Chapitre 5. Références

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

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

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

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

Dans ce chapitre, nous allons passer en revue le fonctionnement des références dans Rust; montrer comment les références, les fonctions et les types définis par l'utilisateur intègrent tous les informations de durée de vie pour s'assurer qu'ils sont utilisés en toute sécurité ; et illustrer certaines catégories courantes de bogues que ces efforts empêchent, au moment de la compilation et sans pénalités de performance au moment de l'exécution.

Références aux valeurs

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

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

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

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

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

La construction et l'impression de la table sont simples :

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

    show(table);
}
```

Et tout fonctionne bien:

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

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

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

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

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

En fait, si nous examinons la définition de , la boucle externe prend possession de la table de hachage et la consomme entièrement; et la boucle

interne fait de même pour chacun des vecteurs. (Nous avons vu ce comportement plus tôt, dans l'exemple « liberté, égalité, fraternité ».) En raison de la sémantique de déplacement, nous avons complètement détruit toute la structure simplement en essayant de l'imprimer. Merci, Rust! `show for for`

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

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

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

La fonction d'impression de notre exemple n'a pas besoin de modifier le tableau, il suffit de lire son contenu. Ainsi, l'appelant devrait être en mesure de lui transmettre une référence partagée à la table, comme suit :

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

Les références sont des pointeurs non propriétaires, de sorte que la variable reste propriétaire de l'ensemble de la structure ; vient de l'emprunter un peu. Naturellement, nous devrons ajuster la définition de pour correspondre, mais vous devrez regarder de près pour voir la différence: `table show show`

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

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

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

La boucle interne est modifiée de la même manière. L'itération sur une référence partagée à un vecteur est définie pour produire des références partagées à ses éléments, de sorte qu'un . Aucun propriétaire ne change de mains dans cette fonction; c'est juste passer autour de références non propriétaires. `work &String`

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

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

Et nous devons l'adopter un:

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

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

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

Utilisation des références

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

Références Rust versus références C++

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

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

