Chapitre 22. Code dangereux

> *Que personne ne pense de moi que je suis humble ou faible ou passif ;*
> *Qu'ils comprennent que je suis d'un genre différent :*
> *dangereux pour mes ennemis, loyal envers mes amis.*
> *A une telle vie appartient la gloire.*
>
> —Euripide, *Médée*

Le secretLa joie de la programmation système est que, sous chaque langage sûr et chaque abstraction soigneusement conçue, se trouve un maelström tourbillonnant de langage machine extrêmement dangereux et de petits bricolages. Vous pouvez aussi écrire cela dans Rust.

Le langage que nous avons présenté jusqu'à présent dans le livre garantit que vos programmes sont exempts d'erreurs de mémoire et de courses de données de manière entièrement automatique, via des types, des durées de vie, des vérifications de limites, etc. Mais ce type de raisonnement automatisé a ses limites ; il existe de nombreuses techniques précieuses que Rust ne peut pas reconnaître comme sûres.

Le code non sécurisé vous permet de dire à Rust : "Je choisis d'utiliser des fonctionnalités dont vous ne pouvez pas garantir la sécurité". En marquant un bloc ou une fonction comme non sécurisé, vous acquérez la capacité d'appeler `unsafe` des fonctions dans la bibliothèque standard, de déréférencer des pointeurs non sécurisés et d'appeler des fonctions écrites dans d'autres langages comme C et C++, entre autres pouvoirs. Les autres vérifications de sécurité de Rust s'appliquent toujours : les vérifications de type, les vérifications de durée de vie et les vérifications de limites sur les index se produisent toutes normalement. Le code non sécurisé active simplement un petit ensemble de fonctionnalités supplémentaires.

Cette capacité à sortir des limites de Rust sûr est ce qui permet d'implémenter bon nombre des fonctionnalités les plus fondamentales de Rust dans Rust lui-même, tout comme C et C++ sont utilisés pour implémenter leurs propres bibliothèques standard. Le code non sécurisé est ce qui permet au `Vec` type de gérer efficacement son tampon ; le `std::io` module pour parler au système d'exploitation ; et les modules `std::thread` et pour fournir des primitives `std::sync` de concurrence .

Ce chapitre couvre les éléments essentiels de l'utilisation de fonctions non sécurisées :

- `unsafe` Blocs de rouilleétablir la frontière entre le code Rust ordinaire et sûr et le code qui utilise des fonctionnalités non sûres.
- Vous pouvez marquer des fonctionscomme `unsafe` , alertant les appelants de la présence de contrats supplémentaires qu'ils doivent suivre pour éviter un comportement indéfini.
- Pointeurs brutset leurs méthodes permettent un accès illimité à la mémoire et vous permettent de créer des structures de données que le système de type de Rust interdirait autrement. Alors que les références de Rust sont sûres mais contraintes, les pointeurs bruts, comme tout programmeur C ou C++ le sait, sont un outil puissant et pointu.
- Comprendre la définition d'un comportement indéfini vous aidera à comprendre pourquoi il peut avoir des conséquences bien plus graves que le simple fait d'obtenir des résultats incorrects.
- Les traits non sûrs, analogues aux `unsafe` fonctions, imposent un contrat que chaque implémentation (plutôt que chaque appelant) doit suivre.

Pas à l'abri de quoi ?

Au début de ce livre, nous avons montré un programme C qui plante de manière surprenante parce qu'il ne respecte pas l'une des règles prescrites par le standard C. Vous pouvez faire la même chose dans Rust :

```
$cat crash.rs
 fn main() {
    let mut a: usize = 0;
    let ptr = &mut a as *mut usize;
    unsafe {
        *ptr.offset(3) = 0x7ffff72f484c;
    }
}
$cargo build
    Compiling unsafe-samples v0.1.0
    Finished debug [unoptimized + debuginfo] target(s) in 0.44s
$../../target/debug/crash
crash: Error: .netrc file is readable by others.
crash: Remove password or make file unreadable by others.
Segmentation fault (core dumped)
$
```

Ce programme emprunte une référence mutable à la variable locale `a`, la convertit en un pointeur brut de type `*mut usize`, puis utilise la `offset` méthodepour produire un pointeur trois mots plus loin dans la mémoire. Il se trouve que c'est là `main` que l'adresse de retour de est stockée. Le programme remplace l'adresse de retour par une constante, de sorte que le retour de `main` se comporte de manière surprenante. Ce qui rend ce plantage possible, c'est l'utilisation incorrecte par le programme de fonctionnalités non sécurisées, dans ce cas, la possibilité de déréférencer les pointeurs bruts.

Une fonctionnalité non sécurisée est une fonctionnalité qui impose un *contrat*: règles que Rust ne peut pas appliquer automatiquement, mais que vous devez néanmoins suivre pour éviter *un comportement indéfini* .

Un contrat va au-delà des contrôles de type habituels et des contrôles de durée de vie, imposant des règles supplémentaires spécifiques à cette fonctionnalité dangereuse. Typiquement, Rust lui-même n'est pas du tout au courant du contrat ; c'est juste expliqué dans la documentation de la fonctionnalité. Par exemple, le type pointeur brut a un contrat vous interdisant de déréférencerun pointeur qui a été avancé au-delà de la fin de son référent d'origine. L'expression `*ptr.offset(3) = ...` dans cet exemple rompt ce contrat. Mais, comme le montre la transcription, Rust compile le programme sans se plaindre : ses contrôles de sécurité ne détectent pas cette violation. Lorsque vous utilisez des fonctionnalités non sécurisées, vous, en tant que programmeur, êtes responsable de vérifier que votre code respecte leurs contrats.

De nombreuses fonctionnalités ont des règles que vous devez suivre pour les utiliser correctement, mais ces règles ne sont pas des contrats au sens où nous l'entendons ici, à moins que les conséquences possibles incluent un comportement indéfini. Indéfinile comportement est le comportement Rust suppose fermement que votre code ne pourrait jamais s'afficher. Par exemple, Rust suppose que vous n'écraserez pas l'adresse de retour d'un appel de fonction avec quelque chose d'autre. Un code qui passe les contrôles de sécurité habituels de Rust et est conforme aux contrats des fonctionnalités dangereuses qu'il utilise ne peut pas faire une telle chose. Étant donné que le programme viole le contrat de pointeur brut, son comportement n'est pas défini et il déraille.

Si votre code présente un comportement indéfini, vous avez rompu votre moitié de votre marché avec Rust, et Rust refuse de prédire les conséquences. Extraire des messages d'erreur non pertinents des profondeurs

des bibliothèques système et planter est une conséquence possible ; donner le contrôle de votre ordinateur à un attaquant en est une autre. Les effets peuvent varier d'une version de Rust à l'autre, sans avertissement. Parfois, cependant, un comportement indéfini n'a pas de conséquences visibles. Par exemple, si la `main` fonction ne revient jamais (peut-être appelle-t-elle `std::process::exit` pour terminer le programme plus tôt), alors l'adresse de retour corrompue n'aura probablement pas d'importance.

Vous ne pouvez utiliser que des fonctionnalités non sécurisées dans un `unsafe` bloc ou une `unsafe` fonction ; nous expliquerons les deux dans les sections qui suivent. Cela rend plus difficile l'utilisation de fonctionnalités dangereuses sans le savoir : en vous forçant à écrire un `unsafe` bloc ou une fonction, Rust s'assure que vous avez reconnu que votre code peut avoir des règles supplémentaires à suivre..

Blocs dangereux

Un `unsafe` blocressemble à un bloc Rust ordinaire précédé du mot-`unsafe` clé, à la différence que vous pouvez utiliser des fonctionnalités non sécurisées dans le bloc :

```
unsafe {
    String::from_utf8_unchecked(ascii)
}
```

Sans le mot- `unsafe` clé devant le bloc, Rust s'opposerait à l'utilisation de `from_utf8_unchecked` , qui est une `unsafe` fonction. Avec le `unsafe` bloc qui l'entoure, vous pouvez utiliser ce code n'importe où.

