Chapitre 4. Propriété et déménagements

Quand il s'agit de gérer la mémoire, il y a deux caractéristiques que nous aimerions avoir dans nos langages de programmation :

- Nous aimerions la mémoireêtre libéré 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 qu'il a été libéré. Ce serait un comportement indéfini, entraînant des plantages et des failles de sécurité.

Mais ceux-ci semblent s'exclure mutuellement : libérer une valeur alors que des pointeurs existent sur celle-ci laisse nécessairement ces pointeurs en suspens. Presque tous les principaux langages de programmation appartiennent à l'un des deux camps, en fonction de laquelle des deux qualités ils renoncent :

- Le camp "Safety First" utilise la collecte des ordurespour gérer la mémoire, en 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 ne reste plus de pointeurs vers eux. Presque tous les langages modernes appartiennent à ce camp, de Python, JavaScript et Ruby à Java, C# et Haskell.
  Mais s'appuyer sur le ramasse-miettes signifie renoncer au contrôle sur le moment exact où les objets sont remis au collecteur. En général, les ramasse-miettes sont des bêtes surprenantes, et comprendre pourquoi la mémoire n'a pas été libérée quand on s'y attendait peut être un défi.
- Le camp "Control First" vous laisse le soin 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 traditionnels de ce camp.
  C'est très bien si vous ne faites jamais d'erreurs, mais les preuves suggèrent que vous finirez par le faire. L'utilisation abusive de pointeurs a été un coupable courant dans les problèmes de sécurité signalés aussi longtemps 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 sort de l'impasse d'une manière surprenante : en limitant la façon dont vos programmes peuvent utiliser des 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 respecter les règles, et vous devrez rechercher des alternatives. Mais l'effet net de ces restrictions est d'apporter juste assez d'ordre dans le chaos pour permettre aux contrôles de compilation de Rust de vérifier que votre programme est exempt d'erreurs de sécurité mémoire : pointeurs suspendus, doubles libérations, utilisation de mémoire non initialisée, etc. Au moment de l'exécution, vos pointeurs sont de simples adresses en mémoire, comme ils le seraient en C et C++. La différence est qu'il a été prouvé que votre code les utilise en toute sécurité.

Ces mêmes règles constituent également la base de la prise en charge par Rust de la sécurité simultanéeprogrammation. En utilisant les primitives de threading soigneusement conçues de 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 défaillances difficiles à reproduire dans des parties non liées du système. Le comportement non déterministe inhérent au code multithread est isolé des fonctionnalités conçues pour le gérer (mutex, canaux de messages, valeurs atomiques, etc.) plutôt que d'apparaître dans les références mémoire ordinaires. Le code multithread en C et C++ a gagné sa mauvaise réputation, 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 qu'assez flexible pour presque toutes les tâches et que les avantages - les l'élimination de grandes classes de bogues de gestion de la mémoire et de simultanéité - justifiera les adaptations que vous devrez apporter à votre style. Les auteurs de ce livre sont optimistes sur Rust précisément à cause 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 transformer 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 produisent dans d'autres langages. 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 scé-

narios et les types qui contournent ou enfreignent certaines de ces règles afin de fournir plus de flexibilité.

La possession

Si vous avez lu beaucoup de code C ou C++, vous avez probablement rencontré un commentaire indiquant 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 possessions avec lui.

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

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

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

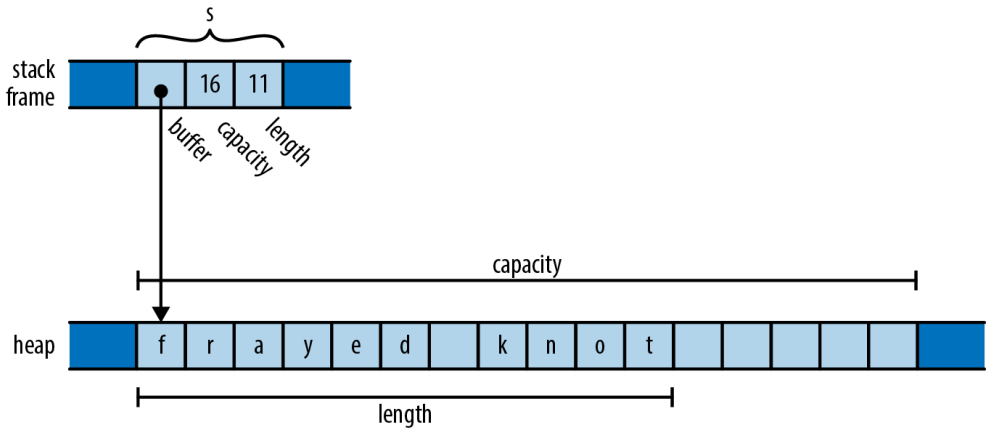

Figure 4-1. Une valeur C++ `std::string` sur la pile, pointant vers son tampon alloué par tas

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

A `std::string` 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 `std::string` valeurs, en utilisant un compteur 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.

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 est de la responsabilité de 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 `std::string` le tampon d'un , 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 de la propriété et tous les autres doivent respecter ses décisions.

Nous avons utilisé `std::string` ici un 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 finalement de vous.

Dans Rust, cependant, le concept de propriété est intégré au langage lui-même et appliqué par des vérifications au moment de la compilation. Chaque valeur a un propriétaire unique qui détermine sa durée de vie. Lorsque le propriétaire est libéré— *abandonné*, dans la terminologie Rust - la valeur possédée est également supprimée. Ces règles sont destinées à vous permettre de trouver facilement la durée de vie d'une valeur donnée simplement en inspectant le code, vous donnant 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, donc 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 `padovan` est `Vec<i32>` , un vecteur d'entiers 32 bits. En mémoire, la valeur finale de `padovan` ressemblera à la Figure 4-2 .