```
// C++ code!
int x = 10;
int &r = x;               // initialization creates reference implicitly
assert(r == 10);          // implicitly dereference r to see x's value
r = 20;                   // stores 20 in x, r itself still points to x
```

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

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

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

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

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

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

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

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

La macro utilisée dans la fonction s'étend au code qui utilise l'opérateur, de sorte qu'elle tire également parti de cette déréférence implicite. `println! show .`

L'opérateur peut également emprunter implicitement une référence à son opérande gauche, si nécessaire pour un appel de méthode. Par exem-

ple, la méthode de ' prend une référence mutable au vecteur, de sorte que ces deux appels sont équivalents : . `Vec sort`

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

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

Affectation de références

L'affectation d'une référence à une variable rend ce point de variable quelque chose de nouveau :

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

if b { r = &y; }

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

La référence pointe d'abord vers . Mais si c'est vrai, le code le pointe plutôt, comme illustré à la figure 5-1. `r x b y`

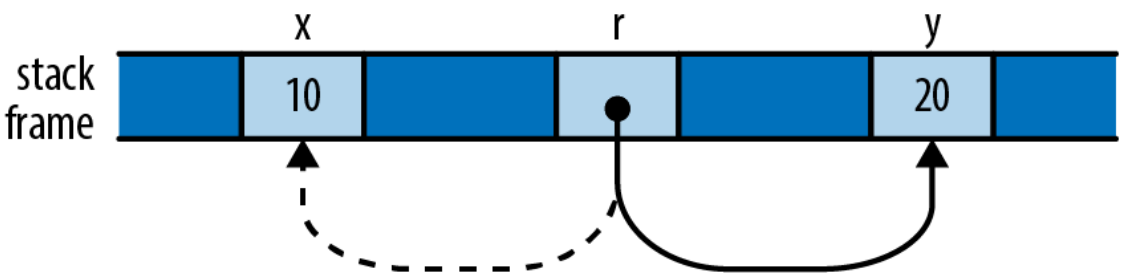

Figure 5-1. La référence , pointant maintenant vers au lieu de `r y x`

Ce comportement peut sembler trop évident pour mériter d'être mentionné: bien sûr, cela pointe maintenant vers , puisque nous y avons stocké. Mais nous le soulignons parce que les références C++ se comportent très différemment : comme indiqué précédemment, l'attribution d'une valeur à une référence en C++ stocke la valeur dans son référent. Une fois

qu'une référence C++ a été initialisée, il n'y a aucun moyen de la faire pointer vers autre chose. `r y &y`

Références aux références

La rouille autorise les références aux références :

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

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

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

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

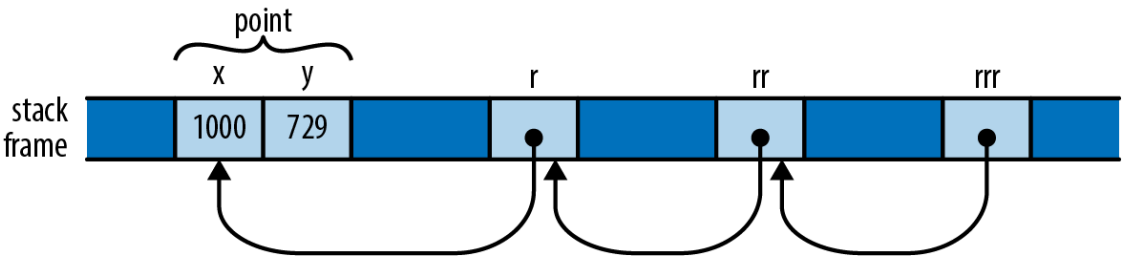

Figure 5-2. Une chaîne de références aux références

Ainsi, l'expression , guidée par le type de , traverse en fait trois références pour arriver à l'avant d'aller chercher son champ. `rrr.y rrr Point y`

Comparaison des références

Comme l'opérateur, les opérateurs de comparaison de Rust « voient à travers » n'importe quel nombre de références: `.`

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

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

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

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

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

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

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

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

Les références ne sont jamais nulles

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

Le code C et C++ utilise souvent un pointeur NULL pour indiquer l'absence de valeur : par exemple, la fonction renvoie soit un pointeur vers un nouveau bloc de mémoire, soit s'il n'y a pas assez de mémoire disponible pour satisfaire la demande. Dans Rust, si vous avez besoin d'une valeur qui est une référence à quelque chose ou non, utilisez le type . Au niveau de la machine, Rust représente comme un pointeur null et , où est une valeur, comme l'adresse non nulle, est donc tout aussi efficace qu'un pointeur nullable en C ou C ++, même s'il est plus sûr: son type vous oblige à vérifier si c'est avant de pouvoir

l'utiliser. `malloc nullptr Option<&T> None Some(r) r &T Option<& T> None`

Emprunt de références à des expressions arbitraires

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

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

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

- Si vous affectez immédiatement la référence à une variable dans une instruction (ou si vous l'intégrez à une structure ou à un tableau qui est immédiatement affecté), Rust fait vivre la variable anonyme aussi longtemps que la variable initialise. Dans l'exemple précédent, Rust le ferait pour le référent de . `let let r`
- Sinon, la variable anonyme vit jusqu'à la fin de l'instruction englobante. Dans notre exemple, la variable anonyme créée pour tenir ne dure que jusqu'à la fin de l'instruction. `1009 assert_eq!`

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

Références aux tranches et aux objets traits

Les références que nous avons montrées jusqu'à présent sont toutes des adresses simples. Cependant, Rust comprend également deux types de *pointeurs de graisse, des valeurs* de deux mots portant l'adresse d'une certaine valeur, ainsi que des informations supplémentaires nécessaires pour utiliser la valeur.

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

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

En plus de transporter ces données supplémentaires, les références d'objets de tranches et de traits se comportent comme les autres types de références que nous avons montrées jusqu'à présent dans ce chapitre : elles ne possèdent pas leurs référents, elles ne sont pas autorisées à survivre à leurs référents, elles peuvent être mutables ou partagées, etc.

Sécurité de référence

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

Pour transmettre les idées fondamentales, nous commencerons par les cas les plus simples, en montrant comment Rust s'assure que les références sont utilisées correctement dans un seul corps de fonction. Ensuite, nous examinerons le passage de références entre les fonctions et leur stockage dans des structures de données. Cela implique de donner à ces fonctions et types *de données des paramètres de durée de vie*, que nous allons expliquer. Enfin, nous présenterons quelques raccourcis fournis par Rust pour simplifier les modèles d'utilisation courants. Tout au long, nous montrerons comment Rust pointe le code cassé et suggère souvent des solutions.

Emprunt d'une variable locale

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

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

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

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

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

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

Rust essaie d'attribuer à chaque type de référence de votre programme une *durée de vie* qui répond aux contraintes imposées par son utilisation. Une durée de vie est un tronçon de votre programme pour lequel une référence pourrait être utilisée en toute sécurité : une instruction, une expression, la portée d'une variable ou autre. Les durées de vie sont entièrement le fruit de l'imagination de Rust au moment de la compilation. Au moment de l'exécution, une référence n'est rien d'autre qu'une adresse ;

sa durée de vie fait partie de son type et n'a pas de représentation au moment de l'exécution.

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

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

Au-delà du point où elle sort du champ d'application, la référence serait un pointeur pendant. Nous disons que la durée de vie de la variable doit *contenir* ou enfermer celle de la référence qui lui est *empruntée*. `x`

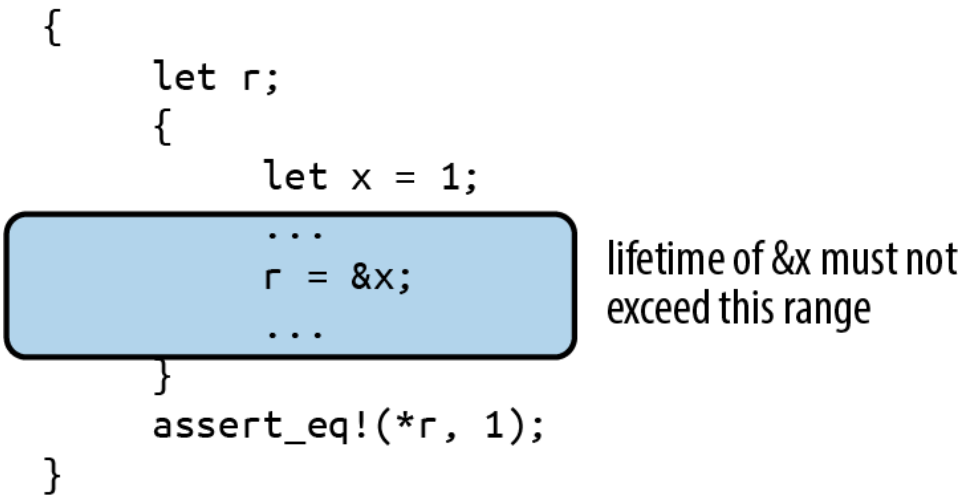

Figure 5-3. Durées de vie admissibles pour &x

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

Si la référence ne peut pas vivre au moins aussi longtemps que la variable, alors à un moment donné sera un pointeur pendant. Nous disons que la durée de vie de la référence doit contenir ou entourer celle de la variable. `r`

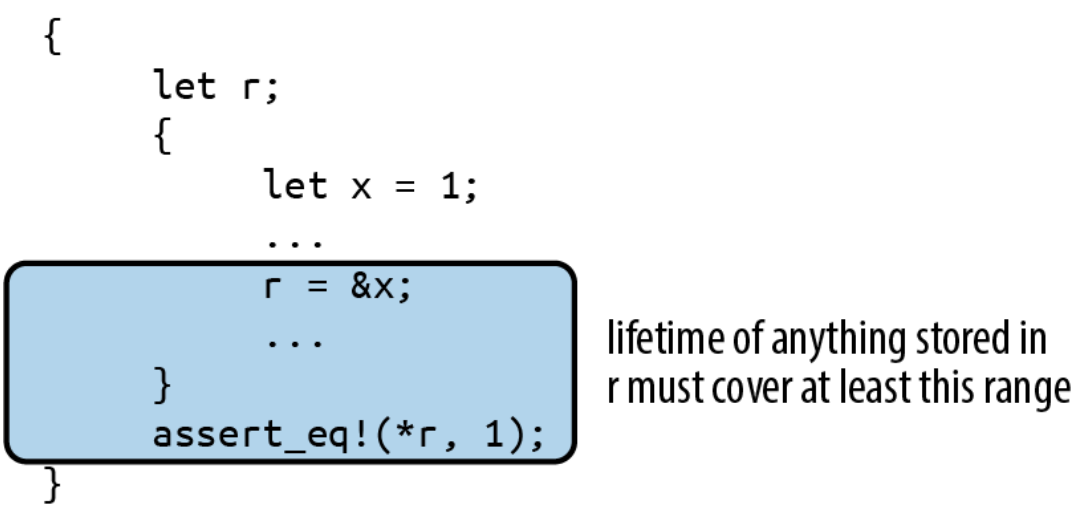

Figure 5-4. Durées de vie admissibles pour la référence stockée dans r

Le premier type de contrainte limite la taille de la durée de vie d'une référence, tandis que le second type limite sa petite taille. Rust essaie simplement de trouver une durée de vie pour chaque référence qui réponde à toutes ces contraintes. Dans notre exemple, cependant, il n'y a pas une telle durée de vie, comme le montre la figure 5-5.

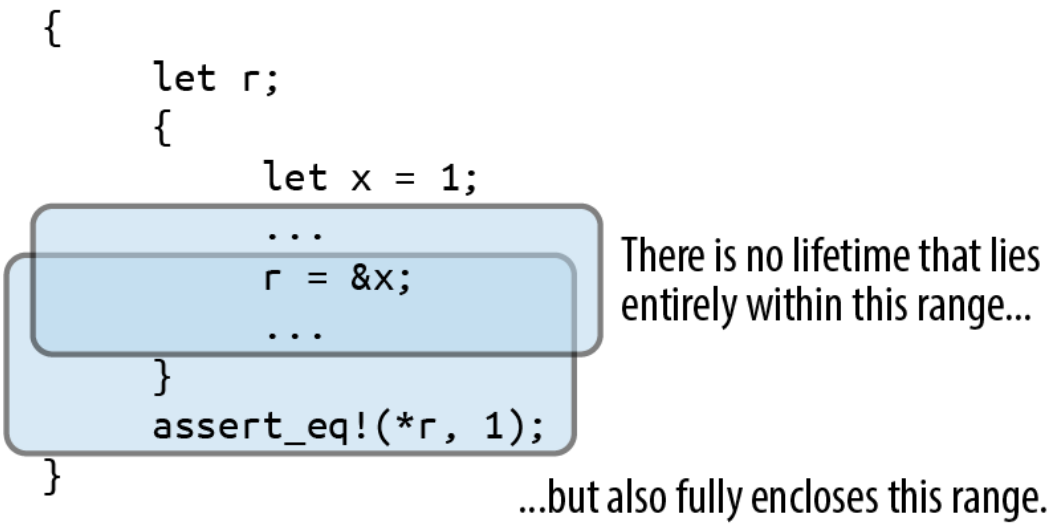

Figure 5-5. Une référence avec des contraintes contradictoires sur sa durée de vie

Considérons maintenant un exemple différent où les choses fonctionnent. Nous avons les mêmes types de contraintes : la durée de vie de la référence doit être contenue par 's, mais entièrement fermée 's. Mais parce que la durée de vie de est plus petite maintenant, il y a une durée de vie qui répond aux contraintes, comme le montre la figure 5-6. `x r r`

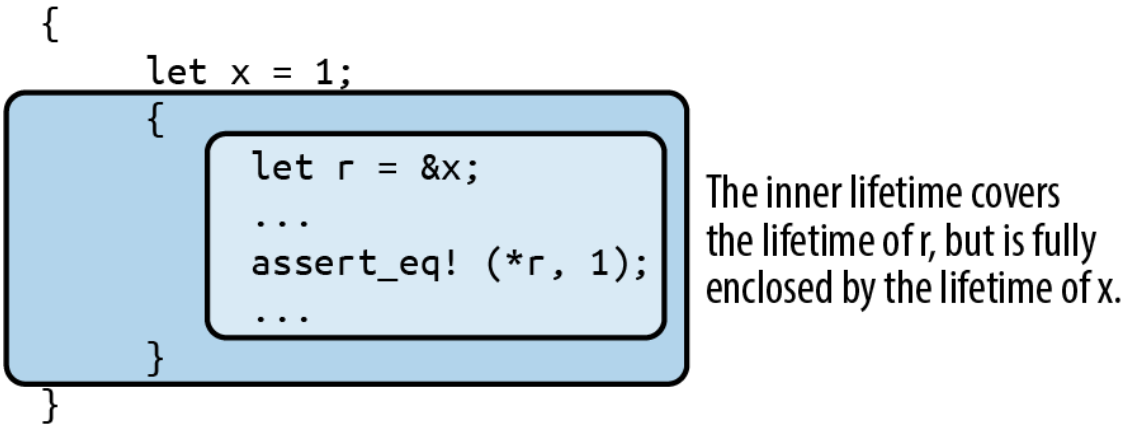

Figure 5-6. Une référence avec une durée de vie englobant la portée de ', mais dans la portée de " `r x`

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

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

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

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

Réception de références en tant qu'arguments de fonction

Lorsque nous passons une référence à une fonction, comment Rust s'assure-t-elle que la fonction l'utilise en toute sécurité ? Supposons que nous ayons une fonction qui prend une référence et la stocke dans une vari-

able globale. Nous devrons apporter quelques révisions à cela, mais voici une première coupe: `f`

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

L'équivalent de Rust d'une variable globale est appelé *statique* : c'est une valeur qui est créée au démarrage du programme et qui dure jusqu'à sa fin. (Comme toute autre déclaration, le système de modules de Rust contrôle où les statiques sont visibles, de sorte qu'elles ne sont que « globales » au cours de leur vie, pas leur visibilité.) Nous couvrons la statique dans le chapitre 8, mais pour l'instant, nous allons simplement appeler quelques règles que le code qui vient d'être affiché ne suit pas:

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

Une fois ces révisions effectuées, nous avons maintenant ce qui suit :

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

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

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

Ici, la durée de vie (prononcée « tick A ») est un *paramètre de durée de vie* de . Vous pouvez lire comme « pour n'importe quelle vie » donc quand

nous écrivons , nous définissons une fonction qui prend une référence à une avec une durée de vie donnée . `'a f <'a> 'a fn f<'a>(p: &'a i32) i32 'a`

Puisque nous devons permettre d'être n'importe quelle vie, les choses feraient mieux de fonctionner si c'est la plus petite vie possible: une seule en enfermant simplement l'appel à . Cette mission devient alors un point de discorde : `'a f`