Comme un bloc Rust ordinaire, la valeur d'un `unsafe` bloc est celle de son expression finale, ou `()` s'il n'en a pas. L'appel à `String::from_utf8_unchecked` indiqué précédemment fournit la valeur du bloc.

Un `unsafe` bloc déverrouille cinq options supplémentaires pour vous :

- Vous pouvez appeler `unsafe` des fonctions. Chaque `unsafe` fonction doit spécifier son propre contrat, en fonction de son objet.
- Vous pouvez déréférencerpointeurs bruts. Le code sécurisé peut transmettre des pointeurs bruts, les comparer et les créer par conversion à

partir de références (ou même d'entiers), mais seul le code non sécurisé peut réellement les utiliser pour accéder à la mémoire. Nous couvrirons les pointeurs bruts en détail et expliquerons comment les utiliser en toute sécurité dans "Raw Pointers" .
- Vous pouvez accéder aux champsde `union` s, dont le compilateur ne peut pas être sûr qu'ils contiennent des modèles de bits valides pour leurs types respectifs.
- Vous pouvez accéder `static` variables mutables . Comme expliqué dans "Variables globales" , Rust ne peut pas être sûr que les threads utilisent des `static` variables mutables, leur contrat vous oblige donc à vous assurer que tous les accès sont correctement synchronisés.
- Vous pouvez accéder aux fonctions et aux variables déclarées via l'étranger de Rustinterface de fonction. Ceux-ci sont considérés `unsafe` même lorsqu'ils sont immuables, car ils sont visibles par du code écrit dans d'autres langages qui peuvent ne pas respecter les règles de sécurité de Rust.

Restreindre les fonctionnalités dangereuses aux `unsafe` blocs ne vous empêche pas vraiment de faire ce que vous voulez. Il est parfaitement possible de simplement coller un `unsafe` bloc dans votre code et de passer à autre chose. L'avantage de la règle réside principalement dans le fait d'attirer l'attention humaine sur du code dont Rust ne peut garantir la sécurité :

- Vous n'utiliserez pas accidentellement des fonctionnalités dangereuses et découvrirez ensuite que vous étiez responsable de contrats dont vous ignoriez même l'existence.
- Un `unsafe` bloc attire davantage l'attention des examinateurs. Certains projets ont même une automatisation pour garantir cela, en signalant les changements de code qui affectent les `unsafe` blocs pour une attention particulière.
- Lorsque vous envisagez d'écrire un `unsafe` bloc, vous pouvez prendre un moment pour vous demander si votre tâche nécessite vraiment de telles mesures. Si c'est pour les performances, avez-vous des mesures pour montrer qu'il s'agit en fait d'un goulot d'étranglement ? Peut-être existe-t-il un bon moyen d'accomplir la même chose dans Rust en toute sécurité.

Exemple : un type de chaîne ASCII efficace

Voici la définitionde `Ascii`, un type de chaîne qui garantit que son contenu est toujours en ASCII valide. Ce type utilise une fonctionnalité non sécurisée pour fournir une conversion sans coût en `String` :

```
mod my_ascii {
    /// An ASCII-encoded string.
    #[derive(Debug, Eq, PartialEq)]
    pub struct Ascii(
        // This must hold only well-formed ASCII text:
        // bytes from `0` to `0x7f`.
        Vec<u8>
    );

    impl Ascii {
        /// Create an `Ascii` from the ASCII text in `bytes`. Return a
        /// `NotAsciiError` error if `bytes` contains any non-ASCII
        /// characters.
        pub fn from_bytes(bytes: Vec<u8>) ->Result<Ascii, NotAsciiError>
            if bytes.iter().any(|&byte| !byte.is_ascii()) {
                return Err(NotAsciiError(bytes));
            }
            Ok(Ascii(bytes))
        }
    }

    // When conversion fails, we give back the vector we couldn't convert
    // This should implement `std::error::Error`; omitted for brevity.
    #[derive(Debug, Eq, PartialEq)]
    pub struct NotAsciiError(pub Vec<u8>);

    // Safe, efficient conversion, implemented using unsafe code.
    impl From<Ascii> for String {
        fn from(ascii: Ascii) -> String {
            // If this module has no bugs, this is safe, because
            // well-formed ASCII text is also well-formed UTF-8.
            unsafe { String::from_utf8_unchecked(ascii.0) }
        }
    }
    ...
}
```

La clé de ce module est la définition du `Ascii` type. Le type lui-même est marqué `pub`, pour le rendre visible en dehors du `my_ascii` module. Mais l' `Vec<u8>` élément du type n'est *pas* public, donc seul le `my_ascii` module peut construire une `Ascii` valeur ou faire référence à son élément. Cela laisse au code du module un contrôle total sur ce qui

peut ou non y apparaître. Tant que les constructeurs publics et les méthodes garantissent que les `Ascii` valeurs fraîchement créées sont bien formées et le restent tout au long de leur vie, alors le reste du programme ne peut pas violer cette règle. Et en effet, le constructeur public `Ascii::from_bytes` vérifie soigneusement le vecteur qui lui est donné avant d'accepter de construire un `Ascii` à partir de cela. Par souci de brièveté, nous ne montrons aucune méthode, mais vous pouvez imaginer un ensemble de méthodes de gestion de texte qui garantissent que les `Ascii` valeurs contiennent toujours le texte ASCII approprié, tout comme `String` les méthodes de a garantissent que son contenu reste UTF-8 bien formé.

Cette disposition nous permet de mettre `From<Ascii>` en œuvre `String` très efficacement. La fonction unsafe `String::from_utf8_unchecked` prend un vecteur d'octets et en construit un `String` sans vérifier si son contenu est du texte UTF-8 bien formé ; le contrat de la fonction tient son appelant responsable de cela. Heureusement, les règles imposées par le `Ascii` type sont exactement ce dont nous avons besoin pour satisfaire `from_utf8_unchecked` le contrat de . Comme nous l'avons expliqué dans "UTF-8" , tout bloc de texte ASCII est également UTF-8 bien formé, de sorte que le sous- `Ascii` jacent d' `Vec<u8>` un est immédiatement prêt à servir de `String` tampon pour un .

Avec ces définitions en place, vous pouvez écrire :

```
use my_ascii::Ascii;

let bytes:Vec<u8> = b"ASCII and ye shall receive".to_vec();

// This call entails no allocation or text copies, just a scan.
let ascii: Ascii = Ascii::from_bytes(bytes)
    .unwrap(); // We know these chosen bytes are ok.

// This call is zero-cost: no allocation, copies, or scans.
let string = String::from(ascii);

assert_eq!(string, "ASCII and ye shall receive");
```

Aucun `unsafe` bloc n'est requis pour utiliser `Ascii` . Nous avons implémenté une interface sécurisée utilisant des opérations non sécurisées et nous nous sommes arrangés pour respecter leurs contrats en fonction

uniquement du code propre au module, et non du comportement de ses utilisateurs.

An `Ascii` n'est rien de plus qu'un wrapper autour d'un `Vec<u8>`, caché à l'intérieur d'un module qui applique des règles supplémentaires sur son contenu. Un type de ce type est appelé un *nouveau type*, un modèle courant dans Rust. Le propre type de Rust `String` est défini exactement de la même manière, sauf que son contenu est limité à UTF-8, et non ASCII. En fait, voici la définition de `String` de la bibliothèque standard :

```
pub struct String {
    vec:Vec<u8>,
}
```

Au niveau de la machine, avec les types de Rust hors de l'image, un nouveau type et son élément ont des représentations identiques en mémoire, donc la construction d'un nouveau type ne nécessite aucune instruction machine. Dans `Ascii::from_bytes`, l'expression `Ascii(bytes)` considère simplement que la `Vec<u8>` représentation de contient maintenant une `Ascii` valeur. De même, `String::from_utf8_unchecked` ne nécessite probablement aucune instruction machine lorsqu'il est en ligne : le `Vec<u8>` est maintenant considéré comme un fichier `String`.