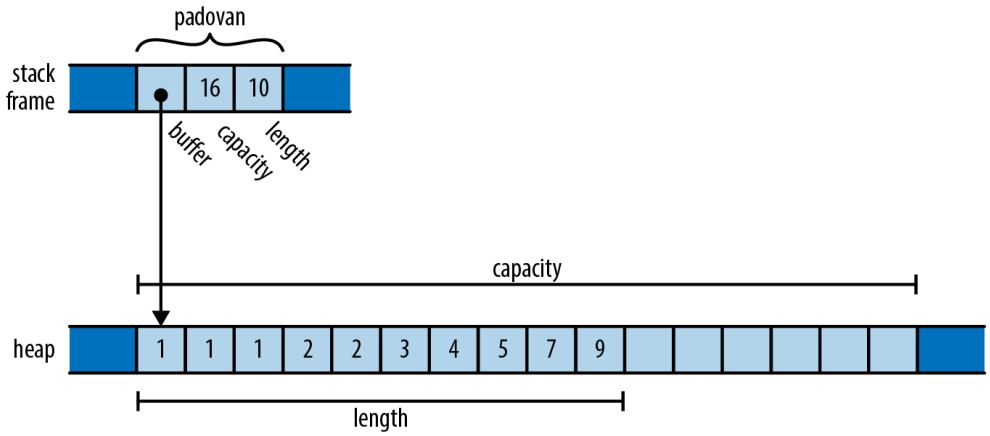

Illustration 4-2. A `Vec<i32>` sur la pile, pointant vers son tampon dans le tas

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

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

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

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

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

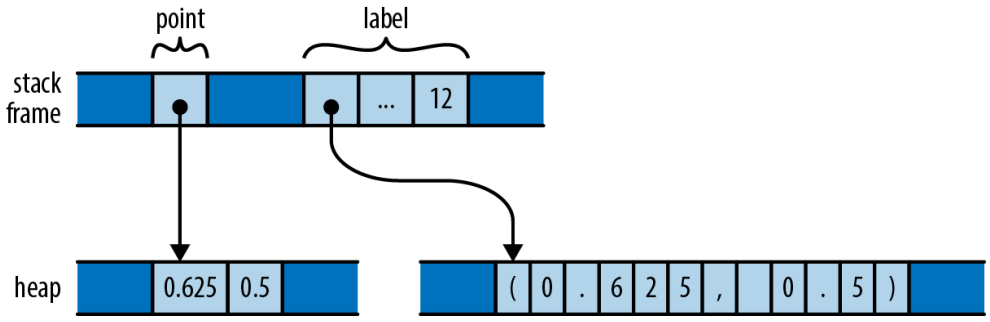

Figure 4-3. Deux variables locales, chacune possédant de la mémoire dans le tas

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

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, `composers` est un `Vec<Person>`, un vecteur de structures, chacune contenant une chaîne et un nombre. En mémoire, la valeur finale de `composers` ressemble à la Figure 4-4 .

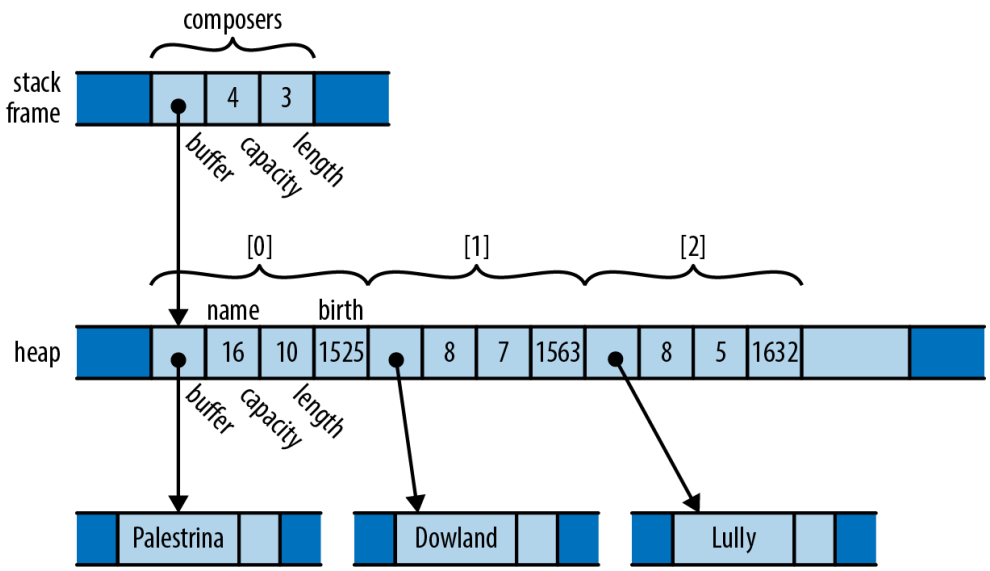

Illustration 4-4. Un arbre de propriété plus complexe

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

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 propriétaire unique, ce qui permet de décider facilement quand la supprimer. Mais une seule valeur peut posséder plusieurs autres valeurs : par exemple, le vecteur `composers` possède tous ses éléments. Et ces valeurs peuvent posséder d'autres valeurs à leur tour : chaque élément de `composers` possède une chaîne, qui possède son texte.