```
STASH = p;
```

Puisque vit pour l'ensemble de l'exécution du programme, le type de référence qu'il détient doit avoir une durée de vie de la même durée; Rust appelle cela la *« durée de vie statique »*. Mais la durée de vie de la référence de est certaine, ce qui pourrait être n'importe quoi, tant qu'elle enferme l'appel à . Ainsi, Rust rejette notre code: `STASH p 'a f`

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

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

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

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

Cette fois, la signature de 's indique qu'il doit s'agir d'une référence avec une durée de vie , il n'y a donc plus de problème à la stocker dans . Nous ne pouvons appliquer qu'aux références à d'autres statiques, mais c'est la seule chose qui est certaine de ne pas laisser pendre de toute façon. Nous pouvons donc écrire : `f p 'static STASH f STASH`

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

Puisque est un statique, le type de est , qui est sûr de passer à `. WORTH_POINTING_AT &WORTH_POINTING_AT &'static i32 f`

Prenez un peu de recul, cependant, et remarquez ce qui est arrivé à la signature de ', alors que nous modifiions notre façon de corriger: l'original a fini par être . En d'autres termes, nous n'avons pas pu écrire une fonction qui cachait une référence dans une variable globale sans refléter cette intention dans la signature de la fonction. Dans Rust, la signature d'une fonction expose toujours le comportement du corps. `f f(p: &i32) f(p: &'static i32)`

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

Transmission de références à des fonctions

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

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

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

Rien que par sa signature, Rust sait qu'il n'économisera aucun endroit qui pourrait survivre à l'appel: toute durée de vie qui entoure l'appel doit fonctionner pendant . Rust choisit donc la plus petite durée de vie possible pour : celle de l'appel à . Cela répond à toutes les contraintes : il ne survit pas, et il enferme l'ensemble de l'appel à . Donc, ce code passe le rassemblement. `g p 'a &x g x g`

Notez que bien que prend un paramètre de durée de vie , nous n'avons pas eu besoin de le mentionner lors de l'appel . Vous n'avez qu'à vous soucier des paramètres de durée de vie lors de la définition des fonctions et des types; lors de leur utilisation, Rust déduit les durées de vie pour vous. `g 'a g`

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

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

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

Cela ne se compile pas : la référence ne doit pas survivre, mais en la transmettant à , on la contraint à vivre au moins aussi longtemps que . Il n'y a aucun moyen de satisfaire tout le monde ici, alors Rust rejette le code. `&x x f 'static`

Renvoi de références

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

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

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

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

Supposons que nous appelions comme ceci: `smallest`

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

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

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

Se déplacer pour que sa durée de vie soit clairement contenue dans 's résout le problème: `s parabola`

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

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

Structures contenant des références

Comment Rust gère-t-il les références stockées dans les structures de données ? Voici le même programme erroné que nous avons examiné précédemment, sauf que nous avons mis la référence à l'intérieur d'une structure:

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

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

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

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

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

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

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

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