Fonctions dangereuses

Une `unsafe` fonctionLa définition ressemble à une définition de fonction ordinaire précédée du mot- `unsafe` clé. Le corps d'une `unsafe` fonction est automatiquement considéré comme un `unsafe` bloc.

Vous ne pouvez appeler `unsafe` des fonctions qu'à l'intérieur de `unsafe` blocs. Cela signifie que le marquage d'une fonction `unsafe` avertit ses appelants que la fonction a un contrat qu'ils doivent satisfaire pour éviter un comportement indéfini.

Par exemple, voici un nouveau constructeur pour le `Ascii` type que nous avons introduit précédemment qui construit un `Ascii` à partir d'un vecteur d'octets sans vérifier si son contenu est en ASCII valide :

```
// This must be placed inside the `my_ascii` module.
impl Ascii {
    /// Construct an `Ascii` value from `bytes`, without checking
```

```
        /// whether `bytes` actually contains well-formed ASCII.
        ///
        /// This constructor is infallible, and returns an `Ascii` directly,
        /// rather than a `Result<Ascii, NotAsciiError>` as the `from_bytes`
        /// constructor does.
        ///
        /// # Safety
        ///
        /// The caller must ensure that `bytes` contains only ASCII
        /// characters: bytes no greater than 0x7f. Otherwise, the effect is
        /// undefined.
        pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) ->Ascii {
            Ascii(bytes)
        }
    }
```

Vraisemblablement, l'appel de code `Ascii::from_bytes_unchecked` sait déjà d'une manière ou d'une autre que le vecteur en main ne contient que des caractères ASCII, de sorte que la vérification qui `Ascii::from_bytes` insiste sur l'exécution serait une perte de temps, et l'appelant devrait écrire du code pour gérer les `Err` résultats dont il sait qu'ils ne se produiront jamais. `Ascii::from_bytes_unchecked` permet à un tel appelant de contourner les vérifications et la gestion des erreurs.

Mais plus haut, nous avons souligné l'importance des `Ascii` constructeurs publics de et des méthodes garantissant que `Ascii` les valeurs sont bien formées. Ne manque-t-il pas `from_bytes_unchecked` à cette responsabilité ?

Pas tout à fait : `from_bytes_unchecked` remplit ses obligations en les répercutant sur son appelant via son contrat. La présence de ce contrat est ce qui rend correct le marquage de cette fonction `unsafe` : malgré le fait que la fonction elle-même n'effectue aucune opération dangereuse, ses appelants doivent suivre des règles que Rust ne peut pas appliquer automatiquement pour éviter un comportement indéfini.

Pouvez-vous vraiment provoquer un comportement indéfini en rompant le contratde `Ascii::from_bytes_unchecked` ? Oui. Vous pouvez construire un `String` holding UTF-8 mal formé comme suit :

```
// Imagine that this vector is the result of some complicated process
// that we expected to produce ASCII. Something went wrong!
let bytes = vec![0xf7, 0xbf, 0xbf, 0xbf];
```

```
let ascii = unsafe {
    // This unsafe function's contract is violated
    // when `bytes` holds non-ASCII bytes.
    Ascii::from_bytes_unchecked(bytes)
};

let bogus:String = ascii.into();

// `bogus` now holds ill-formed UTF-8. Parsing its first character produc
// a `char` that is not a valid Unicode code point. That's undefined
// behavior, so the language doesn't say how this assertion should behave
assert_eq!(bogus.chars().next().unwrap() as u32, 0x1fffff);
```

Dans certaines versions de Rust, sur certaines plates-formes, cette affirmation a échoué avec le message d'erreur divertissant suivant :

```
thread 'main' panicked at 'assertion failed: `(left == right)`
  left: `2097151`,
 right: `2097151`', src/main.rs:42:5
```

Ces deux nombres nous semblent égaux, mais ce n'est pas la faute de Rust ; c'est la faute du `unsafe` bloc précédent. Lorsque nous disons qu'un comportement indéfini conduit à des résultats imprévisibles, c'est le genre de chose que nous voulons dire.

Cela illustre deux faits critiques sur les bogueset code non sécurisé :

- *Les bugs qui se produisent avant le `unsafe` blocage peuvent rompre les contrats.* Le fait qu'un `unsafe` bloc provoque un comportement indéfini peut dépendre non seulement du code dans le bloc lui-même, mais également du code qui fournit les valeurs sur lesquelles il opère. Tout ce sur quoi votre `unsafe` code s'appuie pour satisfaire les contrats est critique pour la sécurité. La conversion de `Ascii` vers `String` basée sur `String::from_utf8_unchecked` n'est bien définie que si le reste du module maintient correctement `Ascii` les invariants de .
- *Les conséquences de la rupture d'un contrat peuvent apparaître après avoir quitté le `unsafe` bloc.* Le comportement indéfini courtisé par le non-respect du contrat d'une fonctionnalité dangereuse ne se produit souvent pas dans le `unsafe` bloc lui-même. Construire un faux `String` comme indiqué précédemment peut ne pas causer de problèmes jusqu'à bien plus tard dans l'exécution du programme.

Essentially, Rust's type checker, borrow checker, and other static checks are inspecting your program and trying to construct proof that it cannot exhibit undefined behavior. When Rust compiles your program successfully, that means it succeeded in proving your code sound. An `unsafe` block is a gap in this proof: "This code," you are saying to Rust, "is fine, trust me." Whether your claim is true could depend on any part of the program that influences what happens in the `unsafe` block, and the consequences of being wrong could appear anywhere influenced by the `unsafe` block. Writing the `unsafe` keyword amounts to a reminder that you are not getting the full benefit of the language's safety checks.

Si vous avez le choix, vous devriez naturellement préférer créer des interfaces sécurisées, sans contrats. Ceux-ci sont beaucoup plus faciles à utiliser, car les utilisateurs peuvent compter sur les contrôles de sécurité de Rust pour s'assurer que leur code est exempt de comportement indéfini. Même si votre implémentation utilise des fonctionnalités non sécurisées, il est préférable d'utiliser les types, les durées de vie et le système de modules de Rust pour respecter leurs contrats tout en utilisant uniquement ce que vous pouvez vous garantir, plutôt que de confier des responsabilités à vos appelants.

Malheureusement, il n'est pas rare de rencontrer des fonctions dangereuses dans la nature dont la documentation ne prend pas la peine d'expliquer leurs contrats. Vous êtes censé déduire les règles vous-même, en fonction de votre expérience et de votre connaissance du comportement du code. Si vous vous êtes déjà demandé avec inquiétude si ce que vous faites avec une API C ou C++ est OK, alors vous savez ce que c'est.

Blocage non sécurisé ou fonction non sécurisée ?

Vous pouvez vous demander s'il faut utiliser un `unsafe` blocou marquez simplement toute la fonction comme non sécurisée. L'approche que nous recommandons est de prendre d'abord une décision concernant la fonction :

- S'il est possible d'abuser de la fonction d'une manière qui se compile correctement mais provoque toujours un comportement indéfini, vous devez la marquer comme non sécurisée. Les règles d'utilisation correcte de la fonction sont son contrat ; l'existence d'un contrat est ce qui rend la fonction dangereuse.

- Sinon, la fonction est sûre : aucun appel correctement typé ne peut provoquer un comportement indéfini. Il ne doit pas être marqué `unsafe`.

Que la fonction utilise des fonctionnalités dangereuses dans son corps n'est pas pertinent ; ce qui compte, c'est la présence d'un contrat. Auparavant, nous avons montré une fonction non sécurisée qui n'utilise aucune fonctionnalité non sécurisée et une fonction sécurisée qui utilise des fonctionnalités non sécurisées.