Il s'ensuit que les propriétaires et leurs valeurs possédées 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, l'arborescence entière l'accompagne. Nous pouvons voir un tel arbre de propriété dans le diagramme pour `composers` : ce n'est pas un "arbre" au sens d'une structure de données d'arbre 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, la règle du propriétaire unique de Rust interdisant tout regroupement 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.

Les programmes Rust ne suppriment généralement pas explicitement les valeursdu tout, de la même manière que les programmes C et C++ utiliseraient `free` et `delete` . 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 du genre. À ce stade, Rust s'assure que la valeur est correctement supprimée, ainsi que tout ce qu'elle possède.

Dans un certain sens, Rust est moins puissant que les 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 justement 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 exactement parce que les relations qu'il peut rencontrer dans votre code sont plus traitables. 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 s'y prend pour résoudre un problème pour s'assurer qu'au moins quelques solutions parfaitement fines respectent les restrictions imposées par le langage. impose.

Cela dit, le concept de propriété tel que nous l'avons expliqué jusqu'ici 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. Ceux-ci sont appelés `Copy` types.
- La bibliothèque standard fournit les types de pointeurs à référence comptée `Rc` et `Arc` , qui permettent aux valeurs d'avoir plusieurs propriétaires, sous certaines restrictions.
- 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 à son tour, avec des références couvertes dans le chapitre suivant.

Se déplace

En rouille, pour la plupart des types, les opérations telles que l'affectation d'une valeur à une variable, sa transmission à une fonction ou son retour à partir d'une fonction ne copient pas la valeur : elles la *déplacent* . La source abandonne 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 Rust construisent et détruisent des structures complexes une valeur à la fois, un mouvement à la fois.

Vous pourriez être surpris que Rust change le sens de telles opérations fondamentales ; l'affectation est sûrement quelque chose qui devrait être assez bien défini à ce stade de l'histoire. Cependant, si vous regardez attentivement la façon dont les différentes langues ont choisi de gérer les devoirs, vous verrez qu'il existe en fait des variations significatives d'une école à l'autre. La comparaison permet également de mieux comprendre la signification et les conséquences du choix de Rust.

Considérez le code Python suivant :

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

Chaque PythonL'objet contient un décompte de références, qui suit le nombre de valeurs qui s'y réfèrent actuellement. Ainsi, après l'affectation à `s`, l'état du programme ressemble à la Figure 4-5 (notez que certains champs sont omis).

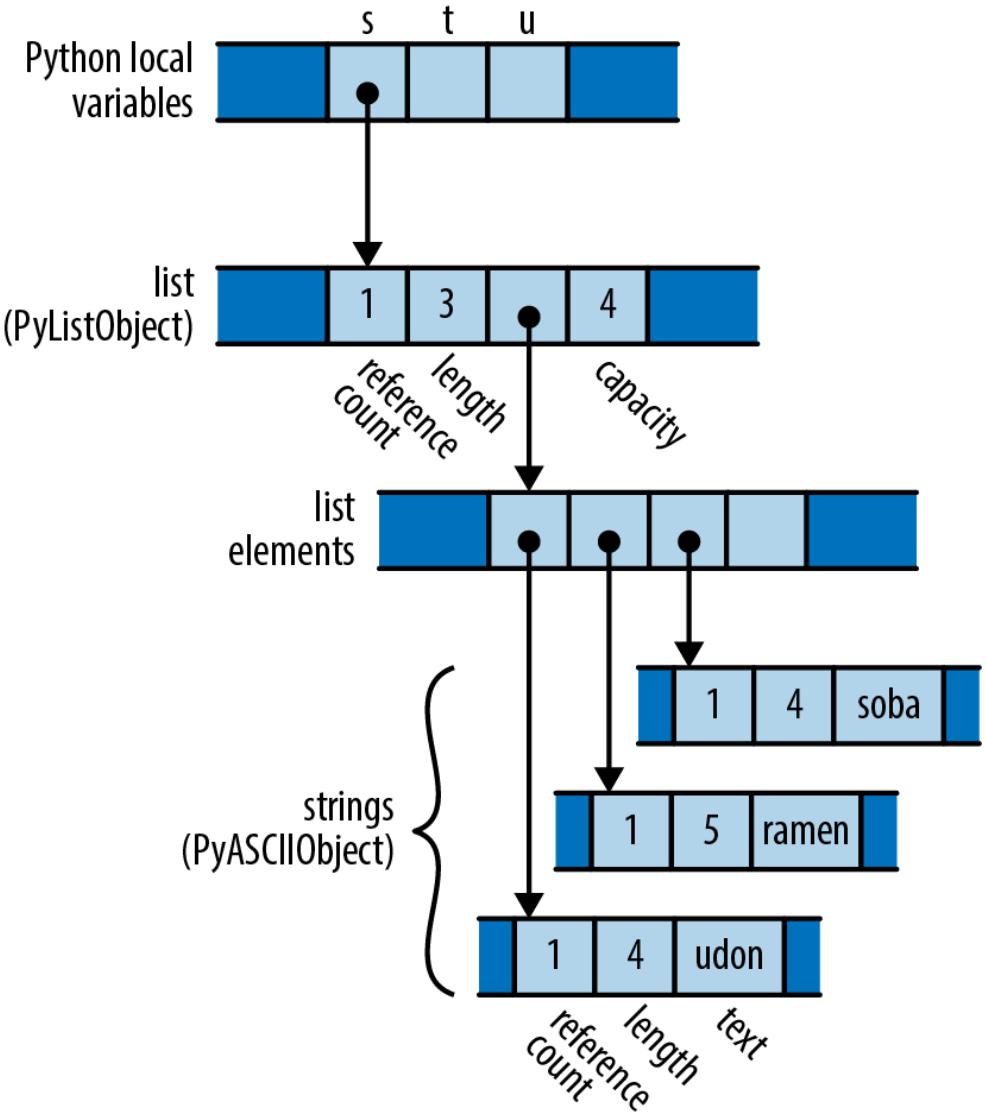

Figure 4-5. Comment Python représente une liste de chaînes en mémoire

Puisque only `s` pointe vers la liste, le nombre de références de la liste est 1 ; et puisque la liste est le seul objet pointant vers les chaînes, chacune de leurs décomptes de références vaut également 1.