```
}
```

Maintenant, le type a une durée de vie, tout comme les types de référence. Chaque valeur que vous créez de type obtient une nouvelle durée de vie, qui devient limitée par la façon dont vous utilisez la valeur. La durée de vie de toute référence dans laquelle vous stockez devrait être incluse, et doit durer plus longtemps que la durée de vie de l'endroit où vous stockez le . `S S 'a r 'a 'a S`

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

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

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

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

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

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

Nous ne pouvons pas laisser de côté le paramètre de durée de vie de Rust ici: Rust a besoin de savoir comment la durée de vie de la référence dans son afin d'appliquer les mêmes contrôles qu'il fait pour et les références simples. `S D S D S`

Nous pourrions donner la vie. Cela fonctionne : `s 'static`

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

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

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

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

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

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

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

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

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

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

Sans examiner du tout la définition du type, nous pouvons dire que, si nous recevons un de , toutes les références qu'il contient doivent pointer dans le tampon d'entrée que nous avons passé, et nulle part ailleurs (sauf peut-être à des valeurs). `Record Record parse_record 'static`

En fait, cette exposition du comportement interne est la raison pour laquelle Rust exige des types qui contiennent des références pour prendre des paramètres de durée de vie explicites. Il n'y a aucune raison pour que Rust ne puisse pas simplement inventer une durée de vie distincte pour chaque référence dans la structure et vous épargner la peine de les écrire. Les premières versions de Rust se comportaient en fait de cette façon, mais les développeurs ont trouvé cela déroutant: il est utile de savoir quand une valeur emprunte quelque chose à une autre valeur, en particulier lorsque vous travaillez sur des erreurs.

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

Paramètres de durée de vie distincts

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

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

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

