Chapitre 5. Références

> *Les bibliothèques ne peuvent pas fournir de nouvelles incapacités.*
>
> —Marc Miller

Tout le pointeurles types que nous avons vus jusqu'à présent (le simple `Box<T>` pointeur de tas et les pointeurs internes à `String` et aux `Vec` valeurs) possèdent des pointeurs : lorsque le propriétaire est supprimé, le référent l'accompagne. Rust a également des types de pointeurs non propriétaires appelés *références* , qui n'ont aucun effet sur la durée de vie de leurs référents.

En fait, c'est plutôt le contraire : les références ne doivent jamais survivre à leurs référents. Vous devez faire apparaître dans votre code qu'aucune référence ne peut survivre à la valeur vers laquelle elle pointe. Pour souligner cela, Rust se réfère à la création d'une référence à une certaine valeur en tant *qu'emprunt*la valeur : ce que vous avez emprunté, vous devez éventuellement le rendre à son propriétaire.

Si vous avez ressenti un moment de scepticisme en lisant la phrase « Vous devez le rendre apparent dans votre code », vous êtes en excellente compagnie. Les références elles-mêmes n'ont rien de spécial - sous le capot, ce ne sont que des adresses. Mais les règles qui assurent leur sécurité sont nouvelles pour Rust ; en dehors des langages de recherche, vous n'aurez jamais rien vu de tel auparavant. Et bien que ces règles soient la partie de Rust qui nécessite le plus d'efforts à maîtriser, l'ampleur des bogues classiques et absolument quotidiens qu'elles empêchent est surprenante, et leur effet sur la programmation multithread est libérateur. C'est encore le pari radical de Rust.

Dans ce chapitre, nous verrons comment les références fonctionnent dans Rust ; montrer comment les références, les fonctions et les types définis par l'utilisateur intègrent tous des informations de durée de vie pour garantir qu'ils sont utilisés en toute sécurité ; et illustrent certaines catégories courantes de bogues que ces efforts évitent, au moment de la compilation et sans pénalités de performances à l'exécution.

Références aux valeurs

Par exemple, supposons que nous allons construire une table d'artistes meurtriers de la Renaissance et des œuvres pour lesquelles ils sont connus. La bibliothèque standard de Rust inclut un type de table de hachage, nous pouvons donc définir notre type comme ceci :

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

type Table = HashMap<String, Vec<String>>;
```

En d'autres termes, il s'agit d'une table de hachage qui mappe des `String` valeurs à des `Vec<String>` valeurs, prenant le nom d'un artiste dans une liste des noms de leurs œuvres. Vous pouvez parcourir les entrées de a `HashMap` avec une `for` boucle, nous pouvons donc écrire une fonction pour imprimer a `Table` :

```
fn show(table:Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Construire et imprimer le tableau est simple :

```
fn main() {
    let mut table = Table::new();
    table.insert("Gesualdo".to_string(),
                 vec!["many madrigals".to_string(),
                      "Tenebrae Responsoria".to_string()]);
    table.insert("Caravaggio".to_string(),
                 vec!["The Musicians".to_string(),
                      "The Calling of St. Matthew".to_string()]);
    table.insert("Cellini".to_string(),
                 vec!["Perseus with the head of Medusa".to_string(),
                      "a salt cellar".to_string()]);

    show(table);
}
```

Et tout fonctionne bien :

```
$course de fret
     Running `/home/jimb/rust/book/fragments/target/debug/fragments`
works by Gesualdo:
  many madrigals
  Tenebrae Responsoria
works by Cellini:
  Perseus with the head of Medusa
  a salt cellar
works by Caravaggio:
  The Musicians
  The Calling of St. Matthew
$
```

Mais si vous avez lu la section du chapitre précédent sur les mouvements, cette définition de `show` devrait soulever quelques questions. En particulier, `HashMap` n'est pas `Copy` — il ne peut pas l'être, puisqu'il possède une table allouée dynamiquement. Ainsi, lorsque le programme appelle `show(table)`, toute la structure est déplacée vers la fonction, laissant la variable `table` non initialisée. (Il parcourt également son contenu sans ordre spécifique, donc si vous avez un ordre différent, ne vous inquiétez pas.) Si le code appelant essaie d'utiliser `table` maintenant, il rencontrera des problèmes :

```
...
show(table);
assert_eq!(table["Gesualdo"][0], "many madrigals");
```

Rust se plaint qu'il `table` n'est plus disponible :

```
error: borrow of moved value: `table`
   |
20 |     let mut table = Table::new();
   |         --------- move occurs because `table` has type
   |                   `HashMap<String, Vec<String>>`,
   |                   which does not implement the `Copy` trait
...
31 |     show(table);
   |          ----- value moved here
32 |     assert_eq!(table["Gesualdo"][0], "many madrigals");
   |                ^^^^^ value borrowed here after move
```

En fait, si nous examinons la définition de `show`, la boucle externe `for` s'approprie la table de hachage et la consomme entièrement ; et la boucle interne `for` fait de même pour chacun des vecteurs. (Nous avons

vu ce comportement plus tôt, dans l'exemple "liberté, égalité, fraternité".) En raison de la sémantique des mouvements, nous avons complètement détruit la structure entière simplement en essayant de l'imprimer. Merci Rust!

La bonne façon de gérer cela est d'utiliser des références. Une référence vous permet d'accéder à une valeur sans affecter sa propriété. Les références sont de deux sortes :

- Une *référence partagée*permet de lire mais pas de modifier son référent. Cependant, vous pouvez avoir autant de références partagées à une valeur particulière à la fois que vous le souhaitez.L'expression `&e` produit une référence partagée à `e` la valeur de ; si `e` a le type `T`, alors `&e` a le type `&T`, prononcé « ref `T` ». Les références partagées sont `Copy`.
- Si vous avez une *référence mutable* à une valeur, vous pouvez à la fois lire et modifier la valeur. Cependant, vous ne pouvez avoir aucune autre référence d'aucune sorte à cette valeur active en même temps. L'expression `&mut e` donne une référence mutable à `e` la valeur de ; vous écrivez son type comme `&mut T`, qui se prononce « ref muet `T` ». Les références mutables ne le sont pas `Copy`.

Vous pouvez considérer la distinction entre les références partagées et mutables comme un moyen d'imposer une *multiplicité de lecteursou auteur unique*règle au moment de la compilation. En fait, cette règle ne s'applique pas uniquement aux références ; elle couvre également le propriétaire de la valeur empruntée. Tant qu'il existe des références partagées à une valeur, même son propriétaire ne peut pas la modifier ; la valeur est verrouillée. Personne ne peut modifier `table` pendant qu'il `show` travaille avec. De même, s'il existe une référence mutable à une valeur, elle a un accès exclusif à la valeur ; vous ne pouvez pas du tout utiliser le propriétaire, jusqu'à ce que la référence mutable disparaisse. Garder le partage et la mutation complètement séparés s'avère essentiel à la sécurité de la mémoire, pour des raisons que nous aborderons plus loin dans ce chapitre.

La fonction d'impression dans notre exemple n'a pas besoin de modifier la table, il suffit de lire son contenu. L'appelant doit donc pouvoir lui transmettre une référence partagée à la table, comme suit :

```
show(&table);
```

Les références ne sont pas propriétairesdes pointeurs, de sorte que la `table` variable reste propriétaire de toute la structure ; `show` vient de l'emprunter un peu. Naturellement, nous devrons ajuster la définition de `show` pour qu'elle corresponde, mais vous devrez regarder attentivement pour voir la différence :

```
fn show(table:&Table) {
    for (artist, works) in table {
        println!("works by {}:", artist);
        for work in works {
            println!("  {}", work);
        }
    }
}
```

Le type du `show` paramètre `table` de est passé de `Table` à `&Table` : au lieu de passer la table par valeur (et donc de déplacer la propriété dans la fonction), nous passons maintenant une référence partagée. C'est le seul changement textuel. Mais comment cela se passe-t-il lorsque nous travaillons à travers le corps ?

`for` Alors que notre boucle externe d'origine s'est emparée du `HashMap` et l'a consommé, dans notre nouvelle version, elle reçoit une référence partagée au `HashMap` . L'itération sur une référence partagée à a `HashMap` est définie pour produire des références partagées à la clé et à la valeur de chaque entrée : `artist` a changé de a `String` à a `&String` , et `works` de a `Vec<String>` à a `&Vec<String>` .

La boucle intérieure est modifiée de la même manière. L'itération sur une référence partagée à un vecteur est définie pour produire des références partagées à ses éléments, il en `work` va de même maintenant pour un `&String` . Aucune propriété ne change de mains dans cette fonction ; il ne s'agit que de faire circuler des références non propriétaires.

Maintenant, si nous voulions écrire une fonction pour classer par ordre alphabétique les œuvres de chaque artiste, une référence partagée ne suffit pas, car les références partagées ne permettent pas de modification. Au lieu de cela, la fonction de tri doit prendre une référence mutable à la table :

```
fn sort_works(table:&mut Table) {
    for (_artist, works) in table {
        works.sort();
```

```
        }
    }
```

Et nous devons en passer un :

```
sort_works(&mut table);
```

Cet emprunt mutable donne `sort_works` la possibilité de lire et de modifier notre structure, comme l'exige la méthode des vecteurs `sort` .

Lorsque nous passons une valeur à une fonction d'une manière qui transfère la propriété de la valeur à la fonction, nous disons que nous l'avons transmise *par valeur*. Si nous passons à la place à la fonction une référence à la valeur, nous disons que nous avons passé la valeur *par référence* . Par exemple, nous avons corrigé notre `show` fonction en la modifiant pour accepter la table par référence plutôt que par valeur. De nombreuses langues établissent cette distinction, mais elle est particulièrement importante dans Rust, car elle explique comment la propriété est affectée.

Travailler avec des références

L'exemple précédent montre une utilisation assez typique des références : permettre aux fonctions d'accéder ou de manipuler une structure sans en prendre possession. Mais les références sont plus flexibles que cela, alors regardons quelques exemples pour avoir une vue plus détaillée de ce qui se passe.

Références Rust versus références C++

Si vous êtes familieravec des références en C++, elles ont quelque chose en commun avec les références Rust. Plus important encore, ce ne sont que des adresses au niveau de la machine. Mais en pratique, les références de Rust ont une sensation très différente.