Que se passe-t-il lorsque le programme exécute les affectations à `t` et `u` ? Python implémente l'affectation simplement en faisant pointer la destination sur le même objet que la source et en incrémentant le nombre de références de l'objet. L'état final du programme ressemble donc à la figure 4-6 .

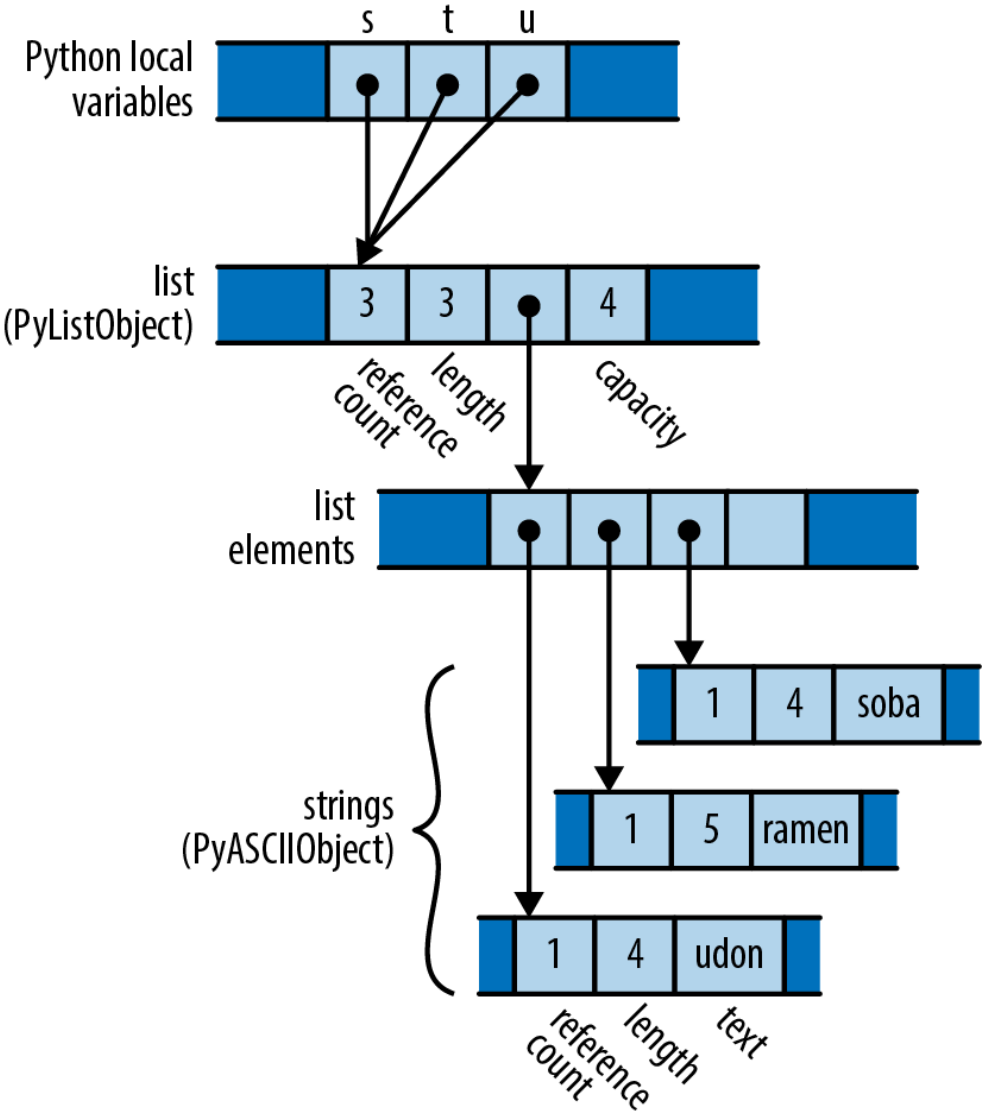

Figure 4-6. Le résultat de l'affectation `s` à la fois `t` et `u` en Python

Python a copié le pointeur de `s` dans `t` et `u` et a 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.

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 `s` ressemble à la Figure 4-7 en mémoire.

Que se passe-t-il lorsque le programme affecte `s` à `t` et `u` ? L'affectation de a `std::vector` produit une copie du vecteur en C++ ; `std::string` se comporte de manière similaire. Ainsi, au moment où le programme atteint la fin de ce code, il a en fait alloué trois vecteurs et neuf chaînes (Figure 4-8).