```
let x = 10;
let r;
{
    let y = 20;
    {
        let s = S { x: &x, y: &y };
        r = s.x;
    }
}
println!("{}", r);
```

Ce code ne crée pas de pointeurs pendants. La référence à reste dans , qui sort du champ d'application avant. La référence à finit dans , qui ne survit pas à `x . y s y x r`

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

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

Ces contraintes sont impossibles à satisfaire : aucune durée de vie n'est plus courte que la portée de ' mais plus longue que celle de 's. Rust rechigne. `y r`

Le problème se pose car les deux références dans ont la même durée de vie. Changer la définition de pour permettre à chaque référence d'avoir une durée de vie distincte corrige tout: `S 'a S`

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

Avec cette définition, et avoir des durées de vie indépendantes. Ce que nous faisons avec n'a aucun effet sur ce que nous stockons dans , il est

donc facile de satisfaire les contraintes maintenant: peut simplement être la durée de vie de ', et peut être celle de '. (la durée de vie fonctionnerait aussi pour , mais Rust essaie de choisir la plus petite durée de vie qui fonctionne.) Tout finit bien. `s.x s.y s.x s.y 'a r 'b s y 'b`

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

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

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

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

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

Omission des paramètres de durée de vie

Nous avons montré beaucoup de fonctions jusqu'à présent dans ce livre qui renvoient des références ou les prennent comme paramètres, mais nous n'avons généralement pas besoin d'expliquer quelle durée de vie est laquelle. Les durées de vie sont là; Rust nous permet simplement de les omettre alors qu'il est raisonnablement évident ce qu'ils devraient être.

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

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

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

La signature de cette fonction est l'abréviation de :

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

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

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

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

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

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

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

Par exemple, vous pouvez écrire ce qui suit :

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

impl StringTable {
    fn find_by_prefix(&self, prefix: &str) -> Option<&String> {
```

```
            for i in 0 .. self.elements.len() {
                if self.elements[i].starts_with(prefix) {
                    return Some(&self.elements[i]);
                }
            }
            None
        }
    }
```

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

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

Rust suppose que quoi que vous empruntiez, vous empruntez à . `self`

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

Partage versus mutation

Jusqu'à présent, nous avons discuté de la façon dont Rust garantit qu'aucune référence ne pointera jamais vers une variable qui est sortie de son champ d'application. Mais il existe d'autres façons d'introduire des pointeurs pendants. Voici un cas simple:

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

Affectation pour déplacer le vecteur, en laissant non initialisé, et se transforme en pointeur pendant, comme illustré à la figure 5-7. `aside v r`

