Chapitre 4. Propriété et déménagements

En ce qui concerne la gestion de la mémoire, il y a deux caractéristiques que nous aimerions de nos langages de programmation :

- Nous aimerions que la mémoire soit libérée rapidement, au moment de notre choix. Cela nous donne le contrôle sur la consommation de mémoire du programme.
- Nous ne voulons jamais utiliser un pointeur vers un objet après sa libération. Il s'agirait d'un comportement indéfini, conduisant à des accidents et à des failles de sécurité.

Mais ceux-ci semblent s'exclure mutuellement : libérer une valeur alors que des pointeurs existent vers elle laisse nécessairement ces pointeurs pendants. Presque tous les principaux langages de programmation tombent dans l'un des deux camps, selon laquelle des deux qualités ils abandonnent:

- Le camp « La sécurité d'abord » utilise le ramassage des ordures pour gérer la mémoire, libérant automatiquement les objets lorsque tous les pointeurs accessibles vers eux ont disparu. Cela élimine les pointeurs pendants en gardant simplement les objets autour jusqu'à ce qu'il n'y ait plus de pointeurs vers eux à pendre. Presque tous les langages modernes tombent dans ce camp, de Python, JavaScript et Ruby à Java, C # et Haskell.
  Mais s'appuyer sur le ramassage des ordures signifie abandonner le contrôle sur le moment exact où les objets sont libérés au collecteur. En général, les éboueurs sont des bêtes surprenantes, et comprendre pourquoi la mémoire n'a pas été libérée alors que vous vous y attendiez peut être un défi.
- Le camp « Control First » vous laisse en charge de libérer la mémoire.
  La consommation de mémoire de votre programme est entièrement entre vos mains, mais éviter les pointeurs pendants devient également entièrement votre préoccupation. C et C++ sont les seuls langages courants dans ce camp.
  C'est génial si vous ne faites jamais d'erreurs, mais les preuves suggèrent que vous finirez par le faire. L'utilisation abusive du pointeur est un coupable courant des problèmes de sécurité signalés depuis que ces données ont été collectées.

Rust vise à être à la fois sûr et performant, donc aucun de ces compromis n'est acceptable. Mais si la réconciliation était facile, quelqu'un l'aurait fait bien avant maintenant. Quelque chose de fondamental doit changer.

Rust brise l'impasse d'une manière surprenante: en limitant la façon dont vos programmes peuvent utiliser les pointeurs. Ce chapitre et le suivant sont consacrés à expliquer exactement ce que sont ces restrictions et pourquoi elles fonctionnent. Pour l'instant, il suffit de dire que certaines structures courantes que vous avez l'habitude d'utiliser peuvent ne pas correspondre aux règles, et vous devrez chercher des alternatives. Mais l'effet net de ces restrictions est d'apporter juste assez d'ordre au chaos pour permettre aux vérifications au moment de la compilation de Rust de vérifier que votre programme est exempt d'erreurs de sécurité de la mémoire: pointeurs pendants, doubles libres, utilisation de mémoire non initialisée, etc. Au moment de l'exécution, vos pointeurs sont de simples adresses en mémoire, tout comme ils le seraient en C et C++. La différence est que votre code a été prouvé pour les utiliser en toute sécurité.

Ces mêmes règles constituent également la base de la prise en charge par Rust pour une programmation simultanée sécurisée. En utilisant les primitives de threading soigneusement conçues par Rust, les règles qui garantissent que votre code utilise correctement la mémoire servent également à prouver qu'il est exempt de courses de données. Un bogue dans un programme Rust ne peut pas amener un thread à corrompre les données d'un autre, introduisant des échecs difficiles à reproduire dans des parties non liées du système. Le comportement non déterministe inhérent au code multithread est isolé aux fonctionnalités conçues pour le gérer (mutex, canaux de message, valeurs atomiques, etc.) plutôt que d'apparaître dans des références de mémoire ordinaires. Le code multithread en C et C ++ a gagné sa réputation laide, mais Rust le réhabilite assez bien.

Le pari radical de Rust, la revendication sur laquelle il mise son succès et qui constitue la racine du langage, est que même avec ces restrictions en place, vous trouverez le langage plus que suffisamment flexible pour presque toutes les tâches et que les avantages – l'élimination de larges classes de bogues de gestion de la mémoire et de concurrence – justifieront les adaptations que vous devrez apporter à votre style. Les auteurs de ce livre sont optimistes sur Rust précisément en raison de notre vaste expérience avec C et C ++. Pour nous, l'accord de Rust est une évidence.

Les règles de Rust sont probablement différentes de ce que vous avez vu dans d'autres langages de programmation. Apprendre à travailler avec

eux et à les tourner à votre avantage est, à notre avis, le défi central de l'apprentissage de Rust. Dans ce chapitre, nous allons d'abord donner un aperçu de la logique et de l'intention derrière les règles de Rust en montrant comment les mêmes problèmes sous-jacents se déroulent dans d'autres langues. Ensuite, nous expliquerons en détail les règles de Rust, en examinant ce que signifie la propriété au niveau conceptuel et mécanique, comment les changements de propriété sont suivis dans divers scénarios et les types qui plient ou enfreignent certaines de ces règles afin d'offrir plus de flexibilité.

Propriété

Si vous avez lu beaucoup de code C ou C++, vous avez probablement rencontré un commentaire disant qu'une instance d'une classe *possède* un autre objet vers lequel elle pointe. Cela signifie généralement que l'objet propriétaire décide quand libérer l'objet possédé: lorsque le propriétaire est détruit, il détruit ses biens avec lui.

Par exemple, supposons que vous écriviez le code C++ suivant :

```
std::string s = "frayed knot";
```

La chaîne est généralement représentée en mémoire, comme illustré à la figure 4-1. `s`

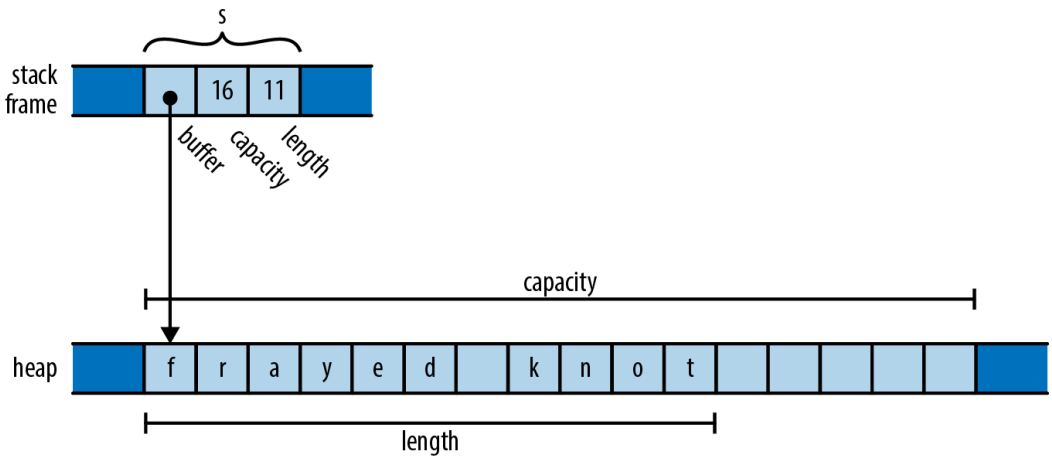

Figure 4-1. Valeur C++ sur la pile, pointant vers sa mémoire tampon allouée au tas `std::string`

Ici, l'objet lui-même est toujours exactement de trois mots, comprenant un pointeur vers un tampon alloué au tas, la capacité globale du tampon (c'est-à-dire la taille du texte peut croître avant que la chaîne ne doive allouer un tampon plus grand pour le contenir) et la longueur du texte qu'il contient maintenant. Il s'agit de champs privés à la classe, non accessibles aux utilisateurs de la chaîne. `std::string` `std::string`

A possède son tampon : lorsque le programme détruit la chaîne, le destructeur de la chaîne libère le tampon. Dans le passé, certaines bibliothèques C++ partageaient un seul tampon entre plusieurs valeurs, en utilisant un nombre de références pour décider quand le tampon devait être libéré. Les versions plus récentes de la spécification C++ excluent effectivement cette représentation ; toutes les bibliothèques C++ modernes utilisent l'approche présentée ici. `std::string std::string`

Dans ces situations, il est généralement entendu que, bien qu'il soit acceptable pour un autre code de créer des pointeurs temporaires vers la mémoire possédée, il incombe à ce code de s'assurer que ses pointeurs ont disparu avant que le propriétaire ne décide de détruire l'objet possédé. Vous pouvez créer un pointeur vers un caractère vivant dans la mémoire tampon d'un tampon, mais lorsque la chaîne est détruite, votre pointeur devient invalide et c'est à vous de vous assurer que vous ne l'utilisez plus. Le propriétaire détermine la durée de vie du propriétaire, et tous les autres doivent respecter ses décisions. `std::string`

Nous avons utilisé ici comme exemple de ce à quoi ressemble la propriété en C ++ : c'est juste une convention que la bibliothèque standard suit généralement, et bien que le langage vous encourage à suivre des pratiques similaires, la façon dont vous concevez vos propres types dépend en fin de compte de vous. `std::string`

Dans Rust, cependant, le concept de propriété est intégré dans le langage lui-même et appliqué par des contrôles au moment de la compilation. Chaque valeur a un seul propriétaire qui détermine sa durée de vie. Lorsque le propriétaire est libéré – *abandonné*, dans la terminologie Rust – la valeur possédée est également abandonnée. Ces règles sont destinées à vous permettre de trouver facilement la durée de vie d'une valeur donnée simplement en inspectant le code, ce qui vous donne le contrôle sur sa durée de vie qu'un langage système devrait fournir.