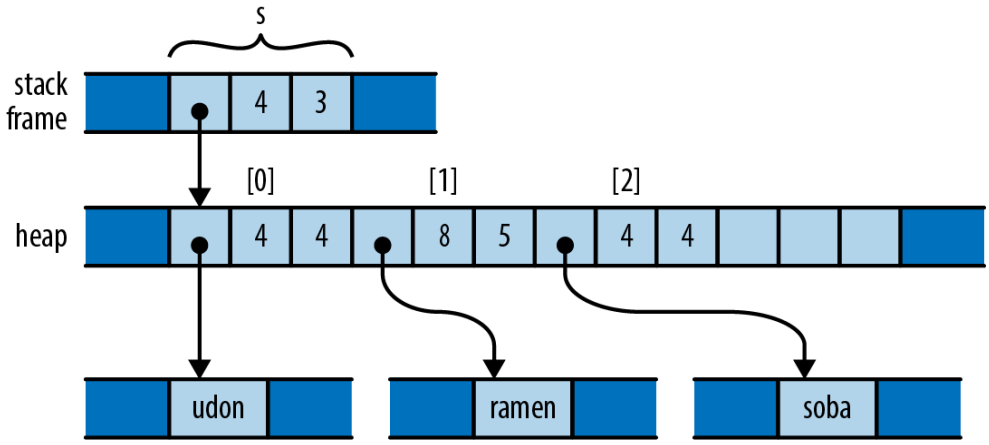

Illustration 4-7. Comment C++ représente un vecteur de chaînes en mémoire

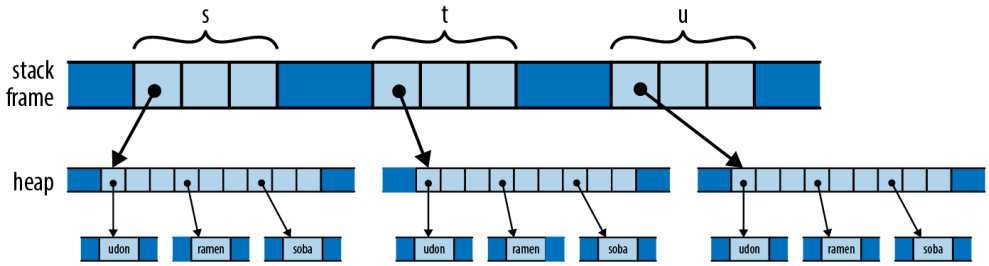

Illustration 4-8. Le résultat de l'affectation `s` à la fois `t` et `u` en C++

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 de la portée, tout ce qui est alloué ici est automatiquement nettoyé.

Dans un sens, C++ et Python ont choisi des compromis opposés : Python rend l'affectation bon marché, au prix d'un comptage de références (et dans le cas général, d'un ramasse-miettes). C++ conserve clairement la propriété de toute la mémoire, au détriment de l'attribution d'une copie complète de l'objet. Les programmeurs C++ sont souvent peu enthousiasmés par ce choix : les copies complètes 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 `"udon"` 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 `to_string` allouées au tas . `String`

Après avoir effectué l'initialisation de `s` , étant donné que Rust et C++ utilisent des représentations similaires pour les vecteurs et les chaînes, la si-

tuation ressemble exactement à ce qu'elle était en C++ (Figure 4-9).

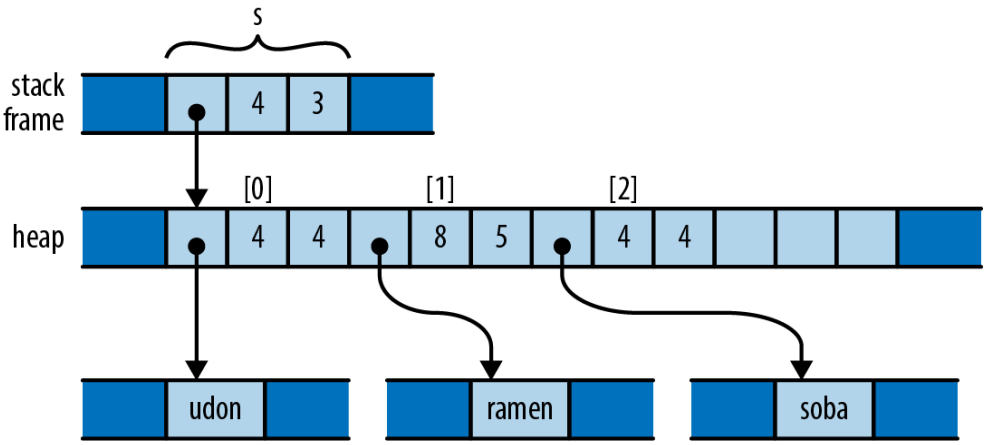

Illustration 4-9. Comment Rust représente un vecteur de chaînes en mémoire

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 `t`, la mémoire du programme ressemble à la Figure 4-10 .