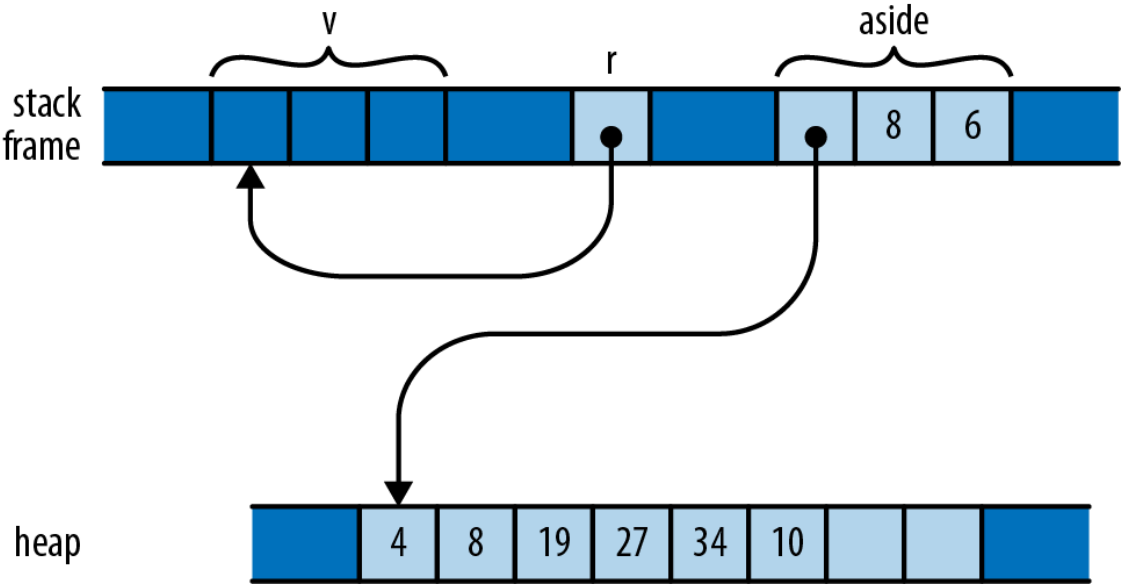

Figure 5-7. Référence à un vecteur qui a été déplacé

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

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

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

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

Dans cette version, sort du champ d'application plus tôt, la durée de vie de la référence se termine avant d'être déplacée, et tout va bien. `r v`

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

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

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

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

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

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

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

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

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

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

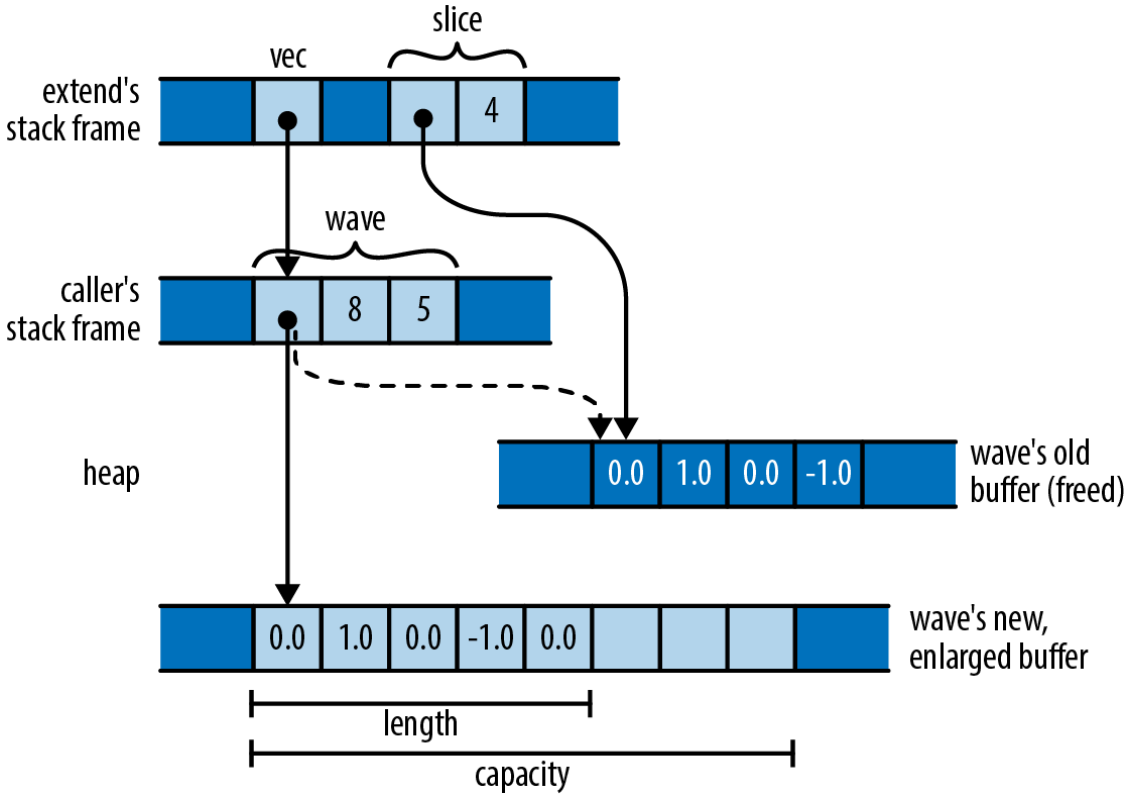

Figure 5-8. Une tranche transformée en pointeur pendant par une réaffectation vectorielle

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

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

Ce qui est particulièrement difficile à propos de ce genre de bug, c'est que cela n'arrive pas tout le temps. Lors des tests, votre vecteur peut toujours avoir suffisamment d'espace, le tampon peut ne jamais être réaffecté et le problème peut ne jamais apparaître.