Ne marquez pas une fonction sûre `unsafe` simplement parce que vous utilisez des fonctionnalités dangereuses dans son corps. Cela rend la fonction plus difficile à utiliser et déroute les lecteurs qui s'attendront (correctement) à trouver un contrat expliqué quelque part. Utilisez plutôt un `unsafe` bloc, même s'il s'agit du corps entier de la fonction.

Comportement indéfini

Dans l'introduction, nous avons dit que le terme *comportement indéfini* signifie "comportement que Rust suppose fermement que votre code ne pourrait jamais présenter". C'est une tournure de phrase étrange, d'autant plus que nous savons par notre expérience avec d'autres langues que ces comportements *se* produisent par accident avec une certaine fréquence. Pourquoi ce concept est-il utile pour définir les obligations d'un code non sécurisé ?

Un compilateur est un traducteur d'un langage de programmation à un autre. Le compilateur Rust prend un programme Rust et le traduit en un programme équivalent en langage machine. Mais qu'est-ce que cela veut dire de dire que deux programmes dans des langages aussi complètement différents sont équivalents ?

Heureusement, cette question est plus facile pour les programmeurs que pour les linguistes. Nous disons généralement que deux programmes sont équivalents s'ils auront toujours le même comportement visible lors de leur exécution : ils effectuent les mêmes appels système, interagissent avec des bibliothèques étrangères de manière équivalente, etc. C'est un peu comme un test de Turing pour les programmes : si vous ne savez pas si vous interagissez avec l'original ou la traduction, alors ils sont équivalents.

Considérez maintenant le code suivant :

```
let i = 10;
very_trustworthy(&i);
println!("{}", i * 100);
```

Même en ne sachant rien de la définition de `very_trustworthy`, nous pouvons voir qu'il ne reçoit qu'une référence partagée à `i`, donc l'appel ne peut pas changer `i` la valeur de. Puisque la valeur passée à `println!` sera toujours `1000`, Rust peut traduire ce code en langage machine comme si nous avions écrit :

```
very_trustworthy(&10);
println!("{}", 1000);
```

Cette version transformée a le même comportement visible que l'original, et elle est probablement un peu plus rapide. Mais il est logique de ne considérer les performances de cette version que si nous convenons qu'elle a la même signification que l'original. Et si `very_trustworthy` étaient définis comme suit ?

```
fn very_trustworthy(shared:&i32) {
    unsafe {
        // Turn the shared reference into a mutable pointer.
        // This is undefined behavior.
        let mutable = shared as *const i32 as *mut i32;
        *mutable = 20;
    }
}
```

Ce code enfreint les règles des références partagées : il change la valeur de `i` en `20`, même s'il devrait être gelé car il `i` est emprunté pour le partage. Par conséquent, la transformation que nous avons faite à l'appelant a maintenant un effet très visible : si Rust transforme le code, le programme imprime `1000` ; s'il laisse le code seul et utilise la nouvelle valeur de `i`, il imprime `2000`. Enfreindre les règles des références partagées dans `very_trustworthy` signifie que les références partagées ne se comporteront pas comme prévu dans ses appelants.

Ce genre de problème se pose avec presque tous les types de transformation que Rust pourrait tenter. Même l'intégration d'une fonction dans son site d'appel suppose, entre autres, que lorsque l'appelé a terminé, le flux de contrôle revient au site d'appel. Mais nous avons ouvert le chapitre avec un exemple de code mal élevé qui viole même cette hypothèse.

Il est fondamentalement impossible pour Rust (ou tout autre langage) d'évaluer si une transformation vers un programme conserve sa signification à moins qu'il ne puisse faire confiance aux caractéristiques fondamentales du langage pour qu'il se comporte comme prévu. Et qu'ils le fassent ou non peuvent dépendre non seulement du code à portée de main, mais aussi d'autres parties potentiellement éloignées du programme. Pour faire quoi que ce soit avec votre code, Rust doit supposer que le reste de votre programme se comporte bien.

Voici donc les règles de Rustpour les programmes bien élevés :

- Le programme ne doit pas lire la mémoire non initialisée.
- Le programme ne doit pas créer de valeurs primitives invalides :
  - Les références, cases ou `fn` pointeurs qui sont `null`
  - `bool` des valeurs qui ne sont ni a `0` ni `1`
  - `enum` valeurs avec des valeurs discriminantes non valides
  - `char` valeurs non valides, points de code Unicode non substituts
  - `str` valeurs qui ne sont pas bien formées UTF-8
  - Pointeurs gras avec vtables/longueurs de tranche non valides
  - Toute valeur du type "jamais", écrite `!` , pour les fonctions qui ne retournent pas
- Les règles de références expliquées au chapitre 5 doivent être respectées. Aucune référence ne peut survivre à son référent ; l'accès partagé est un accès en lecture seule ; et l'accès mutable est un accès exclusif.
- Le programme ne doit pas déréférencer les pointeurs nuls, incorrectement alignés ou pendants .
- Le programme ne doit pas utiliser un pointeur pour accéder à la mémoire en dehors de l'allocation à laquelle le pointeur est associé. Nous expliquerons cette règle en détail dans « Déréférencer les pointeurs bruts en toute sécurité » .
- Le programme doit être exempt de courses aux données. Une course aux données se produit lorsque deux threads accèdent au même emplacement mémoire sans synchronisation et qu'au moins l'un des accès est une écriture.
- Le programme ne doit pas se dérouler sur un appel effectué depuis une autre langue, via l'interface de la fonction étrangère, comme expliqué dans "Déroulement" .
- Le programme doit être conforme aux contrats des fonctions standard de la bibliothèque.

Comme nous ne disposons pas encore d'un modèle complet de la sémantique de Rust pour le `unsafe` code, cette liste évoluera probablement

avec le temps, mais ceux-ci resteront probablement interdits.

Toute violation de ces règles constitue un comportement indéfini et rend les efforts de Rust pour optimiser votre programme et le traduire en langage machine indignes de confiance. Si vous enfreignez la dernière règle et passez l'UTF-8 mal formé à `String::from_utf8_unchecked`, peut-être que 2097151 n'est pas si égal à 2097151 après tout.

Le code Rust qui n'utilise pas de fonctionnalités dangereuses est garanti de suivre toutes les règles précédentes, une fois qu'il est compilé (en supposant que le compilateur n'a pas de bogue ; nous y arrivons, mais la courbe ne croisera jamais l'asymptote). Ce n'est que lorsque vous utilisez des fonctionnalités non sécurisées que ces règles deviennent votre responsabilité.

En C et C++, le fait que votre programme se compile sans erreurs ni avertissements signifie beaucoup moins ; comme nous l'avons mentionné dans l'introduction de ce livre, même les meilleurs programmes C et C++ écrits par des projets très respectés qui maintiennent leur code à des normes élevées présentent un comportement indéfini dans la pratique.

Caractéristiques dangereuses

Un *trait dangereux*est un trait qui a un contratRust ne peut pas vérifier ou appliquer ce que les implémenteurs doivent satisfaire pour éviter un comportement indéfini. Pour implémenter une caractéristique non sécurisée, vous devez marquer l'implémentation comme non sécurisée. C'est à vous de comprendre le contrat du trait et de vous assurer que votre type le satisfait.

Une fonction qui limite ses variables de type avec un trait non sécurisé est généralement une fonction qui utilise elle-même des fonctionnalités non sécurisées et ne satisfait leurs contrats qu'en fonction du contrat du trait non sécurisé. Une implémentation incorrecte du trait pourrait amener une telle fonction à présenter un comportement indéfini.

`std::marker::Send` et `std::marker::Sync` sont les exemples classiques de traits dangereux. Ces traits ne définissent aucune méthode, ils sont donc simples à implémenter pour n'importe quel type que vous aimez. Mais ils ont des contrats : `Send` exige que les implémenteurs soient en sécurité pour passer à un autre thread, et `Sync` exige qu'ils soient en sécurité pour partager entre les threads via des références partagées. La

mise en œuvre `Send` pour un type inapproprié, par exemple, ne ferait `std::sync::Mutex` plus à l'abri des courses de données.