En C++, les références sont créées implicitement par conversion et déréférencées implicitement également :

```
// C++ code!
int x = 10;
int &r = x;             // initialization creates reference implicitly
```

```
assert(r == 10);          // implicitly dereference r to see x's value
r = 20;                   // stores 20 in x, r itself still points to x
```

Dans Rust, les références sont créées explicitement avec l' `&` opérateur et déréférencées explicitement avec l' `*` opérateur:

```
// Back to Rust code from this point onward.
let x = 10;
let r = &x;               // &x is a shared reference to x
assert!(*r == 10);        // explicitly dereference r
```

Pour créer une référence mutable, utilisez l' `&mut` opérateur :

```
let mut y = 32;
let m = &mut y;          // &mut y is a mutable reference to y
*m += 32;                // explicitly dereference m to set y's value
assert!(*m == 64);       // and to see y's new value
```

Mais vous vous souviendrez peut-être que, lorsque nous avons fixé la `show` fonction pour prendre la table des artistes par référence plutôt que par valeur, nous n'avons jamais eu à utiliser l' `*` opérateur. Pourquoi donc?

Étant donné que les références sont si largement utilisées dans Rust, l' `.` opérateur déréférence implicitement son opérande gauche, si nécessaire :

```
struct Anime { name: &'static str, bechdel_pass: bool }
let aria = Anime { name: "Aria: The Animation", bechdel_pass:true };
let anime_ref = &aria;
assert_eq!(anime_ref.name, "Aria: The Animation");

// Equivalent to the above, but with the dereference written out:
assert_eq!((*anime_ref).name, "Aria: The Animation");
```

La `println!` macro utilisée dans la `show` fonction s'étend au code qui utilise l' `.` opérateur, elle tire donc également parti de ce déréférencement implicite.

L' `.` opérateurpeut également emprunter implicitement une référence à son opérande gauche, si nécessaire pour un appel de méthode. Par exemple, `Vec` la `sort` méthode de prend une référence mutable au vecteur, donc ces deux appels sont équivalents :

```
let mut v = vec![1973, 1968];
v.sort();           // implicitly borrows a mutable reference to v
(&mut v).sort();    // equivalent, but more verbose
```

En un mot, alors que C++ convertit implicitement entre les références et les lvalues (c'est-à-dire des expressions faisant référence à des emplacements en mémoire), ces conversions apparaissant partout où elles sont nécessaires, dans Rust vous utilisez les opérateurs `&` et `*` pour créer et suivre des références, à l'exception de l' `.` opérateur, qui emprunte et déréférence implicitement.

Affectation de références

Affectationune référence à une variable fait pointer cette variable quelque part de nouveau :

```
let x = 10;
let y = 20;
let mut r = &x;

if b { r = &y; }

assert!(*r == 10 || *r == 20);
```

La référence `r` pointe initialement vers `x` . Mais si `b` est vrai, le code le pointe à la `y` place, comme illustré à la Figure 5-1 .

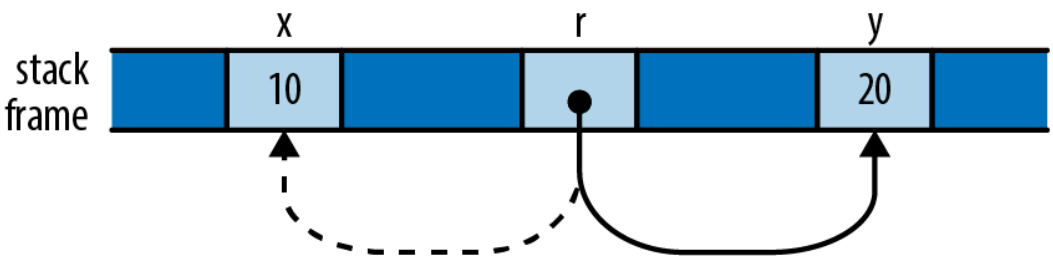

Figure 5-1. La référence `r` , pointant maintenant vers au `y` lieu de `x`

Ce comportement peut sembler trop évident pour mériter d'être mentionné : bien sûr `r` pointe maintenant vers `y` , puisque nous y avons stocké `&y` . Mais nous le soulignons car les références C++ se comportent très différemment : comme indiqué précédemment, l'attribution d'une valeur à une référence en C++ stocke la valeur dans son référent. Une fois qu'une référence C++ a été initialisée, il n'y a aucun moyen de la faire pointer vers autre chose.

Références aux références

Rust autorise les références aux références:

```
struct Point { x: i32, y: i32 }
let point = Point { x: 1000, y: 729 };
let r: &Point = &point;
let rr: &&Point = &r;
let rrr:&&&Point = &rr;
```

(Nous avons écrit les types de référence pour plus de clarté, mais vous pouvez les omettre ; il n'y a rien ici que Rust ne puisse déduire par lui-même.) L' `.` opérateur suit autant de références qu'il en faut pour trouver sa cible :

```
assert_eq!(rrr.y, 729);
```

En mémoire, les références sont disposées comme illustré à la Figure 5-2 .

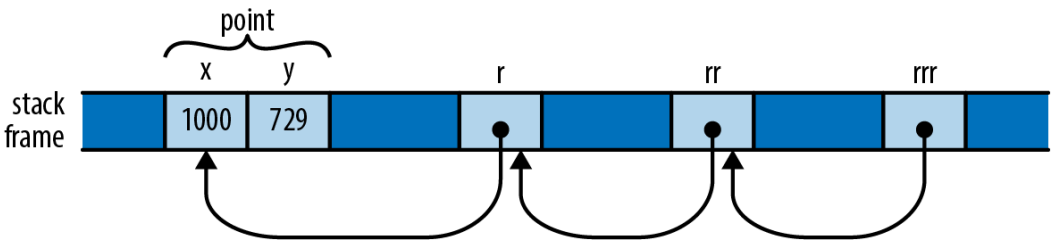

Figure 5-2. Une chaîne de références à références

Ainsi, l'expression `rrr.y` , guidée par le type de `rrr` , traverse en fait trois références pour atteindre le `Point` avant de récupérer son `y` champ.

Comparer des références

Comme l' `.` opérateur, la comparaison de Rustles opérateurs "voient à travers" n'importe quel nombre de références :

```
let x = 10;
let y = 10;

let rx = &x;
let ry = &y;

let rrx = &rx;
let rry = &ry;
```

```
assert!(rrx <= rry);
assert!(rrx == rry);
```

L'assertion finale réussit ici, même si `rrx` et `rry` pointent vers des valeurs différentes (à savoir, `rx` et `ry`), car l' `==` opérateur suit toutes les références et effectue la comparaison sur leurs cibles finales, `x` et `y` . C'est presque toujours le comportement que vous souhaitez, en particulier lors de l'écriture de fonctions génériques. Si vous voulez réellement savoir si deux références pointent vers la même mémoire, vous pouvez utiliser `std::ptr::eq` , qui les compare en tant qu'adresses :

```
assert!(rx == ry);              // their referents are equal
assert!(!std:: ptr::eq(rx, ry)); // but occupy different addresses
```

Notez que les opérandes d'une comparaison doivent avoir exactement le même type, y compris les références :

```
assert!(rx == rrx);    // error: type mismatch: `&i32` vs `&&i32`
assert!(rx == *rrx);   // this is okay
```

Les références ne sont jamais nulles

Les références de rouille ne sont jamais nulles. Il n'y a pas d'analogue au C `NULL` ou au C++ `nullptr` . Il n'y a pas de valeur initiale par défaut pour une référence (vous ne pouvez utiliser aucune variable tant qu'elle n'a pas été initialisée, quel que soit son type) et Rust ne convertira pas les entiers en références (en dehors du `unsafe` code), vous ne pouvez donc pas convertir zéro en une référence.

Le code C et C++ utilise souvent un pointeur nul pour indiquer l'absence de valeur : par exemple, la `malloc` fonction renvoie soit un pointeur vers un nouveau bloc de mémoire, soit `nullptr` s'il n'y a pas assez de mémoire disponible pour satisfaire la requête. Dans Rust, si vous avez besoin d'une valeur qui soit une référence à quelque chose ou non, utilisez le type `Option<&T>` . Au niveau de la machine, Rust représente `None` un pointeur nul et `Some(r)` , où `r` est une `&T` valeur, une adresse différente de zéro, `Option<&T>` est donc tout aussi efficace qu'un pointeur nullable en C ou C++, même s'il est plus sûr : son type vous oblige à vérifier s'il est `None` avant de pouvoir l'utiliser.

Emprunter des références à des expressions arbitraires

Alors que C et C++ ne vous permettent d'appliquer l' `&` opérateur qu'à certains types d'expressions, Rust vous permet d'emprunterune référence à la valeur de n'importe quel type d'expression :

```
fn factorial(n: usize) ->usize {
    (1..n+1).product()
}
let r = &factorial(6);
// Arithmetic operators can see through one level of references.
assert_eq!(r + &1009, 1729);
```

Dans des situations comme celle-ci, Rust crée simplement une variable anonyme pour contenir la valeur de l'expression et en fait le point de référence. La durée de vie de cette variable anonyme dépend de ce que vous faites avec la référence :

- Si vous affectez immédiatement la référence à une variable dans une `let` instruction (ou en faites une partie d'une structure ou d'un tableau qui est immédiatement affecté), alors Rust fait vivre la variable anonyme tant que la variable `let` s'initialise. Dans l'exemple précédent, Rust ferait cela pour le référent de `r` .
- Sinon, la variable anonyme vit jusqu'à la fin de l'instruction englobante. Dans notre exemple, la variable anonyme créée pour contenir `1009` ne dure que jusqu'à la fin de l' `assert_eq!` instruction.

Si vous êtes habitué à C ou C++, cela peut sembler source d'erreurs. Mais rappelez-vous que Rust ne vous laissera jamais écrire de code qui produirait une référence pendante. Si la référence peut être utilisée au-delà de la durée de vie de la variable anonyme, Rust vous signalera toujours le problème au moment de la compilation. Vous pouvez ensuite corriger votre code pour conserver le référent dans une variable nommée avec une durée de vie appropriée.

Références aux tranches et aux objets de trait

Les referencesnous avons montré jusqu'ici sont toutes des adresses simples. Cependant, Rust comprend également deux types de *pointeurs gras*, des valeurs de deux mots portant l'adresse d'une certaine valeur,

ainsi que d'autres informations nécessaires pour mettre la valeur à utiliser.

Une référence à une tranche est un pointeur gras, portant l'adresse de départ de la tranche et sa longueur. Nous avons décrit les tranches en détail au chapitre 3 .