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

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

```
|                    |       |
|                    |       immutable borrow occurs here
|                  mutable borrow occurs here
```

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

Ces erreurs proviennent toutes deux de violations des règles de Rust en matière de mutation et de partage :

L'accès partagé est un accès en lecture seule.

> Les valeurs empruntées par des références partagées sont en lecture seule. Tout au long de la durée de vie d'une référence partagée, ni son référent, ni rien d'accessible à partir de ce référent, ne peut être modifié *par quoi que ce soit*. Il n'existe aucune référence mutable en direct à quoi que ce soit dans cette structure, son propriétaire est tenu en lecture seule, et ainsi de suite. C'est vraiment gelé.

L'accès mutable est un accès exclusif.

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

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

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

Chaque type de référence affecte ce que nous pouvons faire avec les valeurs le long du chemin de propriété vers le référent, et les valeurs ac-

cessibles à partir du référent (Figure 5-9).

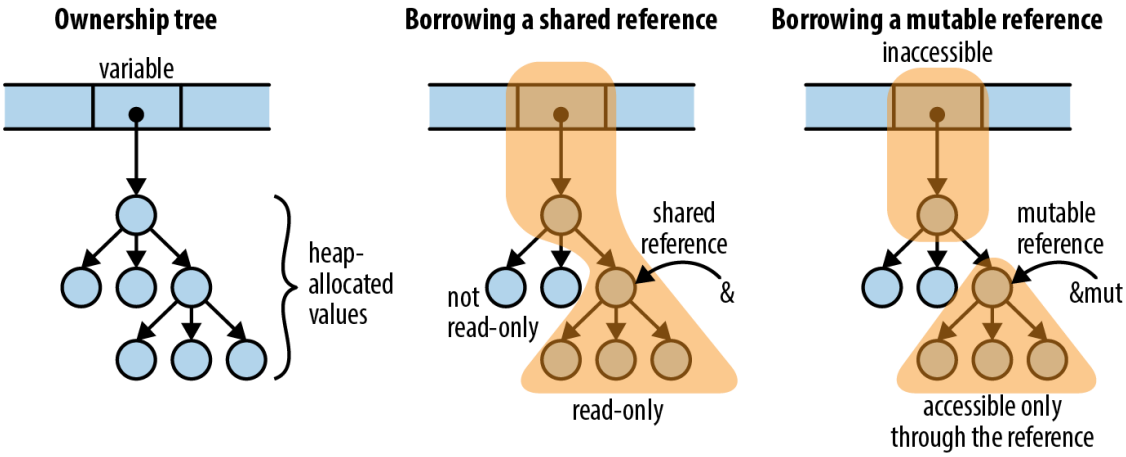

Figure 5-9. L'emprunt d'une référence affecte ce que vous pouvez faire avec d'autres valeurs dans le même arbre de propriété

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 d'accès est en lecture seule ; pour un emprunt mutable, c'est complètement inaccessible. Il n'y a donc aucun moyen pour le programme de faire quoi que ce soit qui invaliderait la référence.

En réduisant ces principes aux exemples les plus simples possibles :

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

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

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

```
let mut w = (107, 109);
let r = &w;
let r0 = &r.0;          // ok: reborrowing shared as shared
```

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

Vous pouvez réemprunter à partir d'une référence modifiable :

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

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

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

Voici un autre exemple du type de bug détecté par ces règles. Considérez le code C++ suivant, destiné à gérer un descripteur de fichier. Pour simplifier les choses, nous n'allons afficher qu'un constructeur et un opérateur d'affectation de copie, et nous allons omettre la gestion des erreurs :

```
struct File {
  int descriptor;
```