Une variable possède sa valeur. Lorsque le contrôle quitte le bloc dans lequel la variable est déclarée, la variable est supprimée, de sorte que sa valeur est supprimée avec elle. Par exemple:

```
fn print_padovan() {
    let mut padovan = vec![1,1,1];  // allocated here
    for i in 3..10 {
        let next = padovan[i-3] + padovan[i-2];
        padovan.push(next);
    }
    println!("P(1..10) = {:?}", padovan);
}                                   // dropped here
```

Le type de la variable est , un vecteur d'entiers 32 bits. En mémoire, la valeur finale de ressemblera à la figure 4-2. `padovan Vec<i32> padovan`

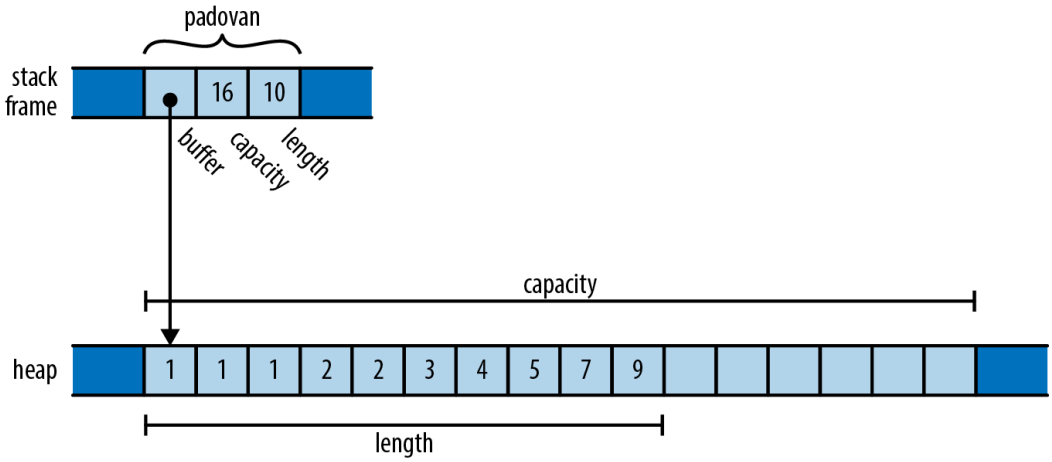

Figure 4-2. A sur la pile, pointant vers son tampon dans le tas `Vec<i32>`

Ceci est très similaire au C ++ que nous avons montré précédemment, sauf que les éléments dans la mémoire tampon sont des valeurs 32 bits, pas des caractères. Notez que les mots qui maintiennent le pointeur, la capacité et la longueur de la fonction vivent directement dans le cadre de la pile de la fonction; seul le tampon du vecteur est alloué sur le tas. `std::string padovan print_padovan`

Comme pour la chaîne précédente, le vecteur possède le tampon contenant ses éléments. Lorsque la variable sort de la portée à la fin de la fonction, le programme laisse tomber le vecteur. Et puisque le vecteur possède son tampon, le tampon va avec. `s padovan`

Le type de rouille sert d'autre exemple de propriété. A est un pointeur vers une valeur de type stockée sur le tas. L'appel alloue de l'espace de tas, y déplace la valeur et renvoie un pointage vers l'espace de tas. Puisqu'un possède l'espace vers lequel il pointe, lorsque le est lâché, il libère également l'espace. `Box Box<T> T Box::new(v) v Box Box Box`

Par exemple, vous pouvez allouer un tuple dans le tas comme suit :

```
{
    let point = Box::new((0.625, 0.5));  // point allocated here
    let label = format!("{:?}", point);  // label allocated here
    assert_eq!(label, "(0.625, 0.5)");
}                                        // both dropped here
```

Lorsque le programme appelle , il alloue de l'espace pour un tuple de deux valeurs sur le tas, déplace son argument dans cet espace et lui renvoie un pointeur. Au moment où le contrôle atteint l'appel à , le cadre de la pile ressemble à la figure 4-3. `Box::new f64 (0.625, 0.5) assert_eq!`

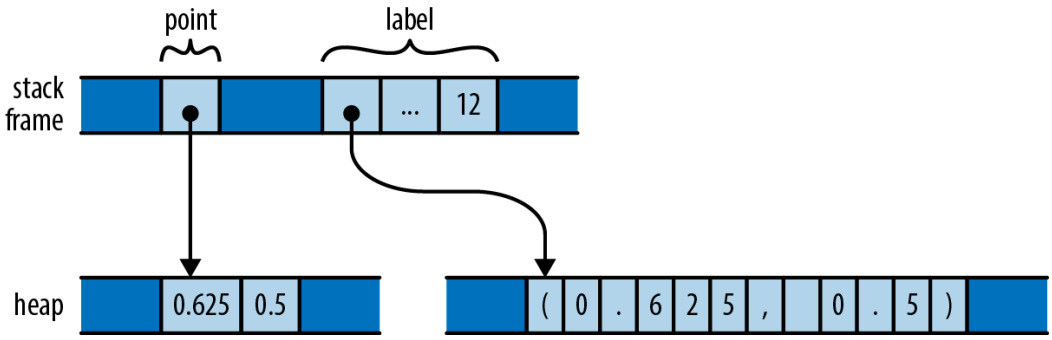

Figure 4-3. Deux variables locales, chacune possédant de la mémoire dans le tas

Le cadre de pile lui-même contient les variables et , chacune d'entre elles faisant référence à une allocation de tas qu'elle possède. Lorsqu'ils sont abandonnés, les allocations qu'ils possèdent sont libérées avec eux. `point label`

Tout comme les variables possèdent leurs valeurs, les structures possèdent leurs champs et les tuples, les tableaux et les vecteurs possèdent leurs éléments :