L'autre type de pointeur gras de Rust est un *objet de trait* , une référence à une valeur qui implémente un certain trait. Un objet de trait porte l'adresse d'une valeur et un pointeur vers l'implémentation du trait appropriée à cette valeur, pour invoquer les méthodes du trait. Nous couvrirons les objets de trait en détail dans "Objets de trait" .

En plus de transporter ces données supplémentaires, les références d'objets slice et trait se comportent exactement comme les autres sortes de références que nous avons montrées jusqu'ici dans ce chapitre : elles ne possèdent pas leurs référents, elles ne sont pas autorisées à survivre à leurs référents, elles peuvent être mutable ou partagé, et ainsi de suite.

Sécurité de référence

Comme nous les avons présentées jusqu'ici, les référencesressemblent à peu près à des pointeurs ordinaires en C ou C++. Mais ceux-ci ne sont pas sûrs; comment Rust garde-t-il ses références sous contrôle ? Peut-être que la meilleure façon de voir les règles en action est d'essayer de les enfreindre.

Pour transmettre les idées fondamentales, nous commencerons par les cas les plus simples, montrant comment Rust garantit que les références sont utilisées correctement dans un corps de fonction unique. Ensuite, nous verrons comment passer des références entre les fonctions et les stocker dans des structures de données. Il s'agit de donner auxdites fonctions et types de données *des paramètres de durée de vie*, que nous allons vous expliquer. Enfin, nous présenterons quelques raccourcis fournis par Rust pour simplifier les modèles d'utilisation courants. Tout au long, nous montrerons comment Rust signale le code défectueux et suggère souvent des solutions.

Emprunter une variable locale

Voici un cas assez évident. Vous ne pouvez pas emprunterune référence à une variable locale et sortez-la de la portée de la variable :

```
{
    let r;
    {
        let x = 1;
        r = &x;
    }
    assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
}
```

Le compilateur Rust rejette ce programme, avec un message d'erreur détaillé :

```
error: `x` does not live long enough
   |
7  |         r = &x;
   |             ^^ borrowed value does not live long enough
8  |     }
   |     - `x` dropped here while still borrowed
9  |     assert_eq!(*r, 1);  // bad: reads memory `x` used to occupy
10 | }
```

La plainte de Rust est qu'il `x` ne vit que jusqu'à la fin du bloc intérieur, alors que la référence reste vivante jusqu'à la fin du bloc extérieur, ce qui en fait un pointeur suspendu, ce qui est verboten.

Bien qu'il soit évident pour un lecteur humain que ce programme est cassé, il vaut la peine de regarder comment Rust lui-même est parvenu à cette conclusion. Même cet exemple simple montre les outils logiques que Rust utilise pour vérifier un code beaucoup plus complexe.

Rust essaie d'attribuer à chaque type de référence de votre programme une *durée* de vie qui respecte les contraintes imposées par son utilisation. Une durée de vie est une partie de votre programme pour laquelle une référence peut être utilisée en toute sécurité : une instruction, une expression, la portée d'une variable, etc. Les durées de vie sont entièrement le fruit de l'imagination de Rust au moment de la compilation. Au moment de l'exécution, une référence n'est rien d'autre qu'une adresse ; sa durée de vie fait partie de son type et n'a pas de représentation à l'exécution.

Dans cet exemple, il y a trois vies dont nous devons établir les relations. Les variables `r` et `x` les deux ont une durée de vie, s'étendant du moment où elles sont initialisées jusqu'au moment où le compilateur peut prouver qu'elles ne sont plus utilisées. La troisième durée de vie est celle d'un type

de référence : le type de la référence que nous empruntons `x` et stockons dans `r`.

Voici une contrainte qui devrait sembler assez évidente : si vous avez une variable `x`, alors une référence à `x` ne doit pas survivre à `x` elle-même, comme le montre la figure 5-3.

Au-delà du point où `x` sort de la portée, la référence serait un pointeur pendant. On dit que la durée de vie de la variable doit *contenir* ou *enfermer* celle de la référence qui lui est empruntée.

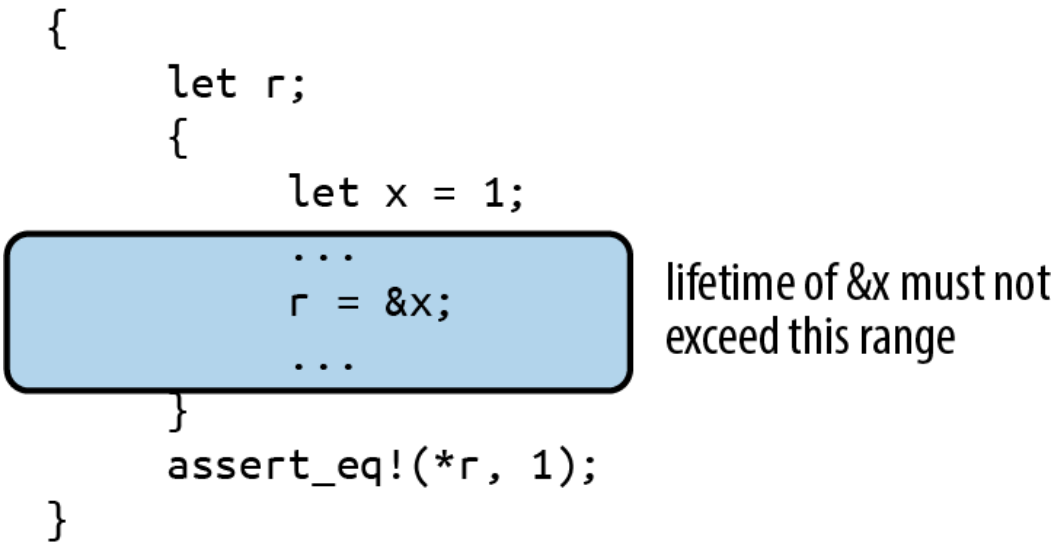

Figure 5-3. Durées de vie autorisées pour &x

Voici un autre type de contrainte : si vous stockez une référence dans une variable `r`, le type de la référence doit être bon pour toute la durée de vie de la variable, depuis son initialisation jusqu'à sa dernière utilisation, comme illustré à la Figure 5-4.

Si la référence ne peut pas vivre au moins aussi longtemps que la variable, alors à un moment donné `r`, il y aura un pointeur pendant. On dit que la durée de vie de la référence doit contenir ou enfermer celle de la variable.

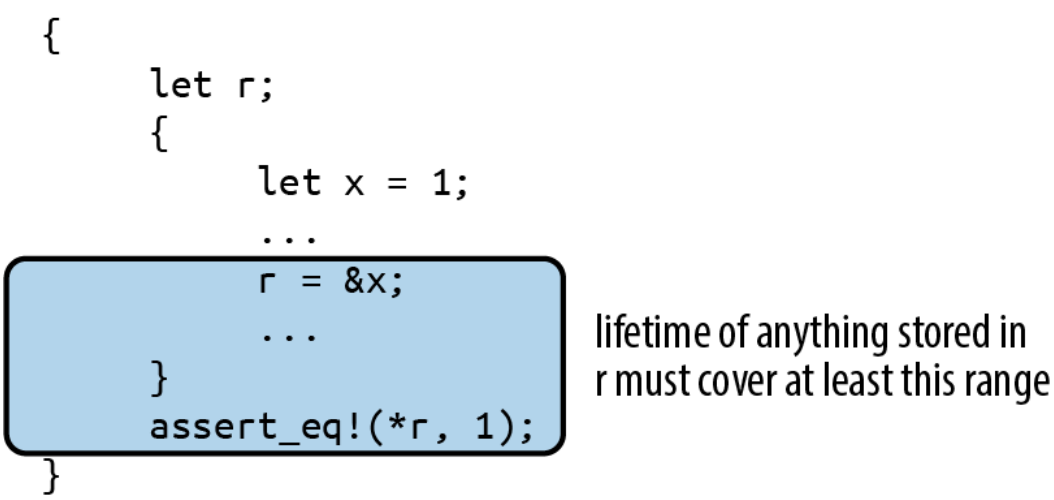

Figure 5-4. Durées de vie admissibles pour la référence stockée dans r

Le premier type de contrainte limite la durée de vie d'une référence, tandis que le second limite sa taille. Rust essaie simplement de trouver une durée de vie pour chaque référence qui satisfait toutes ces contraintes. Dans notre exemple, cependant, cette durée de vie n'existe pas, comme le montre la Figure 5-5 .

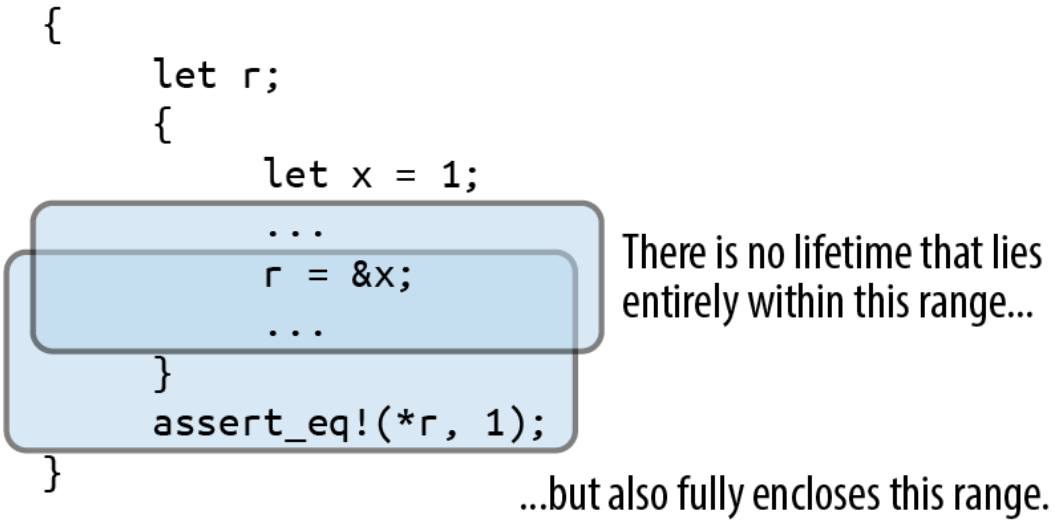

Figure 5-5. Une référence aux contraintes contradictoires sur sa durée de vie

Considérons maintenant un exemple différent où les choses fonctionnent. Nous avons les mêmes types de contraintes : la durée de vie de la référence doit être contenue par `x` 's, mais enfermer complètement `r` 's. Mais comme `r` la durée de vie de est maintenant plus petite, il existe une durée de vie qui respecte les contraintes, comme le montre la figure 5-6 .