À titre d'exemple simple, la bibliothèque standard Rust incluait un trait non sécurisé, `core::nonzero::Zeroable`, pour les typesqui peuvent être initialisés en toute sécurité en mettant tous leurs octets à zéro. De toute évidence, la mise à zéro de a `usize` est correcte, mais la mise à zéro de a `&T` vous donne une référence nulle, ce qui provoquera un plantage s'il est déréférencé. Pour les types qui étaient `Zeroable`, certaines optimisations étaient possibles : vous pouviez en initialiser un tableau rapidement avec `std::ptr::write_bytes` (l'équivalent de Rust de `memset`) ou utiliser des appels du système d'exploitation qui allouent des pages mises à zéro. (`Zeroable` était instable et déplacé vers une utilisation interne uniquement dans la `num` caisse de Rust 1.26, mais c'est un bon exemple simple et concret.)

`Zeroable` était un trait marqueur typique, dépourvu de méthodes ou de types associés :

```
pub unsafe trait Zeroable {}
```

Les implémentations pour les types appropriés étaient tout aussi simples :

```
unsafe impl Zeroable for u8 {}
unsafe impl Zeroable for i32 {}
unsafe impl Zeroable for usize {}
// and so on for all the integer types
```

Avec ces définitions, on pourrait écrire une fonction qui alloue rapidement un vecteur d'une longueur donnée contenant un `Zeroable` type :

```
use core:: nonzero::Zeroable;

fn zeroed_vector<T>(len: usize) -> Vec<T>
    where T: Zeroable
{
    let mut vec = Vec:: with_capacity(len);
    unsafe {
        std:: ptr::write_bytes(vec.as_mut_ptr(), 0, len);
        vec.set_len(len);
    }
    vec
}
```

Cette fonction commence par créer un vide `Vec` avec la capacité requise, puis appelle `write_bytes` pour remplir le tampon inoccupé avec des zéros. (La `write_byte` fonction traite `len` comme un nombre d' `T` éléments, pas un nombre d'octets, donc cet appel remplit tout le tampon.) La `set_len` méthode d'un vecteur change sa longueur sans rien faire au tampon ; ceci n'est pas sûr, car vous devez vous assurer que l'espace tampon nouvellement inclus contient réellement des valeurs correctement initialisées de type `T` . Mais c'est exactement ce que la `T: Zeroable` limite établit : un bloc de zéro octet représente une `T` valeur valide. Notre utilisation de `set_len` était sûre.

Ici, nous l'utilisons :

```
let v:Vec<usize> = zeroed_vector(100_000);
assert!(v.iter().all(|&u| u == 0));
```

De toute évidence, `Zeroable` doit être un trait non sûr, car une implémentation qui ne respecte pas son contrat peut conduire à un comportement indéfini :

```
struct HoldsRef<'a>(&'a mut i32);

unsafe impl<'a> Zeroable for HoldsRef<'a> { }

let mut v:Vec<HoldsRef> = zeroed_vector(1);
*v[0].0 = 1;   // crashes: dereferences null pointer
```

Rust n'a aucune idée de ce qui `Zeroable` est censé signifier, il ne peut donc pas dire quand il est implémenté pour un type inapproprié. Comme pour toute autre fonctionnalité dangereuse, c'est à vous de comprendre et de respecter le contrat d'un trait dangereux.

Notez que le code non sécurisé ne doit pas dépendre de la mise en œuvre correcte de traits ordinaires et sûrs. Par exemple, supposons qu'il y ait une implémentation du `std::hash::Hasher` trait qui renvoie simplement une valeur de hachage aléatoire, sans relation avec les valeurs hachées. Le trait exige que le hachage deux fois des mêmes bits produise la même valeur de hachage, mais cette implémentation ne répond pas à cette exigence ; c'est tout simplement incorrect. Mais comme `Hasher` il ne s'agit pas d'un trait non sécurisé, le code non sécurisé ne doit pas présenter de comportement indéfini lorsqu'il utilise ce hachage. Le `std::collections::HashMap` type est écrit avec soin pour respecter les

contrats des fonctionnalités non sécurisées qu'il utilise, quel que soit le comportement du hacheur. Certes, la table ne fonctionnera pas correctement : les recherches échoueront et les entrées apparaîtront et disparaîtront au hasard. Mais la table ne présentera pas de comportement indéfini.

Pointeurs bruts

Un *pointeur brut*dans Rust est un pointeur sans contrainte. Vous pouvez utiliser des pointeurs bruts pour former toutes sortes de structures que les types de pointeurs vérifiés de Rust ne peuvent pas, comme des listes doublement liées ou des graphiques arbitraires d'objets. Mais parce que les pointeurs bruts sont si flexibles, Rust ne peut pas dire si vous les utilisez en toute sécurité ou non, vous ne pouvez donc les déréférencer que dans un `unsafe` bloc.

Les pointeurs bruts sont essentiellement équivalents aux pointeurs C ou C++, ils sont donc également utiles pour interagir avec du code écrit dans ces langages.

Il existe deux types de pointeurs bruts:

- A `*mut T` est un brutpointeur vers a `T` qui permet de modifier son référent.
- A `*const T` est un brutpointeur vers a `T` qui ne permet de lire que son référent.

(Il n'y a pas de type simple `*T` ; vous devez toujours spécifier soit `const` ou `mut` .)

Vous pouvez créer un pointeur brut par conversion à partir d'une référence, et le déréférencer avec l' `*` opérateur:

```
let mut x = 10;
let ptr_x = &mut x as *mut i32;

let y = Box::new(20);
let ptr_y = &*y as *const i32;

unsafe {
    *ptr_x += *ptr_y;
}
assert_eq!(x, 30);
```

Contrairement aux boîtes et aux références, les pointeurs brutspeut être nul, comme `NULL` en C ou `nullptr` en C++ :

```
fn option_to_raw<T>(opt: Option<&T>) -> *const T {
    match opt {
        None => std:: ptr::null(),
        Some(r) => r as *const T
    }
}

assert!(!option_to_raw(Some(&("pea", "pod"))).is_null());
assert_eq!(option_to_raw:: <i32>(None), std:: ptr::null());
```

Cet exemple n'a pas de `unsafe` blocs : créer des pointeurs bruts, les faire circuler et les comparer sont tous sûrs. Seul le déréférencement d'un pointeur brut n'est pas sûr.

Un pointeur brut vers un type non dimensionné est un pointeur gras, tout comme le serait la référence ou `Box` le type correspondant. Un `*const [u8]` pointeur inclut une longueur avec l'adresse, et un objet trait comme un `*mut dyn std::io::Write` pointeur porte une vtable.

Bien que Rust déréférence implicitement les types de pointeurs sûrs dans diverses situations, les déréférencements de pointeurs bruts doivent être explicites :

- L' `.` opérateur ne déréférencera pas implicitement un pointeur brut ; vous devez écrire `(*raw).field` ou `(*raw).method(...)`.
- Les pointeurs bruts ne s'implémentent pas `Deref`, donc déréférencez les coercionsne s'applique pas à eux.
- Les opérateurs aiment `==` et `<` comparentpointeurs bruts comme adresses : deux pointeurs bruts sont égaux s'ils pointent vers le même emplacement en mémoire. De même, le hachage d'un pointeur brut hache l'adresse vers laquelle il pointe, pas la valeur de son référent.
- Mise en pagetraits comme `std::fmt::Display` suivreréférences automatiquement, mais ne gèrent pas du tout les pointeurs bruts. Les exceptions sont `std::fmt::Debug` et `std::fmt::Pointer`, quiafficher les pointeurs brutssous forme d'adresses hexadécimales, sans les déréférencer.