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
composers.push(Person { name: "Dowland".to_string(),
                        birth: 1563 });
composers.push(Person { name: "Lully".to_string(),
                        birth: 1632 });
for composer in &composers {
    println!("{}, born {}", composer.name, composer.birth);
}
```

Ici, est un , un vecteur de structs, dont chacun contient une chaîne et un nombre. En mémoire, la valeur finale de ressemble à la figure 4-4. `composers Vec<Person> composers`

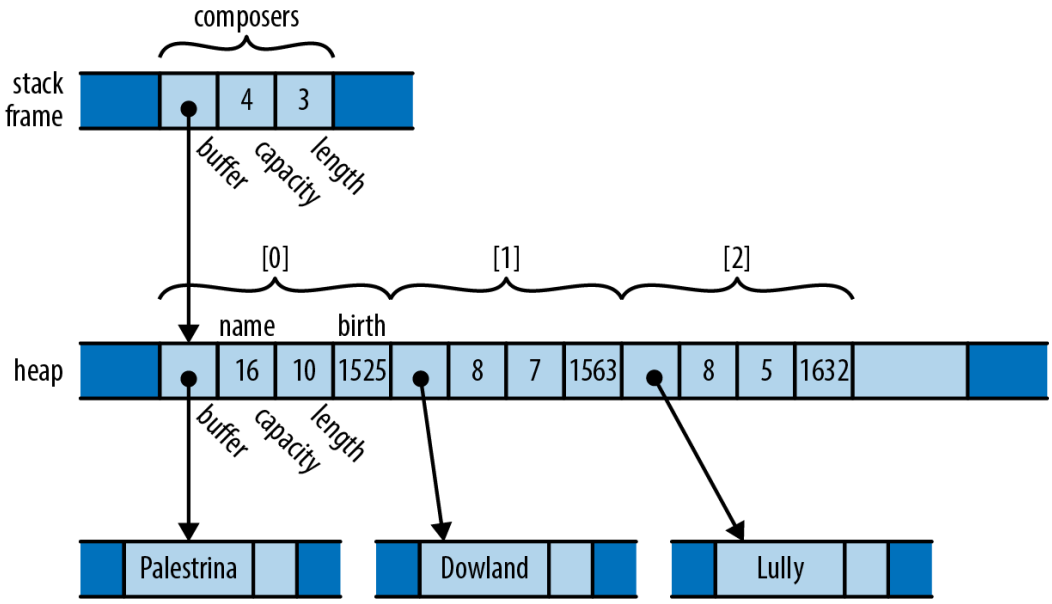

Figure 4-4. Un arbre de propriété plus complexe

Il existe de nombreuses relations de propriété ici, mais chacune est assez simple: possède un vecteur; le vecteur possède ses éléments, dont chacun est une structure; chaque structure possède ses champs; et le champ de chaîne possède son texte. Lorsque le contrôle quitte la portée dans laquelle est déclaré, le programme abandonne sa valeur et emporte l'ensemble de l'arrangement avec lui. S'il y avait d'autres types de collections dans l'image – une, peut-être, ou une – l'histoire serait la même. `composers Person composers HashMap BTreeSet`

À ce stade, prenez du recul et considérez les conséquences des relations de propriété que nous avons présentées jusqu'à présent. Chaque valeur a un seul propriétaire, ce qui facilite la décision de la laisser tomber. Mais une seule valeur peut posséder beaucoup d'autres valeurs : par exemple, le vecteur possède tous ses éléments. Et ces valeurs peuvent posséder d'autres valeurs à tour de rôle: chaque élément de possède une chaîne, qui possède son texte. `composers composers`

Il s'ensuit que les propriétaires et leurs valeurs propres forment *des arbres*: votre propriétaire est votre parent, et les valeurs que vous possédez sont vos enfants. Et à la racine ultime de chaque arbre se trouve une variable; lorsque cette variable sort de la portée, tout l'arbre l'accompagne. On peut voir un tel arbre de propriété dans le diagramme pour : ce n'est pas un « arbre » au sens d'une structure de données d'arborescence de recherche, ou un document HTML fait à partir d'éléments DOM. Nous avons plutôt un arbre construit à partir d'un mélange de types, avec la règle du propriétaire unique de Rust interdisant toute réunification de structure qui pourrait rendre l'arrangement plus complexe qu'un arbre. Chaque valeur d'un programme Rust est membre d'un arbre, enraciné dans une variable. `composers`

Les programmes Rust ne suppriment généralement pas explicitement les valeurs, de la même manière que les programmes C et C ++ utiliseraient et . La façon de supprimer une valeur dans Rust est de la supprimer de l'arbre de propriété d'une manière ou d'une autre: en quittant la portée d'une variable, ou en supprimant un élément d'un vecteur, ou quelque chose de ce genre. À ce stade, Rust s'assure que la valeur est correctement abandonnée, ainsi que tout ce qu'elle possède. `free delete`

Dans un certain sens, Rust est moins puissant que d'autres langages : tous les autres langages de programmation pratiques vous permettent de construire des graphiques arbitraires d'objets qui pointent les uns vers les autres de la manière que vous jugez appropriée. Mais c'est précisément parce que Rust est moins puissant que les analyses que le langage peut effectuer sur vos programmes peuvent être plus puissantes. Les garanties de sécurité de Rust sont possibles précisément parce que les relations qu'il peut rencontrer dans votre code sont plus faciles à gérer. Cela fait partie du « pari radical » de Rust que nous avons mentionné plus tôt: dans la pratique, affirme Rust, il y a généralement plus qu'assez de flexibilité dans la façon dont on résout un problème pour s'assurer qu'au moins quelques solutions parfaitement fines entrent dans les restrictions imposées par le langage.

Cela dit, le concept de propriété tel que nous l'avons expliqué jusqu'à présent est encore beaucoup trop rigide pour être utile. Rust étend cette idée simple de plusieurs façons:

- Vous pouvez déplacer des valeurs d'un propriétaire à un autre. Cela vous permet de construire, de réorganiser et de démolir l'arbre.
- Les types très simples comme les entiers, les nombres à virgule flottante et les caractères sont exemptés des règles de propriété. C'est ce qu'on appelle les types. `Copy`
- La bibliothèque standard fournit les types de pointeurs comptés par référence et , qui permettent aux valeurs d'avoir plusieurs propriétaires, sous certaines restrictions. `Rc Arc`
- Vous pouvez « emprunter une référence » à une valeur; les références sont des pointeurs non propriétaires, avec des durées de vie limitées.

Chacune de ces stratégies apporte de la flexibilité au modèle de propriété, tout en respectant les promesses de Rust. Nous expliquerons chacun d'eux à tour de rôle, avec des références couvertes dans le chapitre suivant.

Se déplace

Dans Rust, pour la plupart des types, les opérations telles que l'affectation d'une valeur à une variable, son passage à une fonction ou son renvoi à partir d'une fonction ne copient pas la valeur : elles la *déplacent*. La source cède la propriété de la valeur à la destination et devient non initialisée; la destination contrôle désormais la durée de vie de la valeur. Les programmes de rouille construisent et démolissent des structures complexes une valeur à la fois, un mouvement à la fois.

Vous serez peut-être surpris que Rust change le sens de ces opérations fondamentales; L'affectation est sûrement quelque chose qui devrait être assez bien cloué à ce stade de l'histoire. Cependant, si vous regardez de près comment différentes langues ont choisi de gérer les devoirs, vous verrez qu'il y a en fait une variation significative d'une école à l'autre. La comparaison rend également la signification et les conséquences du choix de Rust plus faciles à voir.

Considérez le code Python suivant :