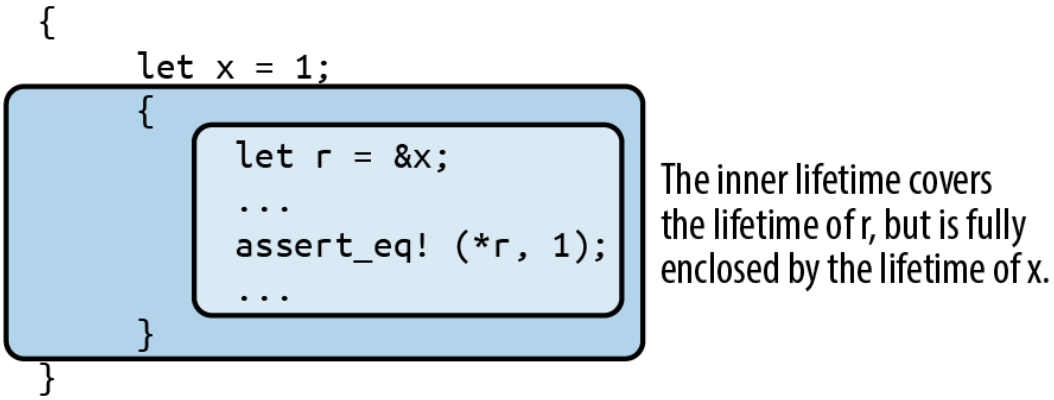

Figure 5-6. Une référence avec une durée de vie englobant `r` la portée de , mais dans `x` la portée de

Ces règles s'appliquent de manière naturelle lorsque vous empruntez une référence à une partie d'une structure de données plus large, comme un élément d'un vecteur :

```
let v = vec![1, 2, 3];
let r = &v[1];
```

Puisque `v` possède le vecteur, qui possède ses éléments, la durée de vie de `v` doit contenir celle du type de référence de `&v[1]`. De même, si vous stockez une référence dans une structure de données, sa durée de vie doit contenir celle de la structure de données. Par exemple, si vous construisez un vecteur de références, toutes doivent avoir des durées de vie renfermant celle de la variable qui possède le vecteur.

C'est l'essence du processus que Rust utilise pour tout le code. Apporter plus de fonctionnalités de langage dans l'image - par exemple, les structures de données et les appels de fonctions - introduit de nouvelles sortes de contraintes, mais le principe reste le même : premièrement, comprendre les contraintes résultant de la façon dont le programme utilise les références ; puis, trouvez des durées de vie qui les satisfont. Ce n'est pas si différent du processus que les programmeurs C et C++ s'imposent à eux-mêmes ; la différence est que Rust connaît les règles et les applique.

Recevoir des références en tant qu'arguments de fonction

Quand on passe une référenceà une fonction, comment Rust s'assure-t-il que la fonction l'utilise en toute sécurité ? Supposons que nous ayons une fonction `f` qui prend une référence et la stocke dans une variable globale. Nous devrons apporter quelques révisions à cela, mais voici une première coupe :

```
// This code has several problems, and doesn't compile.
static mut STASH: &i32;
fn f(p:&i32) { STASH = p; }
```

L'équivalent de Rust d'une variable globale est appelé un *statique*: c'est une valeur créée au démarrage du programme et qui dure jusqu'à ce qu'il se termine. (Comme toute autre déclaration, le système de module de Rust contrôle où les statiques sont visibles , donc elles ne sont que "globales" dans leur durée de vie, pas leur visibilité.) règles que le code qui vient d'être affiché ne suit pas :

- Chaque statique doit être initialisé.
- Les statiques mutables ne sont par nature pas thread-safe (après tout, n'importe quel thread peut accéder à un statique à tout moment), et même dans les programmes à un seul thread, ils peuvent être la proie d'autres types de problèmes de réentrance. Pour ces raisons, vous ne pouvez accéder à un statique mutable qu'à l'intérieur d'un `unsafe` bloc. Dans cet exemple, nous ne sommes pas concernés par ces problèmes particuliers, nous allons donc ajouter un `unsafe` bloc et passer à autre chose.

Avec ces révisions effectuées, nous avons maintenant ce qui suit :

```
static mut STASH: &i32 = &128;
fn f(p:&i32) { // still not good enough
    unsafe {
        STASH = p;
    }
}
```

Nous avons presque terminé. Pour voir le problème restant, nous devons écrire quelques éléments que Rust nous laisse utilement omettre. La signature de `f` tel qu'écrit ici est en fait un raccourci pour ce qui suit :

```
fn f<'a>(p:&'a i32) { ... }
```

Ici, la durée de vie `'a` (prononcez "tick A") est un *paramètre de durée de vie*de `f` . Vous pouvez lire `<'a>` "pour toute durée de vie `'a` ", donc lorsque nous écrivons `fn f<'a>(p: &'a i32)` , nous définissons une fonction qui prend une référence à un `i32` avec une durée de vie donnée `'a` .

Puisque nous devons autoriser `'a` n'importe quelle durée de vie, il vaudrait mieux que les choses se passent si c'est la plus petite durée de vie possible : une qui enferme juste l'appel à `f`. Cette affectation devient alors un point de discorde :

```
STASH = p;
```

Puisque `STASH` vit pour toute l'exécution du programme, le type de référence qu'il contient doit avoir une durée de vie de la même longueur ; Rust appelle cela la *`'static` durée* de vie. Mais la durée de vie de `p` la référence est some `'a`, qui peut être n'importe quoi, tant qu'elle contient l'appel à `f`. Ainsi, Rust rejette notre code :

```
error: explicit lifetime required in the type of `p`
  |
5 |         STASH = p;
  |                 ^ lifetime `'static` required
```

À ce stade, il est clair que notre fonction ne peut pas accepter n'importe quelle référence comme argument. Mais comme le souligne Rust, il devrait être capable d'accepter une référence qui a une `'static` durée de vie : stocker une telle référence dans `STASH` ne peut pas créer de pointeur pendant. Et en effet, le code suivant se compile parfaitement :

```
static mut STASH:&i32 = &10;

fn f(p:&'static i32) {
    unsafe {
        STASH = p;
    }
}
```

Cette fois, `f` la signature de précise qu'il `p` doit s'agir d'une référence avec life `'static`, il n'y a donc plus de problème à stocker cela dans `STASH`. Nous ne pouvons appliquer que `f` des références à d'autres statiques, mais c'est la seule chose qui ne restera pas en `STASH` suspens de toute façon. Alors on peut écrire :

```
static WORTH_POINTING_AT:i32 = 1000;
f(&WORTH_POINTING_AT);
```

Étant donné `WORTH_POINTING_AT` que est un statique, le type de `&WORTH_POINTING_AT` est `&'static i32` , qui peut être transmis en toute sécurité à `f` .

Prenez du recul, cependant, et remarquez ce qui est arrivé à `f` la signature de lorsque nous avons modifié notre chemin vers l'exactitude : l'original `f(p: &i32)` s'est terminé par `f(p: &'static i32)` . En d'autres termes, nous étions incapables d'écrire une fonction qui stockait une référence dans une variable globale sans refléter cette intention dans la signature de la fonction. Dans Rust, la signature d'une fonction expose toujours le comportement du corps.

Inversement, si nous voyons une fonction avec une signature comme `g(p: &i32)` (ou avec les durées de vie écrites, `g<'a>(p: &'a i32)`), nous pouvons dire qu'elle *ne* cache pas son argument `p` n'importe où qui survivra à l'appel. Il n'est pas nécessaire d'examiner `g` la définition de ; la signature seule nous dit ce qui `g` peut et ne peut pas faire avec son argument. Ce fait finit par être très utile lorsque vous essayez d'établir la sécurité d'un appel à la fonction.

Passer des références aux fonctions

À présentque nous avons montré comment la signature d'une fonction est liée à son corps, examinons comment elle est liée aux appelants de la fonction. Supposons que vous ayez le code suivant :

```
// This could be written more briefly: fn g(p: &i32),
// but let's write out the lifetimes for now.
fn g<'a>(p:&'a i32) { ... }

let x = 10;
g(&x);
```

À partir `g` de la seule signature de , Rust sait qu'il n'enregistrera `p` aucun élément susceptible de survivre à l'appel : toute durée de vie qui entoure l'appel doit fonctionner pour `'a` . Donc Rust choisit la plus petite durée de vie possible pour `&x` : celle de l'appel à `g` . Cela répond à toutes les contraintes : il ne survit pas à `x` , et il contient l'intégralité de l'appel à `g` . Donc, ce code passe le cap.

Notez que bien que `g` prenant un paramètre de durée de vie `'a` , nous n'avons pas besoin de le mentionner lors de l'appel `g` . Vous n'avez qu'à vous soucier des paramètres de durée de vie lors de la définition des

fonctions et des types ; lors de leur utilisation, Rust déduit les durées de vie pour vous.

Et si nous essayions de passer `&x` à notre fonction `f` précédente qui stocke son argument dans un statique ?

```
fn f(p:&'static i32) { ... }

let x = 10;
f(&x);
```

Cela échoue à compiler : la référence `&x` ne doit pas survivre à `x`, mais en la passant à `f`, nous la contraignons à vivre au moins aussi longtemps que `'static`. Il n'y a aucun moyen de satisfaire tout le monde ici, donc Rust rejette le code.