Contrairement à l' `+` opérateur en C et C++, Rust `+` ne gère pas les pointeurs bruts, mais vous pouvez effectuer une arithmétique de pointeur via

leurs méthodes `offset` et `wrapping_offset` , ou les méthodes , , et , `add` plus pratiques. Inversement, la méthode donne la distance entre deux pointeurs en octets, bien que nous soyons responsables de nous assurer que le début et la fin sont dans la même région mémoire (le même , par exemple) : `sub wrapping_add wrapping_sub offset_from Vec`

```
let trucks = vec!["garbage truck", "dump truck", "moonstruck"];
let first: *const &str = &trucks[0];
let last:*const &str = &trucks[2];
assert_eq!(unsafe { last.offset_from(first) }, 2);
assert_eq!(unsafe { first.offset_from(last) }, -2);
```

Aucune conversion explicite n'est nécessaire pour `first` et `last` ; il suffit de spécifier le type. Rust contraint implicitement les références aux pointeurs bruts (mais pas l'inverse, bien sûr).

L' `as` opérateur permet presque toutes les conversions plausibles de références en pointeurs bruts ou entre deux types de pointeurs bruts. Cependant, vous devrez peut-être décomposer une conversion complexe en une série d'étapes plus simples. Par exemple:

```
&vec![42_u8] as *const String;  // error: invalid conversion
&vec![42_u8] as *const Vec<u8> as *const String;  // permitted
```

Notez que `as` cela ne convertira pas les pointeurs bruts en références. De telles conversions seraient dangereuses et `as` devraient rester une opération sûre. Au lieu de cela, vous devez déréférencer le pointeur brut (dans un `unsafe` bloc) puis emprunter la valeur résultante.

Soyez très prudent lorsque vous faites cela : une référence ainsi produite a une durée de vie illimitée: il n'y a pas de limite à sa durée de vie, car le pointeur brut ne donne à Rust rien sur quoi fonder une telle décision. Dans « Une interface sécurisée pour libgit2 », plus loin dans ce chapitre, nous montrons plusieurs exemples de la façon de contraindre correctement les durées de vie.

De nombreux types ont `as_ptr` et `as_mut_ptr` méthodesqui renvoient un pointeur brut vers leur contenu. Par exemple, les tranches de tableau et les chaînes renvoient des pointeurs vers leurs premiers éléments, et certains itérateurs renvoient un pointeur vers l'élément suivant qu'ils produiront. Posséder des types de pointeurs tels que `Box` , `Rc` et `Arc` have `into_raw` et des `from_raw` fonctions qui convertissent vers et

à partir de pointeurs bruts. Certains des contrats de ces méthodes imposent des exigences surprenantes, alors vérifiez leur documentation avant de les utiliser.

Vous pouvez également construire des pointeurs bruts par conversion à partir d'entiers, bien que les seuls entiers auxquels vous pouvez faire confiance pour cela soient généralement ceux que vous avez obtenus à partir d'un pointeur en premier lieu. "Exemple : RefWithFlag" utilise des pointeurs bruts de cette façon.

Contrairement aux références, les pointeurs bruts ne sont ni `Send` ni `Sync`. Par conséquent, tout type qui inclut des pointeurs bruts n'implémente pas ces traits par défaut. Il n'y a rien d'intrinsèquement dangereux à envoyer ou à partager des pointeurs bruts entre les threads ; après tout, où qu'ils aillent, vous avez toujours besoin d'un `unsafe` bloc pour les déréférencer. Mais étant donné les rôles que jouent généralement les pointeurs bruts, les concepteurs de langage ont considéré que ce comportement était le comportement par défaut le plus utile. Nous avons déjà discuté de la façon de mettre en œuvre `Send` et `Sync` de vous-même dans "Unsafe Traits".

Déréférencer les pointeurs bruts en toute sécurité

IciVoici quelques règles de bon sens pour utiliser les pointeurs bruts en toute sécurité :

- Le déréférencement des pointeurs nuls ou des pointeurs pendants est un comportement indéfini, tout comme la référence à une mémoire non initialisée ou à des valeurs qui sont sorties de la portée.
- Le déréférencement des pointeurs qui ne sont pas correctement alignés pour leur type de référent est un comportement indéfini.
- Vous ne pouvez emprunter des valeurs à un pointeur brut déréférencé que si cela respecte les règles de sécurité des références expliquées au chapitre 5 : aucune référence ne peut survivre à son référent, l'accès partagé est un accès en lecture seule et l'accès mutable est un accès exclusif. (Cette règle est facile à violer par accident, car les pointeurs bruts sont souvent utilisés pour créer des structures de données avec un partage ou une propriété non standard.)
- Vous ne pouvez utiliser le référent d'un pointeur brut que s'il s'agit d'une valeur bien formée de son type. Par exemple, vous devez vous assurer que le déréférencement de a `*const char` génère un point de code Unicode approprié et non de substitution.

- Vous pouvez utiliser les méthodes `offset` et `wrapping_offset` sur les pointeurs bruts uniquement pour pointer vers des octets dans la variable ou le bloc de mémoire alloué par tas auquel le pointeur d'origine faisait référence, ou vers le premier octet au-delà d'une telle région.
  Si vous faites de l'arithmétique de pointeur en convertissant le pointeur en entier, en faisant de l'arithmétique sur l'entier, puis en le reconvertissant en pointeur, le résultat doit être un pointeur que les règles de la `offset` méthode vous auraient permis de produire.
- Si vous affectez au référent d'un pointeur brut, vous ne devez pas violer les invariants de tout type dont le référent fait partie. Par exemple, si vous avez a `*mut u8` pointant vers un octet de a `String`, vous ne pouvez stocker que des valeurs dans ce `u8` qui laisse le `String` maintien UTF-8 bien formé.

La règle d'emprunt mise à part, ce sont essentiellement les mêmes règles que vous devez suivre lorsque vous utilisez des pointeurs en C ou C++.

La raison pour ne pas violer les invariants des types doit être claire. De nombreux types standard de Rust utilisent du code non sécurisé dans leur implémentation, mais fournissent toujours des interfaces sûres en supposant que les contrôles de sécurité, le système de modules et les règles de visibilité de Rust seront respectés. L'utilisation de pointeurs bruts pour contourner ces mesures de protection peut entraîner un comportement indéfini.

Le contrat complet et exact pour les pointeurs bruts n'est pas facile à énoncer et peut changer à mesure que le langage évolue. Mais les principes décrits ici devraient vous garder en territoire sûr.

Exemple : RefWithFlag

Voici un exemplede la façon de prendre un hack classique au niveau [1] bit rendu possible par des pointeurs bruts et de le transformer en un type Rust complètement sûr. Ce module définit un type, `RefWithFlag<'a, T>`, qui contient à la fois a `&'a T` et a `bool`, comme le tuple `(&'a T, bool)` et parvient toujours à n'occuper qu'un seul mot machine au lieu de deux. Ce type de technique est régulièrement utilisé dans les ramasse-miettes et les machines virtuelles, où certains types (par exemple, le type représentant un objet) sont si nombreux que l'ajout d'un seul mot à chaque valeur augmenterait considérablement l'utilisation de la mémoire :

```rust
mod ref_with_flag {
    use std:: marker:: PhantomData;
    use std:: mem::align_of;

    /// A `&T` and a `bool`, wrapped up in a single word.
    /// The type `T` must require at least two-byte alignment.
    ///
    /// If you're the kind of programmer who's never met a pointer whose
    /// 2⁰-bit you didn't want to steal, well, now you can do it safely!
    /// ("But it's not nearly as exciting this way...")
    pub struct RefWithFlag<'a, T> {
        ptr_and_bit: usize,
        behaves_like:PhantomData<&'a T> // occupies no space
    }

    impl<'a, T: 'a> RefWithFlag<'a, T> {
        pub fn new(ptr: &'a T, flag: bool) -> RefWithFlag<T> {
            assert!(align_of:: <T>() % 2 == 0);
            RefWithFlag {
                ptr_and_bit: ptr as *const T as usize | flag as usize,
                behaves_like:PhantomData
            }
        }

        pub fn get_ref(&self) ->&'a T {
            unsafe {
                let ptr = (self.ptr_and_bit & !1) as *const T;
                &*ptr
            }
        }

        pub fn get_flag(&self) ->bool {
            self.ptr_and_bit & 1 != 0
        }
    }
}
```

Ce code tire parti du fait que de nombreux types doivent être placés à des adresses paires en mémoire : puisque le bit le moins significatif d'une adresse paire est toujours zéro, nous pouvons y stocker autre chose, puis reconstruire de manière fiable l'adresse d'origine simplement en masquant le bit inférieur. . Tous les types ne sont pas éligibles ; par exemple, les types `u8` et `(bool, [i8; 2])` peuvent être placés à n'importe quelle adresse. Mais nous pouvons vérifier l'alignement du type sur la construction et les types de déchets qui ne fonctionneront pas.

