous aborderons « Mutex<T> », « Atomics » et « Global Variables ».

Qu'une structure ait des champs nommés ou qu'elle ressemble à un tuple, il s'agit d'une agrégation d'autres valeurs : si j'ai une `SpiderSenses` structure, alors j'ai un `Rc` pointeur vers une structure partagée `SpiderRobot`, et j'ai des yeux, et j'ai un accéléromètre, etc.. Ainsi, l'essence d'une structure est le mot « et » : j'ai un X *et* un Y. Mais que se passerait-il s'il y avait un autre type de type construit autour du mot « ou » ? Autrement dit, lorsque vous avez une valeur d'un tel type, vous auriez *soit* un X, *soit* un Y ? De tels types s'avèrent si utiles qu'ils sont omniprésents dans Rust, et ils font l'objet du chapitre suivant.