```
s = ['udon', 'ramen', 'soba']
t = s
u = s
```

Chaque objet Python porte un nombre de références, suivant le nombre de valeurs qui y font actuellement référence. Ainsi, après l'affectation à , l'état du programme ressemble à la figure 4-5 (notez que certains champs sont omis). `s`

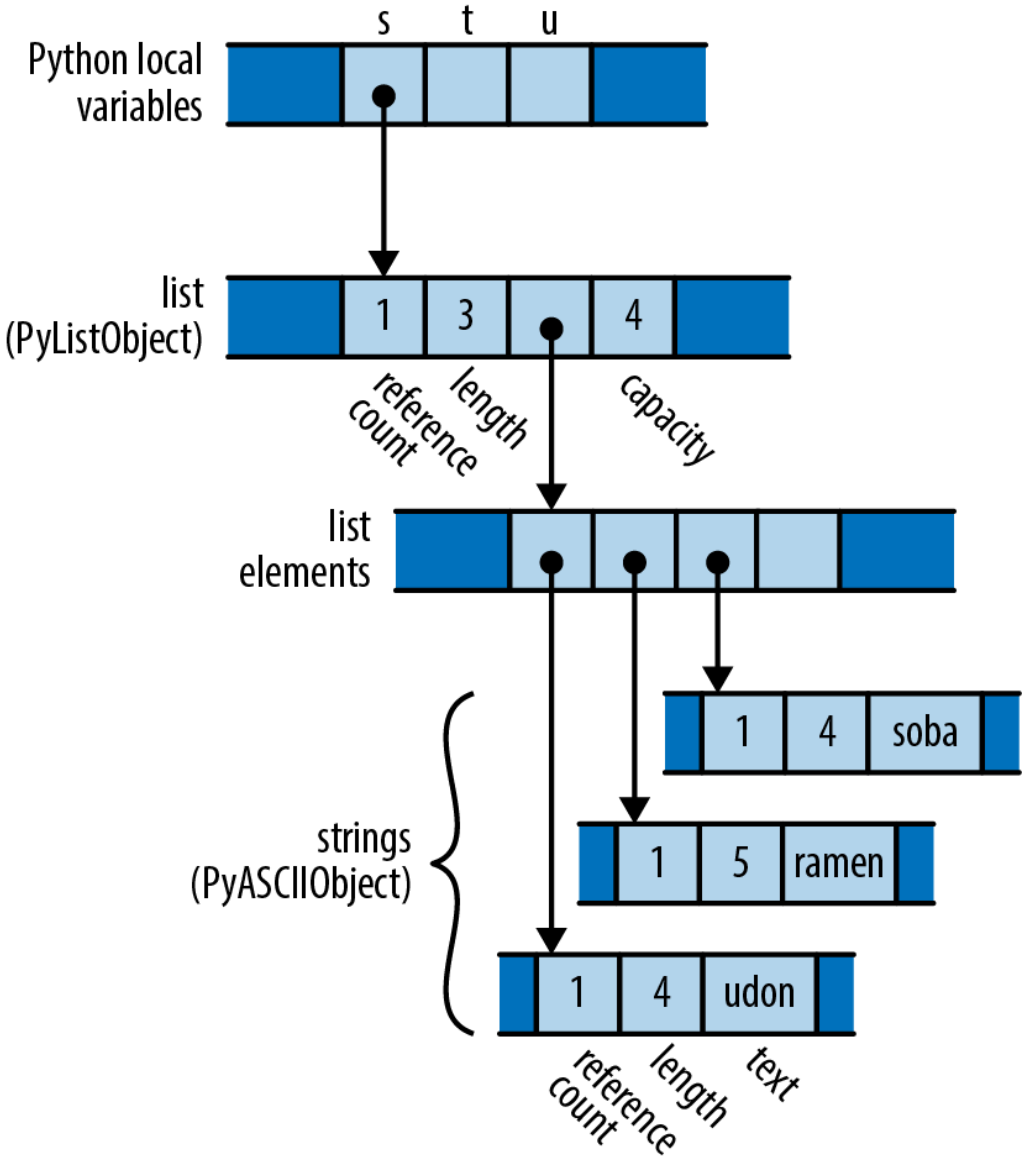

Figure 4-5. Comment Python représente une liste de chaînes en mémoire

Étant donné qu'il ne pointe que vers la liste, le nombre de références de la liste est de 1; et puisque la liste est le seul objet pointant vers les chaînes, chacun de leurs nombres de références est également égal à 1. `s`

Que se passe-t-il lorsque le programme exécute les affectations à et ? Python implémente l'affectation simplement en faisant pointer le point de destination vers le même objet que la source et en incrémentant le nombre de références de l'objet. Donc, l'état final du programme est quelque chose comme la figure 4-6. `t u`

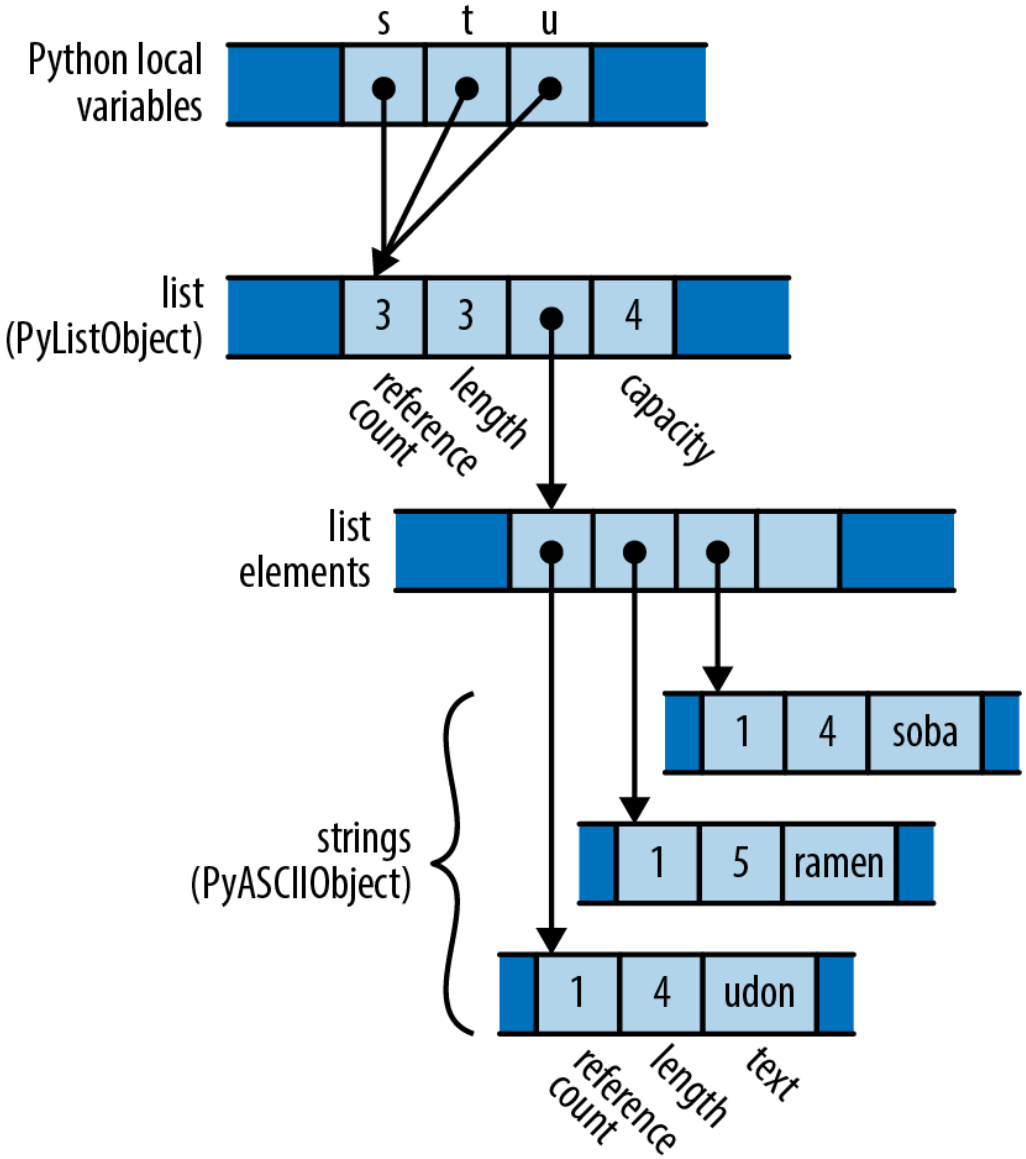

Figure 4-6. Le résultat de l'affectation aux deux et en Python `s t u`

Python a copié le pointeur de l'intérieur et mis à jour le nombre de références de la liste à 3. L'affectation en Python est bon marché, mais comme elle crée une nouvelle référence à l'objet, nous devons maintenir le nombre de références pour savoir quand nous pouvons libérer la valeur. `s t u`

Considérons maintenant le code C++ analogue :

```
using namespace std;
vector<string> s = { "udon", "ramen", "soba" };
vector<string> t = s;
vector<string> u = s;
```

La valeur d'origine de ressemble à la Figure 4-7 en mémoire. `s`

Que se passe-t-il lorsque le programme affecte à et ? L'affectation d'un produit une copie du vecteur en C++ ; se comporte de la même manière. Ainsi, au moment où le programme atteint la fin de ce code, il a en fait al-

loué trois vecteurs et neuf chaînes (Figure 4-8). `s t u std::vector std::string`

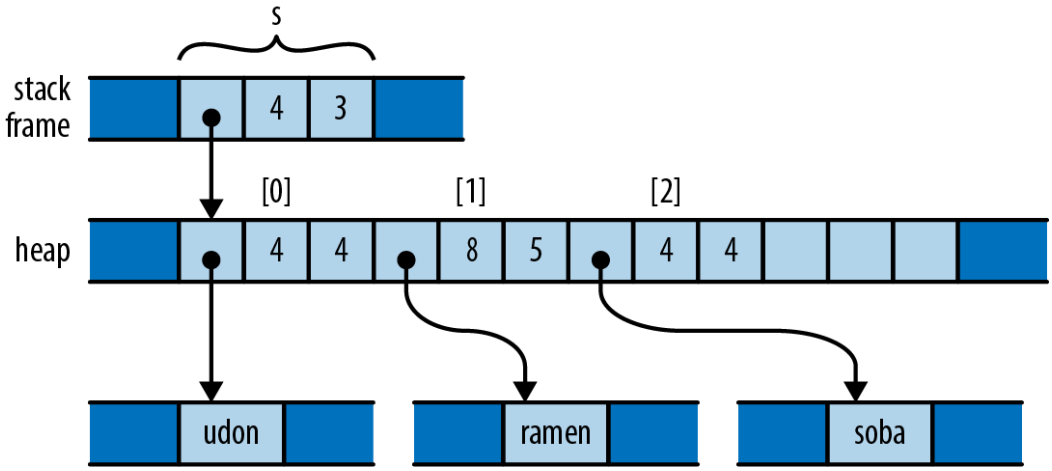

Figure 4-7. Comment C++ représente un vecteur de chaînes en mémoire

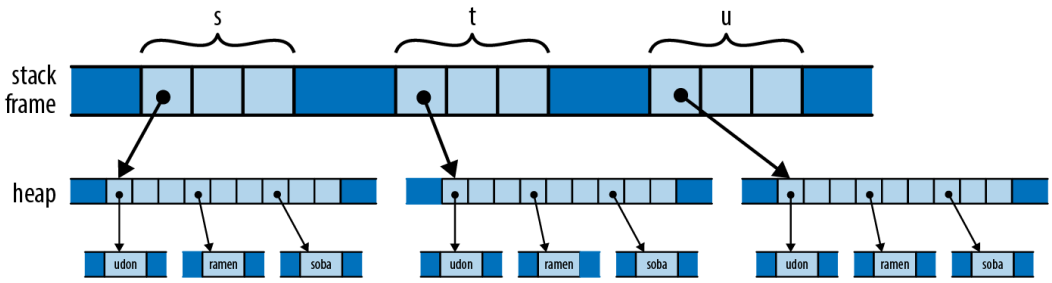

Figure 4-8. Résultat de l'affectation aux deux et en C++ `s t u`

Selon les valeurs impliquées, l'affectation en C++ peut consommer des quantités illimitées de mémoire et de temps processeur. L'avantage, cependant, est qu'il est facile pour le programme de décider quand libérer toute cette mémoire: lorsque les variables sortent du champ d'application, tout ce qui est alloué ici est nettoyé automatiquement.

Dans un sens, C++ et Python ont choisi des compromis opposés : Python rend l'affectation bon marché, au détriment du comptage des références (et dans le cas général, du garbage collection). C++ maintient la propriété de toute la mémoire claire, au détriment de l'affectation effectuer une copie profonde de l'objet. Les programmeurs C++ sont souvent moins qu'enthousiastes à propos de ce choix : les copies profondes peuvent être coûteuses, et il existe généralement des alternatives plus pratiques.

Alors, que ferait le programme analogue dans Rust? Voici le code :

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()]
let t = s;
let u = s;
```