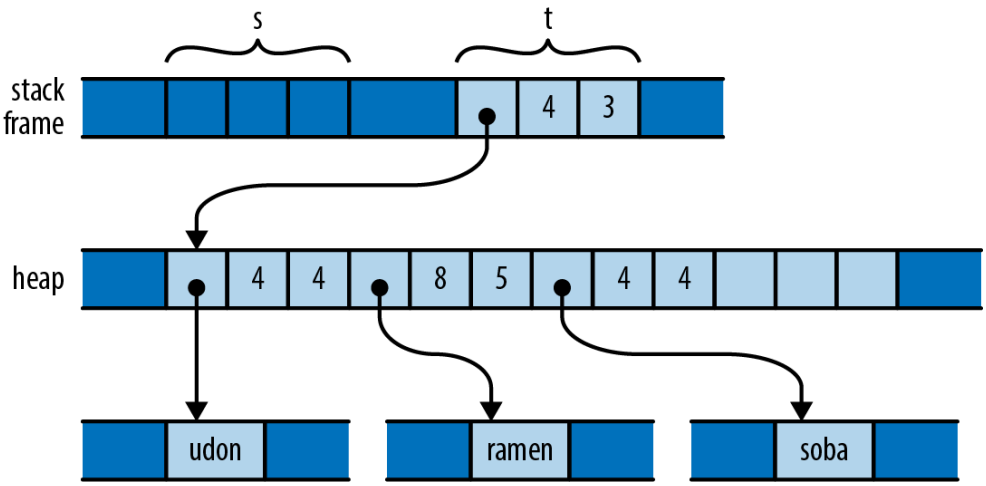

Figure 4-10. Le résultat de l'affectation `s` à `t` dans Rust

Que s'est-il passé ici ? L'initialisation `let t = s;` a déplacé les trois champs d'en-tête du vecteur de `s` vers `t` ; possède maintenant `t` le vecteur. Les éléments du vecteur sont restés là où ils étaient et rien n'est arrivé aux chaînes non plus. Chaque valeur a toujours un propriétaire unique, même si l'un d'entre eux a changé de mains. Il n'y avait pas de comptages de référence à ajuster. Et le compilateur considère maintenant `s` non initialisé.

Alors que se passe-t-il lorsque nous atteignons l'initialisation `let u = s;` ? Cela affecterait la valeur non initialisée `s` à `u`. Rust interdit prudemment l'utilisation de valeurs non initialisées, donc le compilateur rejette ce code avec l'erreur suivante :

```
error: use of moved value: `s`
  |
```

```
7 |     let s = vec!["udon".to_string(), "ramen".to_string(), "soba".to_stri
  |         - 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 mouvement 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 en C++, la propriété est toujours claire : le programme n'a pas besoin de comptage de références ou de récupération de place pour savoir quand libérer les éléments vectoriels et le contenu des chaînes.

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

```
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 : Propriété partagée » .

Plus d'opérations qui bougent

Dans les exemples jusqu'à présent, nous avons montré des initialisations, en fournissant des valeurs pour les variables lorsqu'elles entrent dans la portée d'une `let` instruction. Affectationà une variable est légèrement différente, en ce sens que si vous déplacez une valeur dans une variable qui a déjà été initialisée, Rust supprime la valeur précédente de la variable. Par exemple:

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

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

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

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

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. Passer des arguments aux fonctions déplace la propriété des paramètres de la fonction ; renvoyer une valeur à partir d'une fonction déplace la propriété vers l'appelant. Construire un tuple déplace les valeurs dans le tuple. Etc.

Vous avez peut-être maintenant une meilleure idée 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 montre plusieurs endroits où se produisent les déplacements, au-delà de l'initialisation et de l'affectation :

Renvoyer des valeurs d'une fonction

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

Construire de nouvelles valeurs

Le `name` champde la nouvelle `Person` structure est initialisée avec la valeur de retour de `to_string`. La structure s'approprie la chaîne.

Passer des valeurs à une fonction

Lala structure entière `Person`, pas un pointeur vers elle, est passée à la méthode du vecteur `push`, qui la déplace à la fin de la structure. Le vecteur s'approprie le `Person` et devient ainsi également le propriétaire indirect du nom `String`.

Valeurs mobilesautour comme cela peut sembler inefficace, mais il y a deux choses à garder à l'esprit. Tout d'abord, les déplacements s'appliquent toujours à la valeur proprement dite, et non au stockage de tas qu'ils possèdent. Pour les vecteurs et les chaînes, la *valeur propre* est l'entête de trois mots seul ; les tableaux d'éléments potentiellement volumineux et les tampons de texte restent là 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.

Mouvements et flux de contrôle

La précédenteles exemples 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 éloignée et qu'elle n'ait pas définitivement reçu une nouvelle valeur depuis, elle est considérée comme non initialisée. Par exemple, si une variable a toujours une valeur après avoir évalué `if` la condition d'une expression, nous pouvons l'utiliser dans les deux branches :

```
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, le déplacement d'une variable dans une boucle est interdit :

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

Autrement dit, à moins que nous ne lui ayons définitivement donné une nouvelle valeur à 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 mentionnéqu'un mouvement 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 souvenir que les troisième et cinquième éléments du vecteur sont devenus non initialisés et suivre ces informations jusqu'à ce que le vecteur soit supprimé. Dans le cas le plus général, les vecteurs devraient transporter des informations supplémentaires avec eux pour indiquer quels éléments sont actifs et lesquels ne sont plus initialisés. Ce n'est clairement pas le bon comportement pour un langage de programmation système ; un vecteur ne devrait 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 formule également une plainte similaire concernant le passage à `fifth`. Dans le message d'erreur, Rust suggère d'utiliser une référence, au cas où vous souhaiteriez 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 :

```
// 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 rempli, voire plus petit.