Vous pouvez utiliser `RefWithFlag` comme ceci :

```
use ref_with_flag::RefWithFlag;

let vec = vec![10, 20, 30];
let flagged = RefWithFlag::new(&vec, true);
assert_eq!(flagged.get_ref()[1], 20);
assert_eq!(flagged.get_flag(), true);
```

Le constructeur `RefWithFlag::new` prend une référence et une `bool` valeur, affirme que le type de la référence est approprié, puis convertit la référence en un pointeur brut, puis en un `usize` . Le `usize` type est défini pour être suffisamment grand pour contenir un pointeur sur le processeur pour lequel nous compilons, donc la conversion d'un pointeur brut en a `usize` et inversement est bien définie. Une fois que nous avons un `usize` , nous savons qu'il doit être pair, nous pouvons donc utiliser l' `|` opérateur au niveau du bit ou pour le combiner avec le `bool` , que nous avons converti en un entier 0 ou 1.

La `get_flag` méthode extrait le `bool` composant de a `RefWithFlag` . C'est simple : il suffit de masquer le bit du bas et de vérifier s'il est différent de zéro.

La `get_ref` méthode extrait la référence d'un fichier `RefWithFlag` . Tout d'abord, il masque le `usize` bit inférieur de et le convertit en un pointeur brut. L' `as` opérateur ne convertira pas les pointeurs bruts en références, mais nous pouvons déréférencer le pointeur brut (dans un `unsafe` bloc, naturellement) et l'emprunter. Emprunter le référent d'un pointeur brut vous donne une référence avec une durée de vie illimitée : Rust accordera à la référence la durée de vie qui ferait vérifier le code qui l'entoure, s'il y en a une. Habituellement, cependant, il existe une durée de vie spécifique qui est plus précise et qui détecterait donc plus d'erreurs. Dans ce cas, puisque `get_ref` le type de retour de est `&'a T` , Rust voit que la durée de vie de la référence est la même que `RefWithFlag` le paramètre de durée de vie de `'a` , c'est exactement ce que nous voulons : c'est la durée de vie de la référence avec laquelle nous avons commencé.

En mémoire, a `RefWithFlag` ressemble à a `usize` : puisque `PhantomData` c'est un type de taille nulle, le `behaves_like` champ ne prend pas de place dans la structure. Mais il `PhantomData` est nécessaire que Rust sache comment traiter les durées de vie dans le code qui utilise `RefWithFlag` . Imaginez à quoi ressemblerait le type sans le `behaves_like` champ :

```
// This won't compile.
pub struct RefWithFlag<'a, T: 'a> {
    ptr_and_bit:usize
}
```

Au chapitre 5 , nous avons souligné que toute structure contenant des références ne doit pas survivre aux valeurs qu'elles empruntent, de peur que les références ne deviennent des pointeurs pendants. La structure doit respecter les restrictions qui s'appliquent à ses champs. Cela s'applique certainement à `RefWithFlag` : dans l'exemple de code que nous venons de voir, `flagged` must not outlive `vec` , puisque `flagged.get_ref()` renvoie une référence à celui-ci. Mais notre type réduit `RefWithFlag` ne contient aucune référence et n'utilise jamais son paramètre de durée de vie `'a` . C'est juste un `usize` . Comment Rust doit-il savoir que des restrictions s'appliquent à `flagged` la durée de vie de ? L' inclusion d'un `PhantomData<&'a T>` champ indique à Rust de traiter `RefWithFlag<'a, T>` *comme s'il* contenait un `&'a T` , sans affecter réellement la représentation de la structure.

Bien que Rust ne sache pas vraiment ce qui se passe (c'est ce qui le rend `RefWithFlag` dangereux), il fera de son mieux pour vous aider. Si vous omettez le `behaves_like` champ, Rust se plaindra que les paramètres `'a` et `T` ne sont pas utilisés et suggérera d'utiliser un fichier `PhantomData` .

`RefWithFlag` utilise la même tactique que le `Ascii` type que nous avons présenté précédemment pour éviter un comportement indéfini dans son `unsafe` bloc. Le type lui-même est `pub` , mais ses champs ne le sont pas, ce qui signifie que seul le code du `ref_with_flag` module peut créer ou regarder à l'intérieur d'une `RefWithFlag` valeur. Vous n'avez pas besoin d'inspecter beaucoup de code pour être sûr que le `ptr_and_bit` champ est bien construit.

Pointeurs nullables

Un nulle pointeur brut dans Rust est une adresse zéro, tout comme en C et C++. Pour tout type `T` , la `std::ptr::null<T>` fonction renvoie un `*const T` pointeur null et `std::ptr::null_mut<T>` renvoie un `*mut T` pointeur null.

Il existe plusieurs façons de vérifier si un pointeur brut est nul. Le plus simple est la `is_null` méthode, mais la `as_ref` méthode peut être plus

pratique : elle prend un `*const T` pointeur et renvoie un `Option<&'a T>` , transformant un pointeur nul en un `None` . De même, la `as_mut` méthode convertit les `*mut T` pointeurs en `Option<&'a mut T>` valeurs.

Tailles et alignements des caractères

Une valeurde tout `Sized` typeoccupe un nombre constant d'octets en mémoire et doit être placé à une adresse qui est un multiple d'un certain *alignement*valeur, déterminée par l'architecture de la machine. Par exemple, un `(i32, i32)` tuple occupe huit octets et la plupart des processeurs préfèrent qu'il soit placé à une adresse multiple de quatre.

L'appel `std::mem::size_of::<T>()` renvoie la taille d'une valeur de type `T` , en octets, et `std::mem::align_of::<T>()` renvoie son alignement requis. Par exemple:

```
assert_eq!(std:: mem:: size_of:: <i64>(), 8);
assert_eq!(std:: mem:: align_of::<(i32, i32)>(), 4);
```

L'alignement de n'importe quel type est toujours une puissance de deux.

La taille d'un type est toujours arrondie à un multiple de son alignement, même s'il pourrait techniquement tenir dans moins d'espace. Par exemple, même si un tuple comme `(f32, u8)` ne nécessite que cinq octets, `size_of::<(f32, u8)>()` est `8` , car `align_of::<(f32, u8)>()` est `4` . Cela garantit que si vous avez un tableau, la taille du type d'élément reflète toujours l'espacement entre un élément et le suivant.

Pour les types non dimensionnés, la taille et l'alignement dépendent de la valeur disponible. Étant donné une référence à une valeur non dimensionnée, les fonctions `std::mem::size_of_val` et `std::mem::align_of_val` renvoient la taille et l'alignement de la valeur. Ces fonctions peuvent opérer sur des références à des `Sized` types à la fois et non dimensionnés :

```
// Fat pointers to slices carry their referent's length.
let slice: &[i32] = &[1, 3, 9, 27, 81];
assert_eq!(std:: mem::size_of_val(slice), 20);

let text: &str = "alligator";
assert_eq!(std:: mem::size_of_val(text), 9);

use std:: fmt:: Display;
```

```
let unremarkable: &dyn Display = &193_u8;
let remarkable:&dyn Display = &0.0072973525664;

// These return the size/alignment of the value the
// trait object points to, not those of the trait object
// itself. This information comes from the vtable the
// trait object refers to.
assert_eq!(std:: mem:: size_of_val(unremarkable), 1);
assert_eq!(std:: mem::align_of_val(remarkable), 8);
```

Arithmétique du pointeur

Rouillerprésente les éléments d'un tableau, d'une tranche ou d'un vecteur sous la forme d'un seul bloc de mémoire contigu, comme illustré à la Figure 22-1 . Les éléments sont régulièrement espacés, de sorte que si chaque élément occupe `size` des octets, alors le `i` ème élément commence par le `i * size` ème octet.