Comme C et C ++, Rust met des littéraux de chaîne simples comme dans la mémoire en lecture seule, donc pour une comparaison plus claire avec les

exemples C ++ et Python, nous appelons ici pour obtenir des valeurs allouées au tas. `"udon" to_string String`

Après avoir effectué l'initialisation de , puisque Rust et C++ utilisent des représentations similaires pour les vecteurs et les chaînes, la situation ressemble à ce qu'elle était en C++ (Figure 4-9). `s`

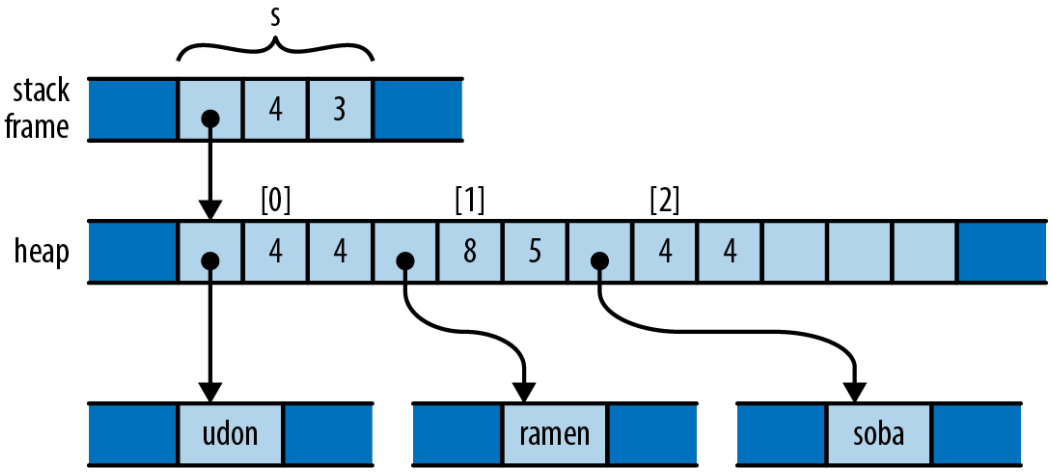

Figure 4-9. Comment Rust représente un vecteur de chaînes en mémoire

Mais rappelez-vous que, dans Rust, les affectations de la plupart des types *déplacent* la valeur de la source vers la destination, laissant la source non initialisée. Ainsi, après l'initialisation , la mémoire du programme ressemble à la Figure 4-10. `t`

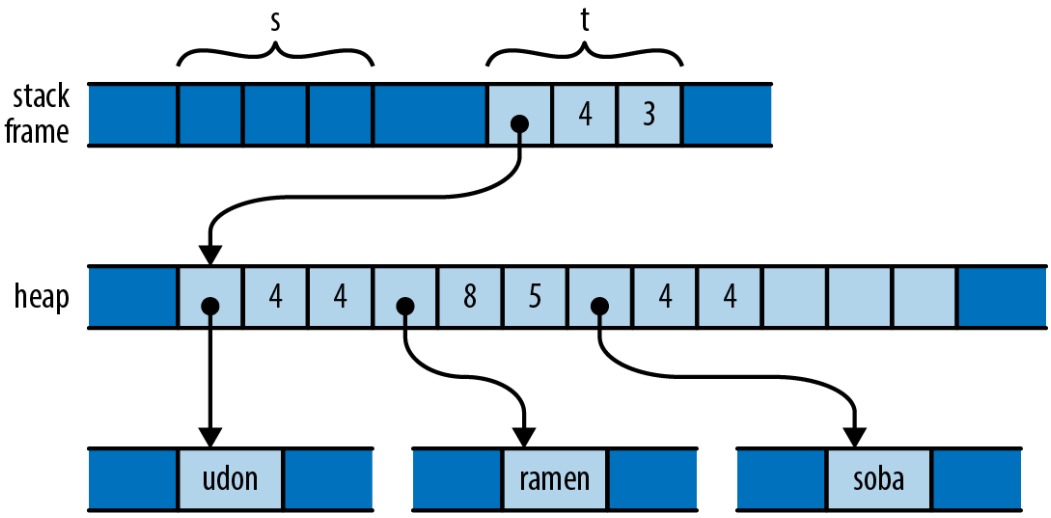

Figure 4-10. Le résultat de l'affectation à dans Rust `s t`

Que s'est-il passé ici? L'initialisation déplaçait les trois champs d'en-tête du vecteur de à ; possède maintenant le vecteur. Les éléments du vecteur sont restés là où ils étaient, et rien n'est arrivé aux cordes non plus. Chaque valeur a toujours un seul propriétaire, bien que l'on ait changé de mains. Il n'y avait pas de dénombrement de référence à ajuster. Et le compilateur considère maintenant comme non initialisé. `let t = s; s t t s`

Alors que se passe-t-il lorsque nous atteignons l'initialisation ? Cela affecterait la valeur non initialisée à . Rust interdit prudemment l'utilisa-

tion de valeurs non initialisées, de sorte que le compilateur rejette ce code avec l'erreur suivante : `let u = s; s u`

```
error: use of moved value: `s`
  |
7 |     let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_s
  |         - move occurs because `s` has type `Vec<String>`,
  |           which does not implement the `Copy` trait
8 |     let t = s;
  |             - value moved here
9 |     let u = s;
  |             ^ value used here after move
```

Considérez les conséquences de l'utilisation d'un déménagement par Rust ici. Comme Python, l'affectation est bon marché: le programme déplace simplement l'en-tête de trois mots du vecteur d'un endroit à un autre. Mais comme C++, la propriété est toujours claire : le programme n'a pas besoin de comptage de références ou de garbage collection pour savoir quand libérer les éléments vectoriels et le contenu des chaînes.

Le prix que vous payez est que vous devez demander explicitement des copies quand vous le souhaitez. Si vous voulez vous retrouver dans le même état que le programme C++, chaque variable contenant une copie indépendante de la structure, vous devez appeler la méthode du vecteur, qui effectue une copie profonde du vecteur et de ses éléments : `clone`

```
let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_string()]
let t = s.clone();
let u = s.clone();
```

Vous pouvez également recréer le comportement de Python en utilisant les types de pointeurs comptés par référence de Rust ; nous en discuterons sous peu dans « Rc et Arc: Shared Ownership ».

Plus d'opérations qui se déplacent

Dans les exemples jusqu'à présent, nous avons montré des initialisations, fournissant des valeurs pour les variables au fur et à mesure qu'elles entrent dans la portée d'une instruction. L'affectation à une variable est légèrement différente, en ce sens que si vous déplacez une valeur dans une variable déjà initialisée, Rust supprime la valeur antérieure de la variable. Par exemple: `let`

```
let mut s = "Govinda".to_string();
s = "Siddhartha".to_string(); // value "Govinda" dropped here
```

Dans ce code, lorsque le programme affecte la chaîne à , sa valeur antérieure est supprimée en premier. Mais considérez ce qui suit: `"Siddhartha" s "Govinda"`

```
let mut s = "Govinda".to_string();
let t = s;
s = "Siddhartha".to_string(); // nothing is dropped here
```

Cette fois, a pris possession de la chaîne d'origine de , de sorte qu'au moment où nous l'affectons à , elle n'est pas initialisée. Dans ce scénario, aucune chaîne n'est supprimée. `t s s`

Nous avons utilisé des initialisations et des affectations dans les exemples ici parce qu'elles sont simples, mais Rust applique la sémantique de déplacement à presque toutes les utilisations d'une valeur. Le passage d'arguments aux fonctions déplace la propriété vers les paramètres de la fonction ; le renvoi d'une valeur d'une fonction déplace la propriété vers l'appelant. La construction d'un tuple déplace les valeurs dans le tuple. Et ainsi de suite.