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

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

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

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

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

Dans Rust, le code analogue serait :

```
struct File {
    descriptor: i32
}

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

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

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

Si nous écrivons le code Rust correspondant à l'utilisation de , nous obtenons: `File`

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

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

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

Cela devrait vous sembler familier. Il s'avère que deux bogues C++ classiques – l'incapacité à faire face à l'auto-affectation et l'utilisation d'itérateurs invalidés – sont le même type de bogue sous-jacent ! Dans les deux cas, le code suppose qu'il modifie une valeur tout en en consultant une autre, alors qu'en fait, ils ont tous les deux la même valeur. Si vous avez accidentellement laissé la source et la destination d'un appel vers ou se chevaucher en C ou C ++, c'est encore une autre forme que le bogue peut prendre. En exigeant que l'accès mutable soit exclusif, Rust a repoussé une large classe d'erreurs quotidiennes. `memcpy strcpy`

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

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

À première vue, les références partagées de Rust semblent ressembler étroitement aux pointeurs de valeurs de C et C ++. Cependant, les règles de Rust pour les références partagées sont beaucoup plus strictes. Par exemple, considérez le code C suivant : `const`

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

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

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

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

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

Prendre les armes contre une mer d’objets

Depuis l’essor de la gestion automatique de la mémoire dans les années 1990, l’architecture par défaut de tous les programmes a été la *mer d’objets*, illustrée à la figure 5-10.

C’est ce qui se passe si vous avez un garbage collection et que vous commencez à écrire un programme sans rien concevoir. Nous avons tous construit des systèmes qui ressemblent à ceci.

Cette architecture présente de nombreux avantages qui n’apparaissent pas dans le diagramme : la progression initiale est rapide, il est facile de pirater des choses, et quelques années plus tard, vous n’aurez aucune difficulté à justifier une réécriture complète. (Cue AC/DC 'S « Highway to Hell. »)

Bien sûr, il y a aussi des inconvénients. Lorsque tout dépend de tout le reste comme celui-ci, il est difficile de tester, d’évoluer ou même de penser à un composant isolément.

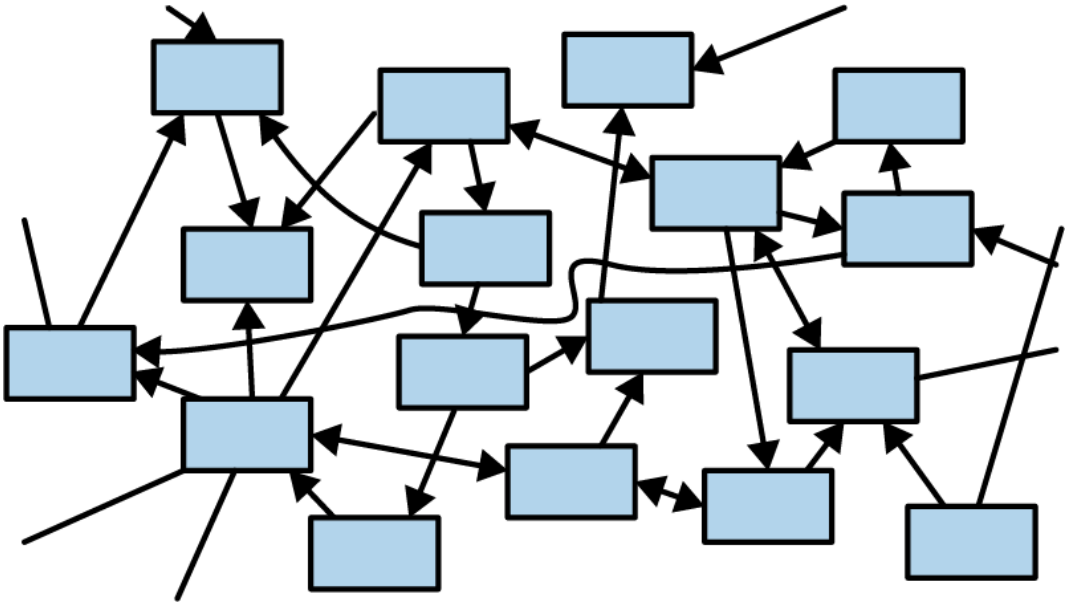

Figure 5-10. Une mer d’objets

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

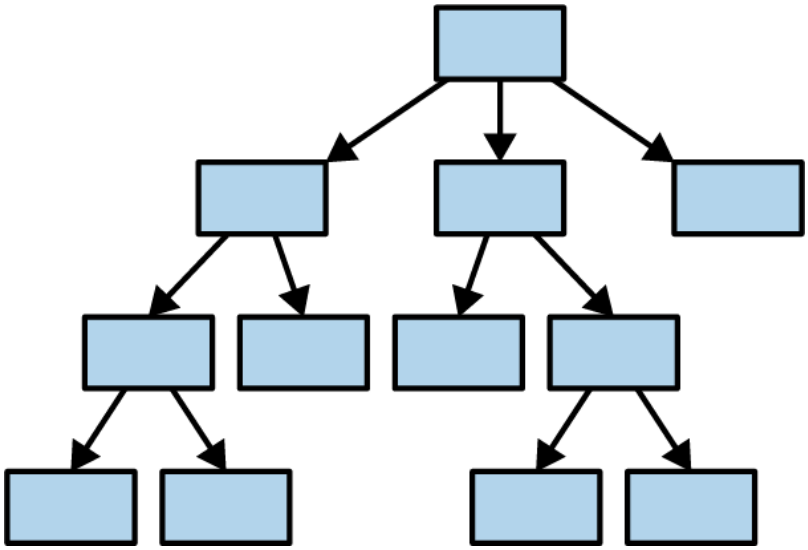

Figure 5-11. Un arbre de valeurs

La raison pour laquelle nous soulevons cette question en ce moment est qu'il serait naturel, après avoir lu ce chapitre, de vouloir courir tout de suite et de créer une « mer de structs », le tout lié avec des pointeurs intelligents, et de recréer tous les antimodèles orientés objet que vous connaissez. Cela ne fonctionnera pas pour vous tout de suite. Le modèle de propriété de Rust vous donnera quelques problèmes. Le remède consiste à faire une conception initiale et à élaborer un meilleur programme. Rc

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