Renvoyer des références

C'est courantpour qu'une fonction prenne une référence à une structure de données, puis renvoie une référence dans une partie de cette structure. Par exemple, voici une fonction qui renvoie une référence au plus petit élément d'une tranche :

```
// v should have at least one element.
fn smallest(v: &[i32]) ->&i32 {
    let mut s = &v[0];
    for r in &v[1..] {
        if *r < *s { s = r; }
    }
    s
}
```

Nous avons omis les durées de vie de la signature de cette fonction de la manière habituelle. Lorsqu'une fonction prend une seule référence comme argument et renvoie une seule référence, Rust suppose que les deux doivent avoir la même durée de vie. Écrire cela explicitement nous donnerait:

```
fn smallest<'a>(v: &'a [i32]) ->&'a i32 { ... }
```

Supposons que nous appelons `smallest` ainsi :

```
let s;
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    s = smallest(&parabola);
}
assert_eq!(*s, 0); // bad: points to element of dropped array
```

À partir `smallest` de la signature de , nous pouvons voir que son argument et sa valeur de retour doivent avoir la même durée de vie, `'a` .
Dans notre appel, l'argument `&parabola` ne doit pas survivre à `parabola` lui-même, mais `smallest` la valeur de retour de doit vivre au moins aussi longtemps que `s` . Il n'y a pas de durée de vie possible `'a` qui puisse satisfaire les deux contraintes, donc Rust rejette le code :

```
error: `parabola` does not live long enough
   |
11 |         s = smallest(&parabola);
   |                      -------- borrow occurs here
12 |     }
   |     ^ `parabola` dropped here while still borrowed
13 |     assert_eq!(*s, 0); // bad: points to element of dropped array
   |                - borrowed value needs to live until here
14 | }
```

Déplacer `s` de sorte que sa durée de vie soit clairement contenue dans `parabola` 's résout le problème :

```
{
    let parabola = [9, 4, 1, 0, 1, 4, 9];
    let s = smallest(&parabola);
    assert_eq!(*s, 0); // fine: parabola still alive
}
```

Les durées de vie dans les signatures de fonction permettent à Rust d'évaluer les relations entre les références que vous transmettez à la fonction et celles que la fonction renvoie, et elles garantissent qu'elles sont utilisées en toute sécurité.

Structures contenant des références

Comment Rust gère-t-il les référencesstocké dans des structures de données ? Voici le même programme erroné que nous avons vu plus tôt, sauf que nous avons placé la référence dans une structure :

```
// This does not compile.
struct S {
    r:&i32
}

let s;
{
    let x = 10;
    s = S { r:&x };
}
assert_eq!(*s.r, 10); // bad: reads from dropped `x`
```

Les contraintes de sécurité que Rust place sur les références ne peuvent pas disparaître comme par magie simplement parce que nous avons caché la référence dans une structure. D'une manière ou d'une autre, ces contraintes doivent finir par s'appliquer `S` également. En effet, Rust est sceptique :

```
error: missing lifetime specifier
  |
7 |         r: &i32
  |            ^ expected lifetime parameter
```

Chaque fois qu'un type de référence apparaît dans la définition d'un autre type, vous devez écrire sa durée de vie. Vous pouvez écrire ceci :

```
struct S {
    r:&'static i32
}
```

Cela signifie que `r` cela ne peut faire référence qu'à des `i32` valeurs qui dureront pendant toute la durée de vie du programme, ce qui est plutôt limitatif. L'alternative est de donner au type un paramètre de durée de vie `'a` et de l'utiliser pour `r` :

```
struct S<'a> {
    r:&'a i32
}
```

Désormais, le `S` type a une durée de vie, tout comme les types de référence. Chaque valeur que vous créez de type `S` obtient une nouvelle durée de vie `'a` , qui devient contrainte par la façon dont vous utilisez la valeur. La durée de vie de toute référence dans laquelle vous stockez `r` doit

contenir `'a`, et `'a` doit durer plus longtemps que la durée de vie de l'endroit où vous stockez le fichier `S`.

En revenant au code précédent, l'expression `S { r: &x }` crée une nouvelle `S` valeur avec une durée de vie `'a`. Lorsque vous stockez `&x` sur le `r` terrain, vous vous engagez `'a` à mentir entièrement pendant `x` la durée de vie de .

L'affectation `s = S { ... }` stocke ceci `S` dans une variable dont la durée de vie s'étend jusqu'à la fin de l'exemple, contraignant `'a` à durer plus longtemps que la durée de vie de `s`. Et maintenant, Rust est arrivé aux mêmes contraintes contradictoires qu'auparavant : `'a` ne doit pas survivre à `x`, mais doit vivre au moins aussi longtemps que `s`. Aucune durée de vie satisfaisante n'existe et Rust rejette le code. Catastrophe évitée !

Comment un type avec un paramètre de durée de vie se comporte-t-il lorsqu'il est placé dans un autre type ?

```
struct D {
    s:S  // not adequate
}
```

Rust est sceptique, tout comme il l'était lorsque nous avons essayé de placer une référence dans `S` sans spécifier sa durée de vie :

```
error: missing lifetime specifier
  |
8 |     s: S  // not adequate
  |        ^ expected named lifetime parameter
  |
```

Nous ne pouvons pas laisser de côté `S` le paramètre de durée de vie de : Rust a besoin de savoir comment `D` la durée de vie de est liée à celle de la référence dans son `S` afin d'appliquer les mêmes vérifications `D` que pour `S` les références simples.

On pourrait donner `s` la `'static` durée de vie. Cela marche:

```
struct D {
    s:S<'static>
}
```

Avec cette définition, le `s` champ ne peut emprunter que des valeurs qui vivent pendant toute l'exécution du programme. C'est quelque peu restrictif, mais cela signifie qu'il `D` est impossible d'emprunter une variable locale ; il n'y a pas de contraintes particulières sur `D` la durée de vie de .

Le message d'erreur de Rust suggère en fait une autre approche, plus générale :

```
help: consider introducing a named lifetime parameter
  |
7 | struct D<'a> {
8 |     s: S<'a>
  |
```

Ici, nous donnons `D` son propre paramètre de durée de vie et le passons à `S` :

```
struct D<'a> {
    s:S<'a>
}
```

En prenant un paramètre de durée de vie `'a` et en l'utilisant dans `s` le type de , nous avons permis à Rust de relier `D` la durée de vie de la valeur à celle de la référence qu'elle `S` contient.

Nous avons montré précédemment comment la signature d'une fonction expose ce qu'elle fait avec les références que nous lui passons. Maintenant, nous avons montré quelque chose de similaire à propos des types : les paramètres de durée de vie d'un type révèlent toujours s'il contient des références avec des durées de vie intéressantes (c'est-à-dire non `'static`) et quelles peuvent être ces durées de vie.

Par exemple, supposons que nous ayons une fonction d'analyse qui prend une tranche d'octets et renvoie une structure contenant les résultats de l'analyse :

```
fn parse_record<'i>(input: &'i [u8]) ->Record<'i> { ... }
```

Sans regarder du tout dans la définition du `Record` type, nous pouvons dire que, si nous recevons un `Record` from `parse_record` , toutes les références qu'il contient doivent pointer vers le tampon d'entrée que nous

avons transmis, et nulle part ailleurs (sauf peut-être aux `'static` valeurs).

En fait, cette exposition du comportement interne est la raison pour laquelle Rust exige que les types contenant des références prennent des paramètres de durée de vie explicites. Il n'y a aucune raison pour que Rust ne puisse pas simplement créer une durée de vie distincte pour chaque référence dans la structure et vous éviter d'avoir à les écrire. Les premières versions de Rust se comportaient en fait de cette façon, mais les développeurs trouvaient cela déroutant : il est utile de savoir quand une valeur emprunte quelque chose à une autre valeur, en particulier lorsque l'on travaille sur des erreurs.

Ce ne sont pas seulement les références et les types comme `S` ceux-là qui ont des durées de vie. Chaque type dans Rust a une durée de vie, y compris `i32` et `String`. La plupart sont simplement `'static`, ce qui signifie que les valeurs de ces types peuvent vivre aussi longtemps que vous le souhaitez ; par exemple, a `Vec<i32>` est autonome et n'a pas besoin d'être supprimé avant qu'une variable particulière ne sorte de la portée. Mais un type comme `Vec<&'a i32>` a une durée de vie qui doit être entourée par `'a` : il doit être supprimé tant que ses référents sont encore vivants.

Paramètres de durée de vie distincts

Supposons que vous ayez défini une structure contenant deux références comme celle-ci :

```
struct S<'a> {
    x: &'a i32,
    y:&'a i32
}
```

Les deux références utilisent la même durée de vie `'a`. Cela pourrait être un problème si votre code veut faire quelque chose comme ceci :

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y:&y };
        r = s.x;
```

```
    }
}
println!("{}", r);
```

Ce code ne crée aucun pointeur pendant. La référence à `y` reste dans `s`, qui sort de la portée avant `y`. La référence à `x` se termine par `r`, qui ne survit `x` pas à .

Si vous essayez de compiler cela, cependant, Rust se plaindra de `y` ne pas vivre assez longtemps, même si c'est clairement le cas. Pourquoi Rust est-il inquiet ? Si vous parcourez attentivement le code, vous pouvez suivre son raisonnement :

- Les deux champs de `S` sont des références avec la même durée de vie `'a`, donc Rust doit trouver une seule durée de vie qui fonctionne pour `s.x` et `s.y`.
- Nous attribuons `r = s.x`, nécessitant `'a` de joindre `r` la durée de vie de .
- Nous avons initialisé `s.y` avec `&y`, nécessitant `'a` de ne pas dépasser la durée de `y` vie de .

Ces contraintes sont impossibles à satisfaire : aucune durée de vie n'est inférieure à `y` la portée de mais supérieure à celle `r` de . Boucliers de rouille.

Le problème survient parce que les deux références `S` ont la même durée de vie `'a`. Changer la définition de `S` pour que chaque référence ait une durée de vie distincte corrige tout :

```
struct S<'a, 'b> {
    x: &'a i32,
    y:&'b i32
}
```

Avec cette définition, `s.x` et `s.y` ont des durées de vie indépendantes. Ce que nous faisons `s.x` n'a aucun effet sur ce que nous stockons dans `s.y`, il est donc facile de satisfaire les contraintes maintenant : `'a` peut simplement être `r` la durée de vie de , et `'b` peut être celle `s` de . (`y` La durée de vie de fonctionnerait aussi pour `'b`, mais Rust essaie de choisir la plus petite durée de vie qui fonctionne.) Tout finit bien.

Les signatures de fonction peuvent avoir des effets similaires. Supposons que nous ayons une fonction comme celle-ci :

```
fn f<'a>(r: &'a i32, s: &'a i32) ->&'a i32 { r } // perhaps too tight
```

Ici, les deux paramètres de référence utilisent la même durée de vie `'a`, ce qui peut inutilement contraindre l'appelant de la même manière que nous l'avons montré précédemment. Si cela pose problème, vous pouvez laisser les durées de vie des paramètres varier indépendamment :