Les types de collection comme `Vec` proposent aussi généralement des méthodes pour consommer tous leurs éléments en boucle :

```
    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 `for ... in v`, cela *déplace* le vecteur hors de `v`, le laissant `v` non initialisé. La `for` machinerie interne de la boucle s'approprie le vecteur et le dissèque en ses éléments. A chaque itération, la boucle déplace un autre élément vers la variable `s`. Puisque `s` maintenant possède la chaîne, nous pouvons la modifier dans le corps de la boucle avant de l'imprimer. Et puisque le vecteur lui-même n'est plus visible pour le code, rien ne peut l'observer au milieu de la boucle dans un état partiellement vidé.

Si vous avez besoin de déplacer une valeur hors 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 ceci :

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

Cela provoquera simplement la même erreur "Impossible de sortir de l'index" indiquée précédemment. Mais parce que vous avez changé le type du `name` champ de `String` à `Option<String>` , cela signifie qu'il `None` s'agit d'une valeur légitime pour le champ, donc cela fonctionne :

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

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

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

Types de copie : l'exception aux déménagements

Les exemplesnous avons montré jusqu'à présent que les valeurs déplacées impliquent des vecteurs, des chaînes et d'autres types qui pourraient potentiellement utiliser beaucoup de mémoire et être coûteux à copier. Les mouvements 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 a `String` avec ce qui se passe lorsque nous attribuons une `i32` valeur :

```
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 .

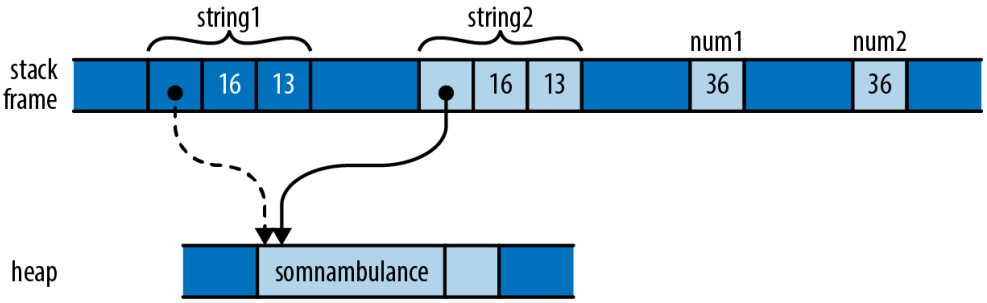

Figure 4-11. L'attribution d'un `String` déplace la valeur, tandis que l'attribution d'un `i32` la copie

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

Le déplacement d'une valeur laisse la source du déplacement non initialisée. Mais alors qu'il sert un objectif essentiel de traiter `string1` comme sans valeur, traiter de `num1` 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.

Précédemment, nous avons pris soin de dire que *la plupart des* types sont déplacés ; nous en sommes maintenant aux exceptions, les types que Rust désigne comme *`Copy` types* . L'attribution d'une valeur d'un `Copy` type copie la valeur, plutôt que de la déplacer. La source de l'affectation reste initialisée et utilisable, avec la même valeur qu'elle avait auparavant. Passer `Copy` des types aux fonctions et aux constructeurs se comporte de la même manière.

Les `Copy` types standard incluent tous les types entiers et numériques à virgule flottante de la machine, les types `char` et `bool` et quelques

autres. Un tuple ou un tableau de `Copy` types de taille fixe est lui-même un `Copy` type.

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

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

Qu'en est-il des types que vous définissez vous-même ? Par défaut, `struct` et `enum` les types ne sont pas `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; La rouille 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 `Label` n'est pas `Copy` , le transmettre à a `print` transféré la propriété de la valeur à la `print` fonction, qui l'a ensuite supprimée avant

de revenir. Mais c'est idiot; a `Label` n'est rien d'autre qu'un `u32` avec des prétentions. Il n'y a aucune raison de passer `l` à `print` devrait déplacer la valeur.

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

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

Avec ce changement, le code précédent se compile sans problème. Cependant, si nous essayons cela sur un type dont les champs ne sont pas all `Copy`, cela ne fonctionne pas. Supposons que nous compilions le code suivant :

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

Il provoque 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 `Copy`, en supposant qu'ils sont éligibles ? Qu'un type soit `Copy` ou non a un effet important sur la façon dont le code est autorisé à l'utiliser : `Copy` 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, le contraire est vrai : `Copy` les types sont très limités dans les types qu'ils peuvent contenir, tandis que les non `Copy` -types peuvent utiliser l'allocation de tas et posséder d'autres types de ressources. Ainsi, la création d'un type `Copy` représente un engagement sérieux de la part de l'implémenteur : s'il est nécessaire de le changer pour une version non `Copy` ultérieure, une grande partie du code qui l'utilise devra probablement être adaptée.

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 mouve-

ment est une copie superficielle octet par octet qui laisse la source non initialisée. Les copies sont identiques, 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 moins prévisibles les opérations de base telles que l'affectation, le passage de paramètres et le retour de valeurs à partir de fonctions. Par exemple, plus tôt dans ce chapitre, nous avons montré comment l'assignation 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 évidents pour le programmeur. Les opérations de base doivent rester simples. Les opérations potentiellement coûteuses doivent être explicites, comme les appels à `clone` dans l'exemple précédent qui font des copies complètes des vecteurs et des chaînes qu'ils contiennent.

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

Rc et Arc : Propriété partagée

Bien quela plupart des valeurs ont 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 le pointeur compté par référencetypes `Rc` et `Arc` . 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 tomber sur l'un des autres types de problèmes qui accompagnent la référence. compter les types de pointeurs en C++.