Vous avez peut-être maintenant un meilleur aperçu de ce qui se passe réellement dans les exemples que nous avons proposés dans la section précédente. Par exemple, lorsque nous construisions notre vecteur de compositeurs, nous écrivions :

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

Ce code indique plusieurs endroits où les déplacements se produisent, au-delà de l'initialisation et de l'affectation :

Renvoi de valeurs à partir d'une fonction

L'appel construit un nouveau vecteur et renvoie, non pas un pointeur vers le vecteur, mais le vecteur lui-même : sa propriété se déplace de vers la variable . De même, l'appel renvoie une nouvelle instance. `Vec::new() Vec::new composers to_string String`

Construire de nouvelles valeurs

Le champ de la nouvelle structure est initialisé avec la valeur de retour de . La structure prend possession de la chaîne. `name Person to_string`

Transmission de valeurs à une fonction

> La structure entière, et non un pointeur vers elle, est transmise à la méthode du vecteur, qui la déplace à l'extrémité de la structure. Le vecteur prend possession du et devient ainsi le propriétaire indirect du nom. `Person push Person String`

Déplacer des valeurs comme celle-ci peut sembler inefficace, mais il y a deux choses à garder à l'esprit. Tout d'abord, les mouvements s'appliquent toujours à la valeur proprement dite, et non au stockage en tas qu'ils possèdent. Pour les vecteurs et les chaînes, la *valeur propre* est l'entête de trois mots seul ; les tableaux d'éléments et les tampons de texte potentiellement volumineux se trouvent à l'endroit où ils se trouvent dans le tas. Deuxièmement, la génération de code du compilateur Rust est bonne pour « voir à travers » tous ces mouvements; en pratique, le code machine stocke souvent la valeur directement là où elle appartient.

Déplacements et contrôle du flux

Les exemples précédents ont tous un flux de contrôle très simple; Comment les mouvements interagissent-ils avec un code plus compliqué ? Le principe général est que, s'il est possible qu'une variable ait vu sa valeur déplacée et qu'elle n'ait pas définitivement reçu une nouvelle valeur depuis, elle est considérée comme non initialisée. Par exemple, si une variable a encore une valeur après avoir évalué la condition d'une expression, nous pouvons l'utiliser dans les deux branches : `if`

```
let x = vec![10, 20, 30];
if c {
    f(x); // ... ok to move from x here
} else {
    g(x); // ... and ok to also move from x here
}
h(x); // bad: x is uninitialized here if either path uses it
```

Pour des raisons similaires, il est interdit de passer d'une variable dans une boucle :

```
let x = vec![10, 20, 30];
while f() {
    g(x); // bad: x would be moved in first iteration,
          // uninitialized in second
}
```

C'est-à-dire, à moins que nous ne lui ayons définitivement donné une nouvelle valeur d'ici la prochaine itération:

```
let mut x = vec![10, 20, 30];
while f() {
    g(x);           // move from x
    x = h();        // give x a fresh value
}
e(x);
```

Déplacements et contenu indexé

Nous avons mentionné qu'un déménagement laisse sa source non initialisée, car la destination prend possession de la valeur. Mais tous les types de propriétaires de valeur ne sont pas prêts à devenir non initialisés. Par exemple, considérez le code suivant :

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// Pull out random elements from the vector.
let third = v[2]; // error: Cannot move out of index of Vec
let fifth = v[4]; // here too
```

Pour que cela fonctionne, Rust devrait en quelque sorte se rappeler que les troisième et cinquième éléments du vecteur sont devenus non initialisés et suivre cette information jusqu'à ce que le vecteur soit abandonné. Dans le cas le plus général, les vecteurs devraient emporter avec eux des informations supplémentaires pour indiquer quels éléments sont vivants et lesquels sont devenus non initialisés. Ce n'est clairement pas le bon comportement pour un langage de programmation de systèmes; un vecteur ne doit être rien d'autre qu'un vecteur. En fait, Rust rejette le code précédent avec l'erreur suivante :

```
error: cannot move out of index of `Vec<String>`
   |
14 |     let third = v[2];
   |                 ^^^^
   |                 |
   |                 move occurs because value has type `String`,
   |                 which does not implement the `Copy` trait
   |                 help: consider borrowing here: `&v[2]`
```

Il fait également une plainte similaire au sujet du déménagement à . Dans le message d'erreur, Rust suggère d'utiliser une référence, au cas où vous

voudriez accéder à l'élément sans le déplacer. C'est souvent ce que vous voulez. Mais que se passe-t-il si vous voulez vraiment déplacer un élément hors d'un vecteur ? Vous devez trouver une méthode qui le fait d'une manière qui respecte les limites du type. Voici trois possibilités : `fifth`

```
// Build a vector of the strings "101", "102", ... "105"
let mut v = Vec::new();
for i in 101 .. 106 {
    v.push(i.to_string());
}

// 1. Pop a value off the end of the vector:
let fifth = v.pop().expect("vector empty!");
assert_eq!(fifth, "105");

// 2. Move a value out of a given index in the vector,
// and move the last element into its spot:
let second = v.swap_remove(1);
assert_eq!(second, "102");

// 3. Swap in another value for the one we're taking out:
let third = std::mem::replace(&mut v[2], "substitute".to_string());
assert_eq!(third, "103");

// Let's see what's left of our vector.
assert_eq!(v, vec!["101", "104", "substitute"]);
```

Chacune de ces méthodes déplace un élément hors du vecteur, mais le fait d'une manière qui laisse le vecteur dans un état entièrement peuplé, même s'il est peut-être plus petit.

Les types de collection comme proposent également généralement des méthodes pour consommer tous leurs éléments dans une boucle: `Vec`

```
let v = vec!["liberté".to_string(),
             "égalité".to_string(),
             "fraternité".to_string()];

for mut s in v {
    s.push('!');
    println!("{}", s);
}
```

Lorsque nous passons directement le vecteur à la boucle, comme dans , cela *déplace* le vecteur hors de , laissant non initialisé. La machinerie interne de la boucle s'approprie le vecteur et le dissèque en ses éléments. À

chaque itération, la boucle déplace un autre élément vers la variable . Puisque maintenant possède la chaîne, nous sommes en mesure de la modifier dans le corps de la boucle avant de l'imprimer. Et puisque le vecteur lui-même n'est plus visible par le code, rien ne peut l'observer en boucle dans un état partiellement vidé. `for ... in v v v for s s`

Si vous avez besoin de déplacer une valeur d'un propriétaire que le compilateur ne peut pas suivre, vous pouvez envisager de changer le type du propriétaire en quelque chose qui peut suivre dynamiquement s'il a une valeur ou non. Par exemple, voici une variante de l'exemple précédent :

```
struct Person { name: Option<String>, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: Some("Palestrina".to_string()),
                        birth: 1525 });
```

Vous ne pouvez pas faire ça:

```
let first_name = composers[0].name;
```

Cela suscitera simplement la même erreur « ne peut pas sortir de l'index » montrée précédemment. Mais parce que vous avez changé le type du champ de à , cela signifie qu'il s'agit d'une valeur légitime pour le champ à conserver, donc cela fonctionne
: `name String Option<String> None`

```
let first_name = std::mem::replace(&mut composers[0].name, None);
assert_eq!(first_name, Some("Palestrina".to_string()));
assert_eq!(composers[0].name, None);
```

L'appel déplace la valeur de , laissant à sa place et transmet la propriété de la valeur d'origine à son appelant. En fait, l'utilisation de cette méthode est suffisamment courante pour que le type fournisse une méthode à cette fin. Vous pouvez écrire la manipulation précédente de manière plus lisible comme suit : `replace composers[0].name None Option take`

```
let first_name = composers[0].name.take();
```

Cet appel à a le même effet que l'appel précédent à . `take replace`

Types de copie : exception aux déplacements

Les exemples que nous avons montrés jusqu'à présent de valeurs déplacées impliquent des vecteurs, des chaînes et d'autres types qui pourraient potentiellement utiliser beaucoup de mémoire et être coûteux à copier. Les déménagements gardent la propriété de ces types claire et l'affectation bon marché. Mais pour les types plus simples comme les entiers ou les caractères, ce genre de manipulation prudente n'est vraiment pas nécessaire.

Comparez ce qui se passe en mémoire lorsque nous attribuons un avec ce qui se passe lorsque nous attribuons une valeur : `String i32`