```
fn f<'a, 'b>(r: &'a i32, s: &'b i32) ->&'a i32 { r } // looser
```

L'inconvénient est que l'ajout de durées de vie peut rendre les types et les signatures de fonction plus difficiles à lire. Vos auteurs ont tendance à essayer d'abord la définition la plus simple possible, puis à assouplir les restrictions jusqu'à ce que le code soit compilé. Étant donné que Rust ne permettra pas au code de s'exécuter à moins qu'il ne soit sûr, attendre simplement d'être informé lorsqu'il y a un problème est une tactique parfaitement acceptable.

Omission des paramètres de durée de vie

Jusqu'à présent, nous avons montré de nombreuses fonctions dans ce livre qui renvoient des références ou les prennent comme paramètres, mais nous n'avons généralement pas eu besoin de préciser quelle durée de vie est laquelle. Les durées de vie sont là; Rust nous laisse simplement les omettre quand il est raisonnablement évident de savoir ce qu'ils devraient être.

Dans les cas les plus simples, vous n'aurez peut-être jamais besoin d'écrire des durées de vie pour vos paramètres. Rust attribue simplement une durée de vie distincte à chaque endroit qui en a besoin. Par exemple:

```
struct S<'a, 'b> {
    x: &'a i32,
    y:&'b i32
}

fn sum_r_xy(r: &i32, s: S) ->i32 {
    r + s.x + s.y
}
```

La signature de cette fonction est un raccourci pour :

```
fn sum_r_xy<'a, 'b, 'c>(r: &'a i32, s: S<'b, 'c>) ->i32
```

Si vous renvoyez des références ou d'autres types avec des paramètres de durée de vie, Rust essaie toujours de faciliter les cas non ambigus. S'il n'y a qu'une seule durée de vie qui apparaît parmi les paramètres de votre fonction, alors Rust suppose que toutes les durées de vie dans votre valeur de retour doivent être celle-là :

```
fn first_third(point: &[i32; 3]) ->(&i32, &i32) {
    (&point[0], &point[2])
}
```

Avec toutes les durées de vie écrites, l'équivalent serait :

```
fn first_third<'a>(point: &'a [i32; 3]) ->(&'a i32, &'a i32)
```

S'il y a plusieurs durées de vie parmi vos paramètres, il n'y a aucune raison naturelle de préférer l'un à l'autre pour la valeur de retour, et Rust vous oblige à préciser ce qui se passe.

Si votre fonction est une méthode sur un type et prend son `self` paramètre par référence, cela rompt le lien : Rust suppose que `self` la durée de vie est celle qui donne tout dans votre valeur de retour. (Un `self` paramètre fait référence à la valeur sur laquelle la méthode est appelée, l'équivalent de Rust `this` en C++, Java ou JavaScript, ou `self` en Python. Nous couvrirons les méthodes dans "Définir des méthodes avec impl" .)

Par exemple, vous pouvez écrire ce qui suit :

```
struct StringTable {
    elements:Vec<String>,
}

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) ->Option<&String> {
        for i in 0 .. self.elements.len() {
            if self.elements[i].starts_with(prefix) {
                return Some(&self.elements[i]);
            }
        }
        None
    }
}
```

La `find_by_prefix` signature de la méthode est un raccourci pour :

```
fn find_by_prefix<'a, 'b>(&'a self, prefix: &'b str) ->Option<&'a String>
```

Rust suppose que tout ce que vous empruntez, vous empruntez à `self` .

Encore une fois, ce ne sont que des abréviations, destinées à être utiles sans introduire de surprises. Quand ils ne sont pas ce que vous voulez, vous pouvez toujours écrire explicitement les durées de vie.

Partage contre mutation

Jusqu'à présent, nous avons discutécomment Rust garantit qu'aucune référence ne pointera jamais vers une variable qui est sortie de la portée. Mais il existe d'autres façons d'introduire des pointeurs pendants. Voici un cas facile :

```
let v = vec![4, 8, 19, 27, 34, 10];
let r = &v;
let aside = v;  // move vector to aside
r[0];           // bad: uses `v`, which is now uninitialized
```

L'affectation à `aside` déplace le vecteur, le laissant `v` non initialisé, et se transforme `r` en un pointeur pendant, comme illustré à la Figure 5-7 .

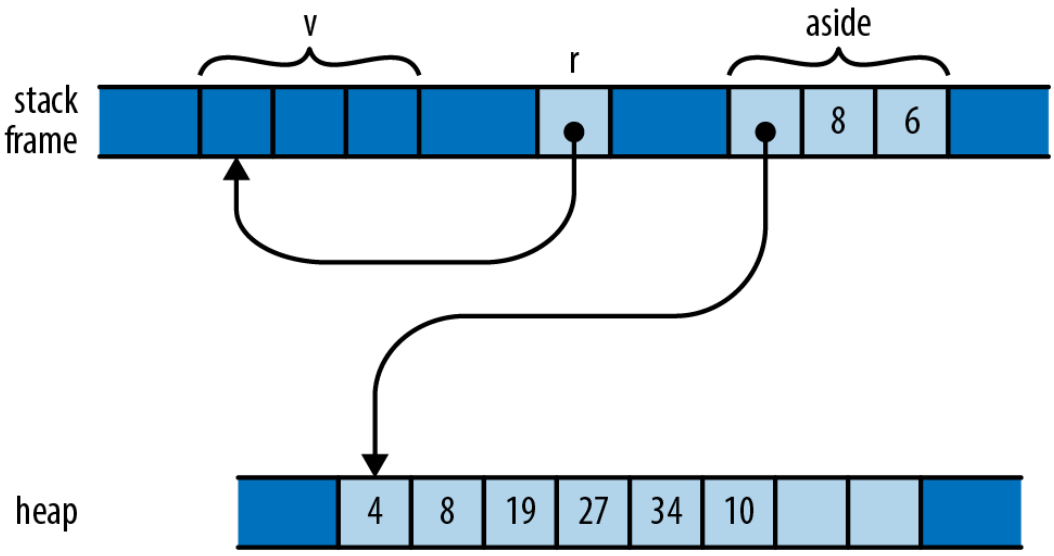

Figure 5-7. Une référence à un vecteur qui a été éloigné

Bien qu'il `v` reste dans la portée pendant `r` toute la durée de vie de , le problème ici est que `v` la valeur de est déplacée ailleurs, laissant `v` non initialisée tout `r` en s'y référant. Naturellement, Rust détecte l'erreur :

```
error: cannot move out of `v` because it is borrowed
   |
9  |     let r = &v;
   |             - borrow of `v` occurs here
10 |     let aside = v;  // move vector to aside
   |         ^^^^^ move out of `v` occurs here
```

Pendant toute sa durée de vie, une référence partagée rend son référent en lecture seule : vous ne pouvez pas l'affecter au référent ou déplacer sa valeur ailleurs. Dans ce code, `r` la durée de vie de contient la tentative de déplacement du vecteur, donc Rust rejette le programme. Si vous modifiez le programme comme indiqué ici, il n'y a pas de problème :

```
let v = vec![4, 8, 19, 27, 34, 10];
{
    let r = &v;
    r[0];       // ok: vector is still there
}
let aside = v;
```

Dans cette version, `r` sort plus tôt du champ d'application, la durée de vie de la référence se termine avant qu'elle ne `v` soit mise de côté, et tout va bien.

Voici une autre façon de faire des ravages. Supposons que nous ayons une fonction pratique pour étendre un vecteur avec les éléments d'une tranche :

```
fn extend(vec: &mut Vec<f64>, slice:&[f64]) {
    for elt in slice {
        vec.push(*elt);
    }
}
```

`extend_from_slice` Il s'agit d'une version moins flexible (et beaucoup moins optimisée) de la méthode de la bibliothèque standardsur les vecteurs. Nous pouvons l'utiliser pour construire un vecteur à partir de tranches d'autres vecteurs ou tableaux :

```
let mut wave = Vec::new();
let head = vec![0.0, 1.0];
let tail = [0.0, -1.0];
```

```
    extend(&mut wave, &head);    // extend wave with another vector
    extend(&mut wave, &tail);    // extend wave with an array

    assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0]);
```

Nous avons donc construit une période d'une onde sinusoïdale ici. Si nous voulons ajouter une autre ondulation, pouvons-nous ajouter le vecteur à lui-même ?

```
    extend(&mut wave, &wave);
    assert_eq!(wave, vec![0.0, 1.0, 0.0, -1.0,
                          0.0, 1.0, 0.0, -1.0]);
```

Cela peut sembler correct lors d'une inspection occasionnelle. Mais rappelez-vous que lorsque nous ajoutons un élément à un vecteur, si son tampon est plein, il doit allouer un nouveau tampon avec plus d'espace. Supposons `wave` qu'il commence par un espace pour quatre éléments et qu'il doive donc allouer un tampon plus grandquand `extend` essaie d'ajouter un cinquième. La mémoire finit par ressembler à la figure 5-8 .

L' argument `extend` de la fonction `vec` emprunte `wave` (appartenant à l'appelant), qui s'est alloué un nouveau tampon avec de l'espace pour huit éléments. Mais `slice` continue de pointer vers l'ancien tampon à quatre éléments, qui a été abandonné.

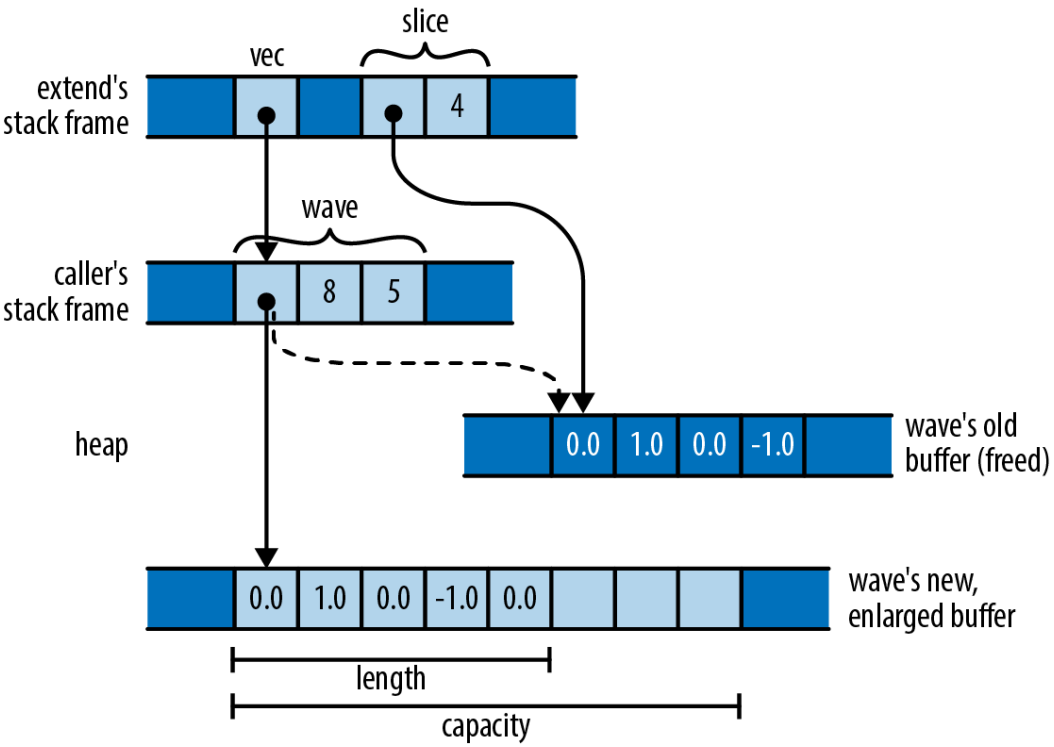

Figure 5-8. Une tranche transformée en un pointeur pendant par une réallocation de vecteur

Ce genre de problème n'est pas propre à Rust : modifier des collections tout en pointant dessus est un territoire délicat dans de nombreuses langues. En C++, la `std::vector` spécification vous avertit que "la réallocation [du tampon du vecteur] invalide toutes les références, pointeurs et itérateurs faisant référence aux éléments de la séquence". De même, dit Java, de modifierun `java.util.Hashtable` objet :

> *Si la table de hachage est structurellement modifiée à tout moment après la création de l'itérateur, de quelque manière que ce soit, sauf via la propre méthode remove de l'itérateur, l'itérateur lèvera un `ConcurrentModificationException`.*

Ce qui est particulièrement difficile avec ce genre de bogue, c'est qu'il n'arrive pas tout le temps. Lors des tests, votre vecteur peut toujours disposer de suffisamment d'espace, le tampon peut ne jamais être réalloué et le problème peut ne jamais apparaître.

Rust, cependant, signale le problème avec notre appel à `extend` au moment de la compilation :