Les types `Rc` et sont très similaires ; `Arc` la seule différence entre eux est qu'un `Arc` peut être partagé directement entre les threads en toute sécurité - le nom `Arc` est l'abréviation de *décompte de références atomiques* - alors qu'un simple `Rc` utilise un code non sécurisé pour les threads plus rapide pour mettre à jour son décompte 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 `Arc`, vous devez donc utiliser `Rc` ; La rouille vous empêchera d'en passer accidentellement un à travers une limite de fil. Les deux types sont par ailleurs équivalents, donc pour le reste de cette section, nous ne parlerons que de `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 `Rc` pour obtenir un effet similaire dans Rust. Considérez le code suivant :

```
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 `T`, une `Rc<T>` valeur est un pointeur vers un tas alloué `T` auquel un nombre de références a été attaché. Le clonage d'une `Rc<T>` valeur ne copie pas le `T` ; à la place, 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.

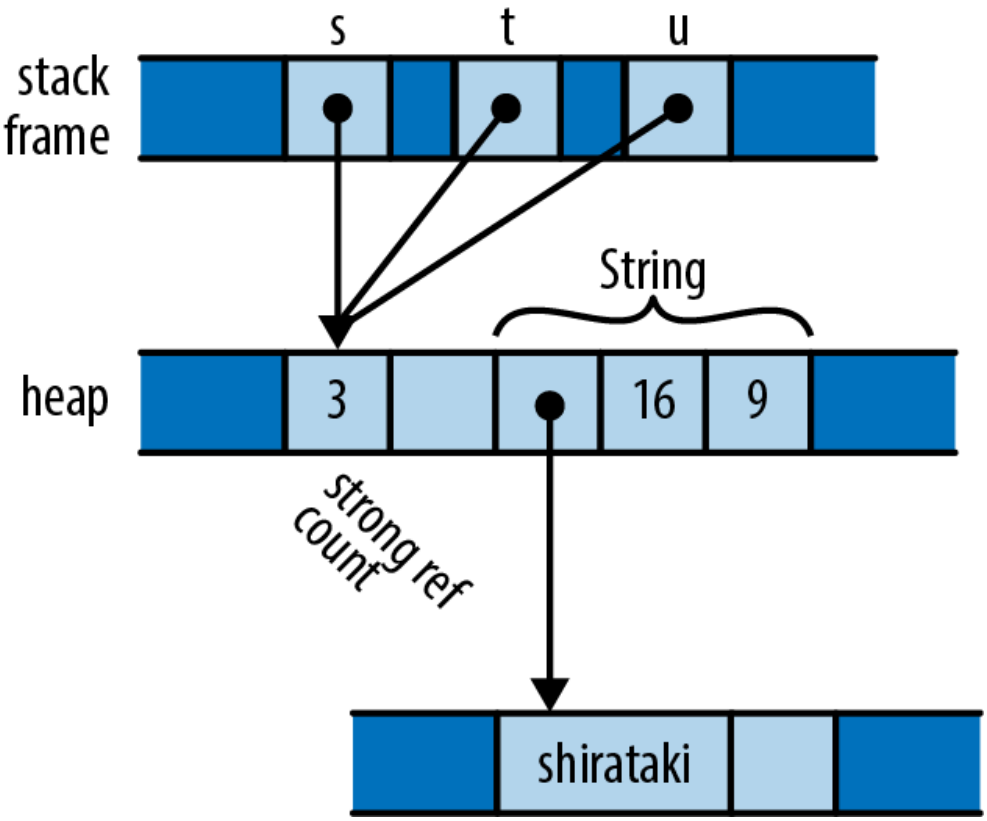

Illustration 4-12. Une chaîne comptée par référence avec trois références

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

Vous pouvez utiliser n'importe laquelle des `String` méthodes habituelles de directement sur un `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 détenue par un `Rc` pointeur est immuable. Supposons que vous essayez d'ajouter du texte à la fin de la chaîne :

```
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 des threads de Rust dépendent de la garantie qu'aucune valeur n'est simultanément partagée et modifiable. Rust suppose que le référent d'un `Rc` pointeur peut en général être partagé, il ne doit donc pas être modifiable. Nous expliquons pourquoi cette restriction est importante au chapitre 5 .

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

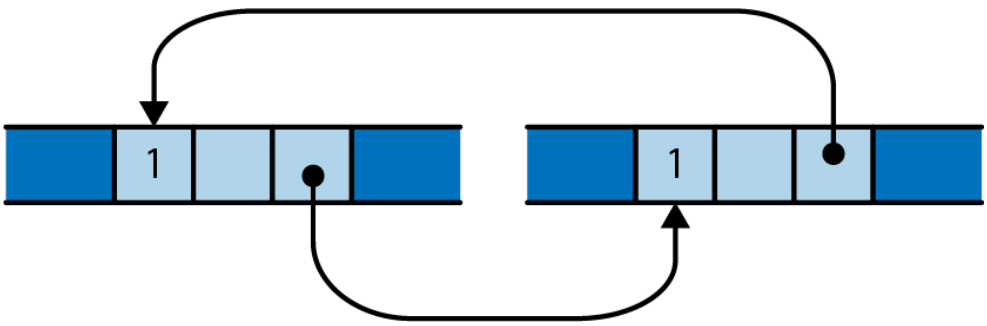

Figure 4-13. Une boucle de comptage de références ; ces objets ne seront pas libérés

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 la valeur la plus ancienne

soit modifiable. Puisque `Rc` les pointeurs maintiennent leurs référents immuables, il n'est normalement pas possible de créer un cycle. Cependant, Rust fournit des moyens de créer des portions mutables de valeurs autrement immuables ; c'est ce qu'on appelle *la mutabilité intérieure*, et nous le couvrons dans « Mutabilité intérieure » . Si vous combinez ces techniques avec des `Rc` pointeurs, vous pouvez créer un cycle et une fuite de mémoire.

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

Les déplacements et les pointeurs comptés par référence sont deux façons d'assouplir 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 serez 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 atouts uniques de Rust..