```
let string1 = "somnambulance".to_string();
let string2 = string1;

let num1: i32 = 36;
let num2 = num1;
```

Après avoir exécuté ce code, la mémoire ressemble à la Figure 4-11.

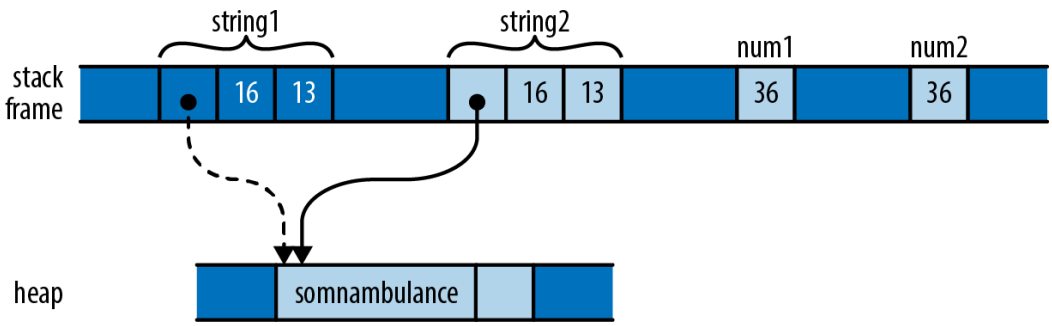

Figure 4-11. L'affectation d'un déplace la valeur, tandis que l'affectation d'un copie `String i32`

Comme pour les vecteurs précédents, l'affectation *se déplace vers* afin que nous ne nous retrouvions pas avec deux chaînes responsables de la libération du même tampon. Cependant, la situation avec et est différente. An est simplement un modèle de bits en mémoire; il ne possède aucune ressource de tas ou ne dépend vraiment de rien d'autre que des octets qu'il comprend. Au moment où nous avons déplacé ses bits vers , nous avons fait une copie complètement indépendante de
`. string1 string2 num1 num2 i32 num2 num1`

Le déplacement d'une valeur laisse la source du déplacement non initialisée. Mais alors qu'il est essentiel de traiter comme sans valeur, traiter de cette façon est inutile; aucun dommage ne pourrait résulter de la poursuite de son utilisation. Les avantages d'un déménagement ne s'appliquent pas ici, et c'est gênant. `string1 num1`

Plus tôt, nous avons pris soin de dire que *la plupart des* types sont déplacés; nous en sommes maintenant arrivés aux exceptions, les types que Rust désigne comme *types `de copie`*. L'affectation d'une valeur d'un

type copie la valeur plutôt que de la déplacer. La source de l'affectation reste initialisée et utilisable, avec la même valeur qu'auparavant. Le passage de types aux fonctions et aux constructeurs se comporte de la même manière. `Copy` `Copy`

Les types standard incluent tous les types numériques entiers et à virgule flottante de la machine, les types et et quelques autres. Un tuple ou un tableau de types de taille fixe est lui-même un type. `Copy` `char` `bool` `Copy` `Copy`

Seuls les types pour lesquels une simple copie bit pour bit suffit peuvent être . Comme nous l'avons déjà expliqué, n'est pas un type, car il possède un tampon alloué au tas. Pour des raisons similaires, n'est pas ; il est propriétaire de son référent attribué en tas. Le type, représentant un descripteur de fichier du système d'exploitation, n'est pas ; la duplication d'une telle valeur impliquerait de demander au système d'exploitation un autre descripteur de fichier. De même, le type, représentant un mutex verrouillé, ne l'est pas : ce type n'a aucun sens à copier, car un seul thread peut contenir un mutex à la fois. `Copy` `String` `Copy` `Box<T>` `Copy` `File` `Copy` `MutexGuard` `Copy`

En règle générale, tout type qui a besoin de faire quelque chose de spécial lorsqu'une valeur est supprimée ne peut pas être : un besoin de libérer ses éléments, un doit fermer son descripteur de fichier, un doit déverrouiller son mutex, et ainsi de suite. La duplication bit pour bit de ces types ne permettrait pas de savoir quelle valeur était maintenant responsable des ressources de l'original. `Copy` `Vec` `File` `MutexGuard`

Qu'en est-il des types que vous définissez vous-même? Par défaut, et les types ne sont pas : `struct` `enum` `Copy`

```
struct Label { number: u32 }

fn print(l: Label) { println!("STAMP: {}", l.number); }

let l = Label { number: 3 };
print(l);
println!("My label number is: {}", l.number);
```

Cela ne compilera pas; Rust se plaint:

```
error: borrow of moved value: `l`
   |
10 |     let l = Label { number: 3 };
   |         - move occurs because `l` has type `main::Label`,
   |           which does not implement the `Copy` trait
```

```
11 |     print(l);
   |           - value moved here
12 |     println!("My label number is: {}", l.number);
   |                                        ^^^^^^^^
   |                   value borrowed here after move
```

Puisque n'est pas , en le passant pour déplacer la propriété de la valeur vers la fonction, qui l'a ensuite abandonnée avant de revenir. Mais c'est idiot; a n'est rien d'autre qu'un avec des prétentions. Il n'y a aucune raison de passer à devrait déplacer la valeur. `Label Copy print print Label u32 l print`

Mais les types définis par l'utilisateur étant non- n'est que la valeur par défaut. Si tous les champs de votre structure sont eux-mêmes , alors vous pouvez également faire le type en plaçant l'attribut au-dessus de la définition, comme suit: `Copy Copy Copy #[derive(Copy, Clone)]`

```
#[derive(Copy, Clone)]
struct Label { number: u32 }
```

Avec cette modification, le code précédent se compile sans plainte. Cependant, si nous essayons cela sur un type dont les champs ne sont pas tous , cela ne fonctionne pas. Supposons que nous compilions le code suivant : `Copy`

```
#[derive(Copy, Clone)]
struct StringLabel { name: String }
```

Il déclenche cette erreur :

```
error: the trait `Copy` may not be implemented for this type
  |
7 | #[derive(Copy, Clone)]
  |          ^^^^
8 | struct StringLabel { name: String }
  |                      ------------ this field does not implement `Copy
```

Pourquoi les types définis par l'utilisateur ne sont-ils pas automatiquement définis par l'utilisateur, en supposant qu'ils sont éligibles ? Qu'un type soit ou non a un effet important sur la façon dont le code est autorisé à l'utiliser : les types sont plus flexibles, car l'affectation et les opérations associées ne laissent pas l'original non initialisé. Mais pour l'implémenteur d'un type, c'est l'inverse qui est vrai : les types sont très limités dans les types qu'ils peuvent contenir, tandis que les non-types peuvent utiliser l'allocation de tas et posséder d'autres types de ressources. Donc,

faire un type représente un engagement sérieux de la part de l'implémenteur: s'il est nécessaire de le changer en non-plus tard, une grande partie du code qui l'utilise devra probablement être adaptée. `Copy Copy Copy Copy Copy Copy Copy`

Alors que C++ vous permet de surcharger les opérateurs d'affectation et de définir des constructeurs de copie et de déplacement spécialisés, Rust ne permet pas ce type de personnalisation. Dans Rust, chaque mouvement est une copie superficielle octet pour octet qui laisse la source non initialisée. Les copies sont les mêmes, sauf que la source reste initialisée. Cela signifie que les classes C++ peuvent fournir des interfaces pratiques que les types Rust ne peuvent pas, où le code d'apparence ordinaire ajuste implicitement le nombre de références, reporte les copies coûteuses pour plus tard ou utilise d'autres astuces d'implémentation sophistiquées.

Mais l'effet de cette flexibilité sur C++ en tant que langage est de rendre les opérations de base telles que l'affectation, la transmission de paramètres et le retour de valeurs à partir de fonctions moins prévisibles. Par exemple, plus haut dans ce chapitre, nous avons montré comment l'affectation d'une variable à une autre en C++ peut nécessiter des quantités arbitraires de mémoire et de temps processeur. L'un des principes de Rust est que les coûts doivent être apparents pour le programmeur. Les opérations de base doivent rester simples. Les opérations potentiellement coûteuses doivent être explicites, comme les appels à dans l'exemple précédent qui font des copies profondes des vecteurs et des chaînes qu'ils contiennent. `clone`

Dans cette section, nous avons parlé en termes vagues des caractéristiques qu'un type pourrait avoir. Ce sont en fait des *exemples de traits*, la facilité ouverte de Rust pour catégoriser les types en fonction de ce que vous pouvez en faire. Nous décrivons les traits en général au chapitre 11, et en particulier au chapitre 13. `Copy Clone Copy Clone`

Rc et Arc : propriété partagée

Bien que la plupart des valeurs aient des propriétaires uniques dans le code Rust typique, dans certains cas, il est difficile de trouver chaque valeur d'un seul propriétaire qui a la durée de vie dont vous avez besoin; vous aimeriez que la valeur vive simplement jusqu'à ce que tout le monde ait fini de l'utiliser. Pour ces cas, Rust fournit les types de pointeur comptés par référence et . Comme on peut s'y attendre de Rust, ceux-ci sont entièrement sûrs à utiliser : vous ne pouvez pas oublier d'ajuster le nombre de références, de créer d'autres pointeurs vers le référent que

Rust ne remarque pas, ou de trébucher sur l'un des autres types de problèmes qui accompagnent les types de pointeurs comptés par référence en C++. `Rc` `Arc`

Les et types sont très similaires; la seule différence entre eux est qu'il est sûr de partager directement entre les threads (le nom est l'abréviation de *atomic reference count*), tandis qu'un simple utilise un code non thread-safe plus rapide pour mettre à jour son nombre de références. Si vous n'avez pas besoin de partager les pointeurs entre les threads, il n'y a aucune raison de payer la pénalité de performance d'un , vous devez donc utiliser ; La rouille vous empêchera d'en passer accidentellement un à travers une limite de filetage. Les deux types sont par ailleurs équivalents, donc pour le reste de cette section, nous ne parlerons que de . `Rc` `Arc` `Arc` `Arc` `Rc` `Arc` `Rc` `Rc`

Plus tôt, nous avons montré comment Python utilise le nombre de références pour gérer la durée de vie de ses valeurs. Vous pouvez utiliser pour obtenir un effet similaire dans Rust. Considérez le code suivant : `Rc`