```
error: cannot borrow `wave` as immutable because it is also
       borrowed as mutable
  |
9 |     extend(&mut wave, &wave);
  |                ----   ^^^^- mutable borrow ends here
  |                |      |
  |                |      immutable borrow occurs here
  |                mutable borrow occurs here
```

En d'autres termes, nous pouvons emprunter une référence mutable au vecteur, et nous pouvons emprunter une référence partagée à ses éléments, mais les durées de vie de ces deux références ne doivent pas se chevaucher. Dans notre cas, les durées de vie des deux références contiennent l'appel à `extend` , donc Rust rejette le code.

Cesles erreurs proviennent toutes deux de violations des règles de Rust pour la mutation et le partage :

> *L'accès partagé est un accès en lecture seule.*
>
> Valeursempruntés par des références partagées sont en lecture seule. Tout au long de la durée de vie d'une référence partagée, ni son référent, ni quoi que ce soit d'accessible à partir de ce référent, ne peut être modifié *par quoi que ce soit* . Il n'existe aucune référence mutable en direct à quoi que ce soit dans cette

structure, son propriétaire est en lecture seule, et ainsi de suite. C'est vraiment gelé.

L'accès modifiable est un accès exclusif.

Une valeur empruntée par une référence mutable est accessible exclusivement via cette référence. Au cours de la durée de vie d'une référence mutable, il n'y a pas d'autre chemin utilisable vers son référent ou vers une valeur accessible à partir de là. Les seules références dont les durées de vie peuvent chevaucher une référence mutable sont celles que vous empruntez à la référence mutable elle-même.

Rust a signalé l' `extend` exemple comme une violation de la deuxième règle : puisque nous avons emprunté une référence mutable à `wave` , cette référence mutable doit être le seul moyen d'atteindre le vecteur ou ses éléments. La référence partagée à la tranche est elle-même un autre moyen d'atteindre les éléments, en violation de la deuxième règle.

Mais Rust aurait également pu traiter notre bogue comme une violation de la première règle : puisque nous avons emprunté une référence partagée aux `wave` éléments de , les éléments et le `Vec` lui-même sont tous en lecture seule. Vous ne pouvez pas emprunter une référence mutable à une valeur en lecture seule.

Chaque type de référence affecte ce que nous pouvons faire avec les valeurs le long du chemin propriétaire vers le référent et les valeurs accessibles à partir du référent (Figure 5-9).

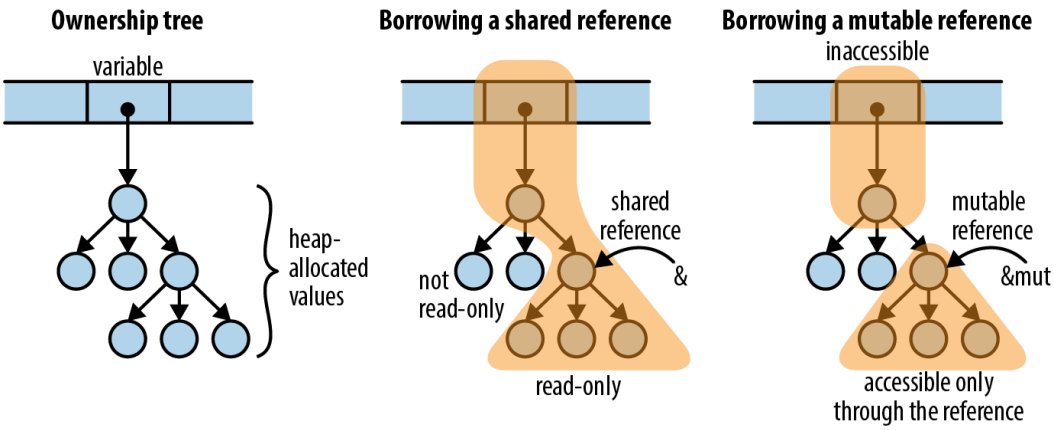

Figure 5-9. L'emprunt d'une référence affecte ce que vous pouvez faire avec d'autres valeurs dans le même arbre de propriété

Notez que dans les deux cas, le chemin de propriété menant au référent ne peut pas être modifié pendant la durée de vie de la référence. Pour un emprunt partagé, le chemin est en lecture seule ; pour un emprunt mutable, c'est complètement inaccessible. Il n'y a donc aucun moyen pour le programme de faire quoi que ce soit qui invalide la référence.

Réduisant ces principes aux exemples les plus simples possibles :

```
let mut x = 10;
let r1 = &x;
let r2 = &x;     // ok: multiple shared borrows permitted
x += 10;         // error: cannot assign to `x` because it is borrowed
let m = &mut x;  // error: cannot borrow `x` as mutable because it is
                 // also borrowed as immutable
println!("{}, {}, {}", r1, r2, m); // the references are used here,
                                   // so their lifetimes must last
                                   // at least this long

let mut y = 20;
let m1 = &mut y;
let m2 = &mut y;  // error: cannot borrow as mutable more than once
let z = y;        // error: cannot use `y` because it was mutably borrowe
println!("{}, {}, {}", m1, m2, z); // references are used here
```

Il est acceptable de réemprunter une référence partagée à partir d'une référence partagée :

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;          // ok: reborrowing shared as shared
let m1 = &mut r.1;      // error: can't reborrow shared as mutable
println!("{}", r0);     // r0 gets used here
```

Vous pouvez réemprunter à partir d'une référence mutable :

```
let mut v = (136, 139);
let m = &mut v;
let m0 = &mut m.0;      // ok: reborrowing mutable from mutable
*m0 = 137;
let r1 = &m.1;          // ok: reborrowing shared from mutable,
                        // and doesn't overlap with m0
v.1;                    // error: access through other paths still forbid
println!("{}", r1);     // r1 gets used here
```

Ces restrictions sont assez strictes. Pour en revenir à notre tentative d'appel `extend(&mut wave, &wave)`, il n'y a pas de moyen rapide et facile de corriger le code pour qu'il fonctionne comme nous le souhaitons. Et Rust applique ces règles partout : si nous empruntons, disons, une référence partagée à une clé dans a `HashMap`, nous ne pouvons pas emprunter une référence mutable à the `HashMap` tant que la durée de vie de la référence partagée n'est pas terminée.

Mais il y a une bonne justification à cela : concevoir des collections pour prendre en charge une itération et une modification simultanées et illimitées est difficile et empêche souvent des implémentations plus simples et plus efficaces. Java `Hashtable` et C++ `vector` ne dérangent pas, et ni les dictionnaires Python ni les objets JavaScript ne définissent exactement comment se comporte un tel accès. D'autres types de collections en JavaScript le font, mais nécessitent par conséquent des implémentations plus lourdes. La promesse de C++ `std::map` que l'insertion de nouvelles entrées n'invalide pas les pointeurs vers d'autres entrées de la carte, mais en faisant cette promesse, la norme empêche les conceptions plus efficaces en termes de cache comme celle de Rust `BTreeMap`, qui stocke plusieurs entrées dans chaque nœud de l'arborescence.

Voici un autre exemple du type de bogue que ces règles attrapent. Considérez le code C++ suivant, destiné à gérer un descripteur de fichier. Pour simplifier les choses, nous allons seulement montrer un constructeur et un opérateur d'affectation de copie, et nous allons omettre la gestion des erreurs :

```
struct File {
  int descriptor;

  File(int d) : descriptor(d) { }

  File& operator=(const File &rhs) {
    close(descriptor);
    descriptor = dup(rhs.descriptor);
    return *this;
  }
};
```

L'opérateur d'affectation est assez simple, mais échoue mal dans une situation comme celle-ci :

```
File f(open("foo.txt", ...));
...
f = f;
```

Si nous attribuons a `File` à lui-même, les deux `rhs` et `*this` sont le même objet, alors `operator=` ferme le descripteur de fichier auquel il est sur le point de passer `dup`. Nous détruisons la même ressource que nous étions censés copier.

Dans Rust, le code analogue serait :

```
struct File {
    descriptor:i32
}

fn new_file(d: i32) -> File {
    File { descriptor:d }
}

fn clone_from(this: &mut File, rhs:&File) {
    close(this.descriptor);
    this.descriptor = dup(rhs.descriptor);
}
```

(Ce n'est pas Rust idiomatique. Il existe d'excellents moyens de donner aux types Rust leurs propres fonctions et méthodes de constructeur, que nous décrivons au chapitre 9 , mais les définitions précédentes fonctionnent pour cet exemple.)

Si on écrit le code Rust correspondant à l'utilisation de `File` , on obtient :

```
let mut f = new_file(open("foo.txt", ...));
...
clone_from(&mut f, &f);
```

Rust, bien sûr, refuse même de compiler ce code :

```
error: cannot borrow `f` as immutable because it is also
       borrowed as mutable
   |
18 |     clone_from(&mut f, &f);
   |                    -   ^- mutable borrow ends here
   |                    |   |
   |                    |   immutable borrow occurs here
   |                    mutable borrow occurs here
```

Cela devrait vous sembler familier. Il s'avère que deux bogues C++ classiques - l'incapacité à gérer l'auto-assignation et l'utilisation d'itérateurs invalidés - sont le même type de bogue sous-jacent ! Dans les deux cas, le code suppose qu'il modifie une valeur tout en en consultant une autre, alors qu'en fait, il s'agit de la même valeur. Si vous avez déjà accidentellement laissé la source et la destination d'un appel vers `memcpy` ou `strcpy` se chevaucher en C ou C++, c'est encore une autre forme que le

bogue peut prendre. En exigeant que l'accès mutable soit exclusif, Rust a repoussé une large classe d'erreurs quotidiennes.

L'immiscibilité des références partagées et mutables démontre vraiment sa valeur lors de l'écriture de code concurrent. Une course aux données n'est possible que lorsqu'une valeur est à la fois modifiable et partagée entre les threads, ce qui est exactement ce que les règles de référence de Rust éliminent. Un programme Rust concurrent qui évite `unsafe` le code est exempt de courses de données *par construction* . Nous aborderons cet aspect plus en détail lorsque nous parlerons de la concurrence au chapitre 19 , mais en résumé, la concurrence est beaucoup plus facile à utiliser dans Rust que dans la plupart des autres langages..

RÉFÉRENCES PARTAGÉES DE RUST PAR RAPPORT AUX POINTEURS DE C VERS CONST

Au premier contrôle `const`, les références partagées de Rust semblent ressembler étroitement aux pointeurs vers les valeurs de C et C++. Cependant, les règles de Rust pour les références partagées sont beaucoup plus strictes. Par exemple, considérons le code C suivant :

```
int x = 42;             // int variable, not const
const int *p = &x;      // pointer to const int
assert(*p == 42);
x++;                    // change variable directly
assert(*p == 43);       // “constant” referent's value has changed
```

Le fait que `p` soit un `const int *` signifie que vous ne pouvez pas modifier son référent via `p` lui-même : `(*p)++` est interdit. Mais vous pouvez également accéder directement au référent en tant que `x`, qui n'est pas `const`, et modifier sa valeur de cette façon. Le mot-clé de la famille C `const` a ses utilisations, mais il ne l'est pas.

En Rust, une référence partagée interdit toute modification de son référent, jusqu'à la fin de sa durée de vie :

```
let mut x = 42;         // non-const i32 variable
let p = &x;             // shared reference to i32
assert_eq!(*p, 42);
x += 1;                 // error: cannot assign to x because it is borrow
assert_eq!(*p, 42);     // if you take out the assignment, this is true
```

Pour garantir qu'une valeur est constante, nous devons garder une trace de tous les chemins possibles vers cette valeur et nous assurer qu'ils ne permettent pas de modification ou qu'ils ne peuvent pas être utilisés du tout. Les pointeurs C et C++ sont trop illimités pour que le compilateur puisse vérifier cela. Les références de Rust sont toujours liées à une durée de vie particulière, ce qui permet de les vérifier au moment de la compilation.

Prendre les armes contre une mer d'objets

Depuis l'essor de la gestion automatique de la mémoire dans les années 1990, l'architecture par défaut de tous les programmes est la *mer d'objets*,

illustré à la Figure 5-10 .

C'est ce qui arrive si vous avez un ramasse-miettes et que vous commencez à écrire un programme sans rien concevoir. Nous avons tous construit des systèmes qui ressemblent à ceci.

Cette architecture présente de nombreux avantages qui n'apparaissent pas dans le schéma : les premiers progrès sont rapides, il est facile de pirater des éléments et, quelques années plus tard, vous n'aurez aucune difficulté à justifier une réécriture complète. (Cue "Highway to Hell" d'AC/DC.)

Bien sûr, il y a aussi des inconvénients. Quand tout dépend de tout le reste comme ça, il est difficile de tester, d'évoluer ou même de penser à un composant isolément.

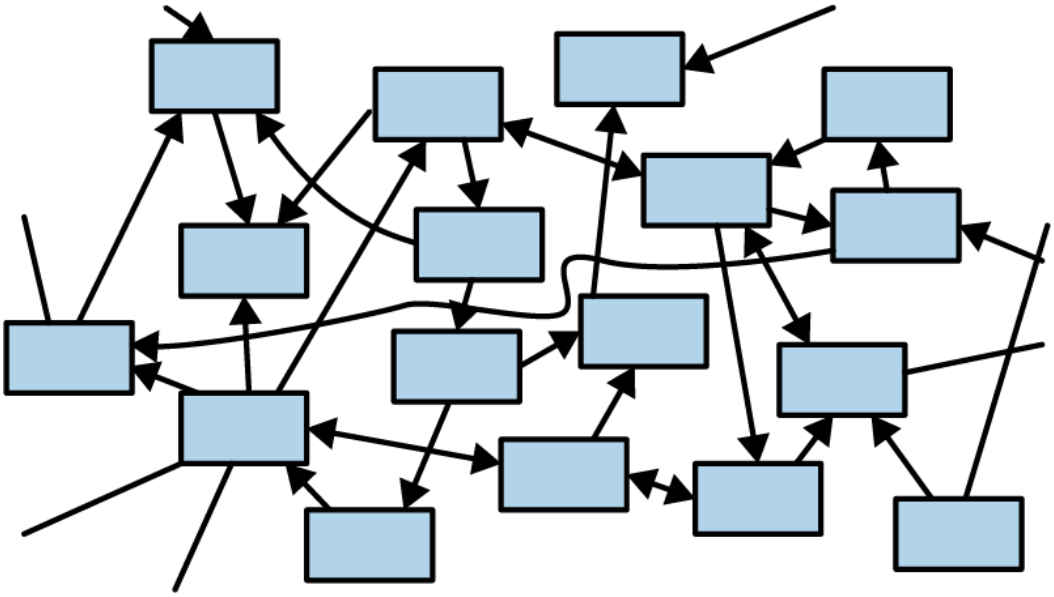

Figure 5-10. Une mer d'objets

Une chose fascinante à propos de Rust est que le modèle de propriété met un ralentisseur sur l'autoroute de l'enfer. Il faut un peu d'effort pour créer un cycle dans Rust - deux valeurs telles que chacune contient une référence pointant vers l'autre. Vous devez utiliser un type de pointeur intelligent, tel que `Rc` , et la mutabilité intérieure , un sujet que nous n'avons même pas encore abordé. Rust préfère que les pointeurs, la propriété et le flux de données traversent le système dans une seule direction, comme illustré à la Figure 5-11 .

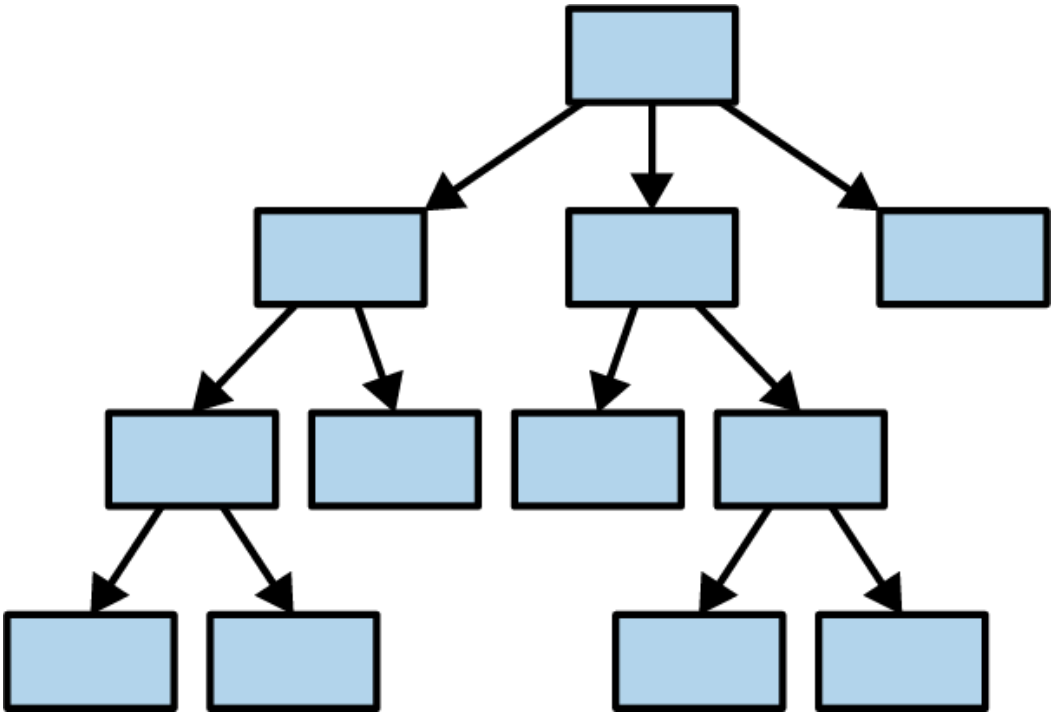

Figure 5-11. Un arbre de valeurs

La raison pour laquelle nous évoquons cela maintenant est qu'il serait naturel, après avoir lu ce chapitre, de vouloir se précipiter et de créer une "mer de structures", toutes liées avec `Rc` des pointeurs intelligents, et de recréer tous les objets- les anti-modèles orientés que vous connaissez. Cela ne fonctionnera pas pour vous tout de suite. Le modèle de propriété de Rust vous causera quelques problèmes. Le remède est de faire une conception initiale et de créer un meilleur programme.

Rust consiste à transférer la difficulté de comprendre votre programme du futur au présent. Cela fonctionne déraisonnablement bien : non seulement Rust peut vous forcer à comprendre pourquoi votre programme est thread-safe, mais il peut même nécessiter une certaine quantité d'architecture de haut niveau.motif.