```
use std::rc::Rc;

// Rust can infer all these types; written out for clarity
let s: Rc<String> = Rc::new("shirataki".to_string());
let t: Rc<String> = s.clone();
let u: Rc<String> = s.clone();
```

Pour tout type , une valeur est un pointeur vers un tas alloué auquel un nombre de références a été apposé. Le clonage d'une valeur ne copie pas le ; au lieu de cela, il crée simplement un autre pointeur vers celui-ci et incrémente le nombre de références. Ainsi, le code précédent produit la situation illustrée à la figure 4-12 en mémoire. `T` `Rc<T>` `T` `Rc<T>` `T`

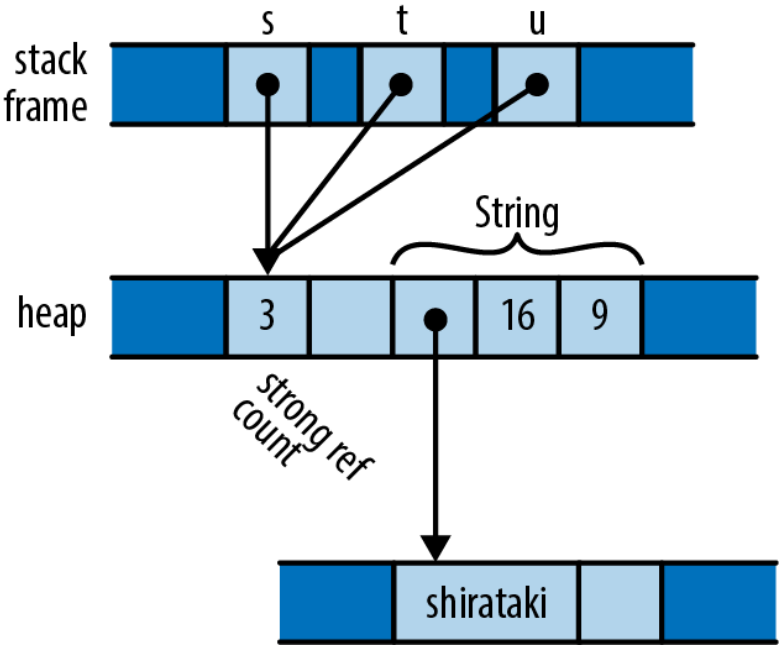

Figure 4-12. Chaîne comptée par références avec trois références

Chacun des trois pointeurs fait référence au même bloc de mémoire, qui contient un nombre de références et un espace pour le . Les règles de propriété habituelles s'appliquent aux pointeurs eux-mêmes, et lorsque le dernier existant est abandonné, Rust le laisse également tomber. `Rc<String>` `String` `Rc` `Rc` `String`

Vous pouvez utiliser n'importe laquelle des méthodes habituelles directement sur un : `String` `Rc<String>`

```
assert!(s.contains("shira"));
assert_eq!(t.find("taki"), Some(5));
println!("{} are quite chewy, almost bouncy, but lack flavor", u);
```

Une valeur appartenant à un pointeur est immuable. Supposons que vous essayiez d'ajouter du texte à la fin de la chaîne : `Rc`

```
s.push_str(" noodles");
```

La rouille diminuera :

```
error: cannot borrow data in an `Rc` as mutable
   |
13 |     s.push_str(" noodles");
   |     ^ cannot borrow as mutable
   |
```

Les garanties de sécurité de la mémoire et du thread de Rust dépendent de la garantie qu'aucune valeur n'est jamais partagée et mutable simultanément. Rust suppose que le référent d'un pointeur peut en général

être partagé, il ne doit donc pas être mutable. Nous expliquons pourquoi cette restriction est importante au chapitre 5. `Rc`

Un problème bien connu avec l'utilisation du nombre de références pour gérer la mémoire est que, si jamais il y a deux valeurs comptées par référence qui pointent l'une vers l'autre, chacune maintiendra le nombre de références de l'autre au-dessus de zéro, de sorte que les valeurs ne seront jamais libérées (Figure 4-13).

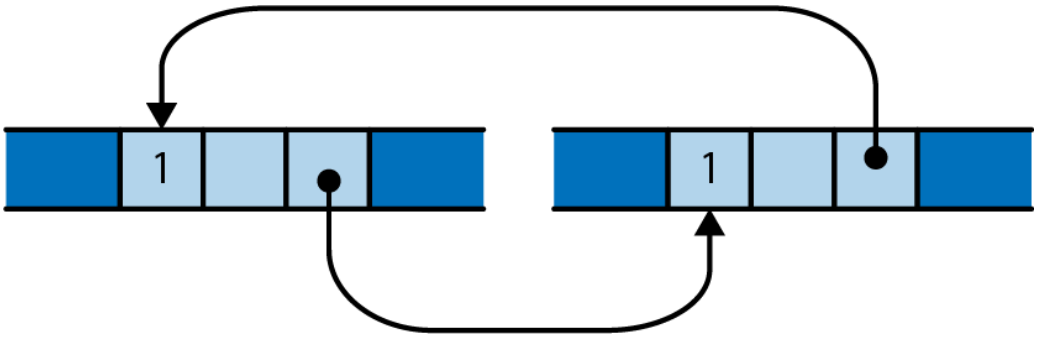

Figure 4-13. Une boucle de comptage de références; ces objets ne seront pas libérés

Il est possible de divulguer des valeurs dans Rust de cette façon, mais de telles situations sont rares. Vous ne pouvez pas créer un cycle sans, à un moment donné, faire pointer une valeur plus ancienne vers une valeur plus récente. Cela nécessite évidemment que l'ancienne valeur soit mutable. Étant donné que les pointeurs maintiennent leurs référents immuables, il n'est normalement pas possible de créer un cycle. Cependant, Rust fournit des moyens de créer des parties mutables de valeurs autrement immuables; c'est ce *qu'on appelle la mutabilité intérieure*, et nous la couvrons dans « Mutabilité intérieure ». Si vous combinez ces techniques avec des pointeurs, vous pouvez créer un cycle et une mémoire de fuite. `Rc` `Rc`

Vous pouvez parfois éviter de créer des cycles de pointeurs en utilisant des *pointeurs faibles*, , pour certains des liens à la place. Cependant, nous ne les couvrirons pas dans ce livre; consultez la documentation de la bibliothèque standard pour plus de détails. `Rc` `std::rc::Weak`

Les mouvements et les pointeurs comptés par référence sont deux façons de détendre la rigidité de l'arbre de propriété. Dans le chapitre suivant, nous examinerons une troisième voie : emprunter des références à des valeurs. Une fois que vous êtes à l'aise avec la propriété et l'emprunt, vous aurez gravi la partie la plus raide de la courbe d'apprentissage de Rust, et vous serez prêt à tirer parti des forces uniques de Rust.