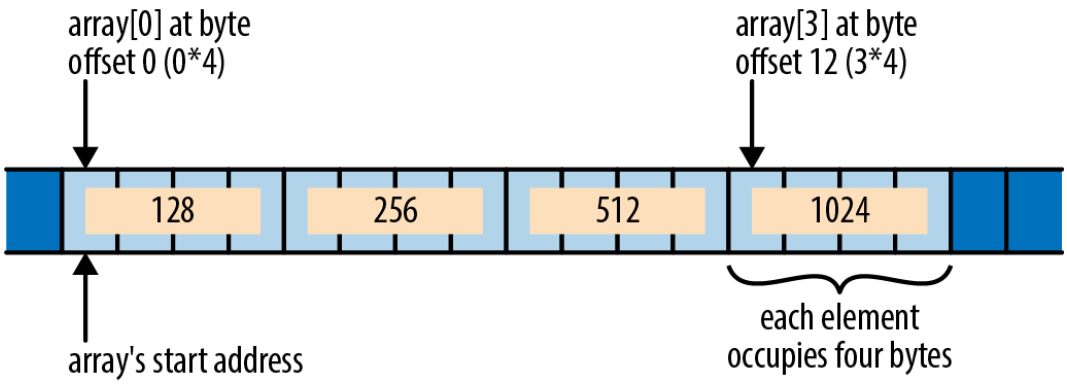

Image 22-1. Un tableau en mémoire

Une bonne conséquence de ceci est que si vous avez deux pointeurs bruts vers des éléments d'un tableau, la comparaison des pointeurs donne les mêmes résultats que la comparaison des indices des éléments : si `i < j`, alors un pointeur brut vers le `i` ème élément est inférieur à un pointeur brut vers le `j` ème élément. Cela rend les pointeurs bruts utiles comme limites sur les traversées de tableaux. En fait, l'itérateur simple de la bibliothèque standard sur une tranche était initialement défini comme ceci :

```
struct Iter<'a, T> {
    ptr: *const T,
    end:*const T,
    ...
}
```

Le `ptr` champ pointe vers l'élément suivant que l'itération doit produire, et le `end` champ sert de limite : quand `ptr == end` , l'itération est terminée.

Une autre conséquence intéressante de la disposition des tableaux : si `element_ptr` est un pointeur brut `*const T` ou vers le ème élément d'un tableau, alors est un pointeur brut vers le ème élément. Sa définition est équivalente à ceci : `*mut T i element_ptr.offset(o) (i + o)`

```
fn offset<T>(ptr: *const T, count: isize) -> *const T
    where T: Sized
{
    let bytes_per_element = std:: mem:: size_of::<T>() as isize;
    let byte_offset = count * bytes_per_element;
    (ptr as isize).checked_add(byte_offset).unwrap() as *const T
}
```

La `std::mem::size_of::<T>` fonction renvoie la taille du type `T` en octets. Comme `isize` est, par définition, assez grand pour contenir une adresse, vous pouvez convertir le pointeur de base en un `isize` , effectuer une arithmétique sur cette valeur, puis reconvertir le résultat en un pointeur.

C'est bien de produire un pointeur sur le premier octet après la fin d'un tableau. Vous ne pouvez pas déréférencer un tel pointeur, mais il peut être utile pour représenter la limite d'une boucle ou pour des contrôles de limites.

Cependant, il s'agit d'un comportement indéfini à utiliser `offset` pour produireun pointeur au-delà de ce point ou avant le début du tableau, même si vous ne le déréférencez jamais. Dans un souci d'optimisation, Rust aimerait supposer que `ptr.offset(i) > ptr` quand `i` est positif et que `ptr.offset(i) < ptr` quand `i` est négatif. Cette hypothèse semble sûre, mais elle peut ne pas tenir si l'arithmétique `offset` dépasse une `isize` valeur. Si `i` est contraint de rester dans le même tableau que `ptr` , aucun débordement ne peut se produire : après tout, le tableau lui-même ne dépasse pas les limites de l'espace d'adressage. (Pour créer des pointeurs vers le premier octet après la fin du coffre-fort, Rust ne place jamais de valeurs à l'extrémité supérieure de l'espace d'adressage.)

Si vous avez besoin de décaler des pointeurs au-delà des limites du tableau auquel ils sont associés, vous pouvez utiliser la `wrapping_offset` méthode. Ceci est équivalent à `offset` , mais Rust ne

fait aucune hypothèse sur l'ordre relatif de `ptr.wrapping_offset(i)` et lui- `ptr` même. Bien sûr, vous ne pouvez toujours pas déréférencer ces pointeurs à moins qu'ils ne fassent partie du tableau.

Entrer et sortir de la mémoire

Si vous implémentez un type qui gère sa propre mémoire, vous devrez suivre quelles parties de votre mémoire contiennent des valeurs actives et lesquelles ne sont pas initialisées, tout comme Rust le fait avec les variables locales. Considérez ce code :

```
let pot = "pasta".to_string();
let plate = pot;
```

Une fois ce code exécuté, la situation ressemble à la Figure 22-2 .

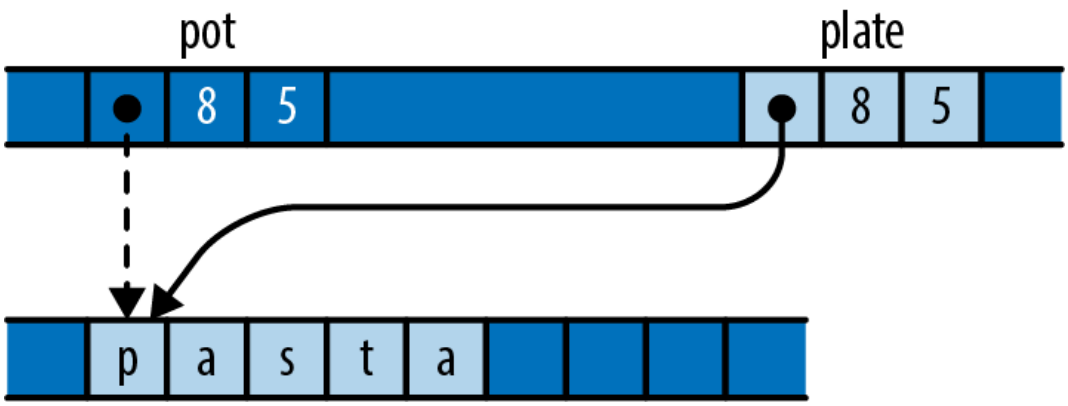

Illustration 22-2. Déplacer une chaîne d'une variable locale à une autre

Après l'affectation, `pot` n'est pas initialisé et `plate` est le propriétaire de la chaîne.

Au niveau de la machine, il n'est pas spécifié ce qu'un mouvement fait à la source, mais en pratique, il ne fait généralement rien du tout. L'affectation laisse probablement `pot` encore un pointeur, une capacité et une longueur pour la chaîne. Naturellement, il serait désastreux de traiter cela comme une valeur en direct, et Rust garantit que vous ne le ferez pas.

Les mêmes considérations s'appliquent aux structures de données qui gèrent leur propre mémoire. Supposons que vous exécutiez ce code :

```
let mut noodles = vec!["udon".to_string()];
let soba = "soba".to_string();
let last;
```

En mémoire, l'état ressemble à la Figure 22-3 .

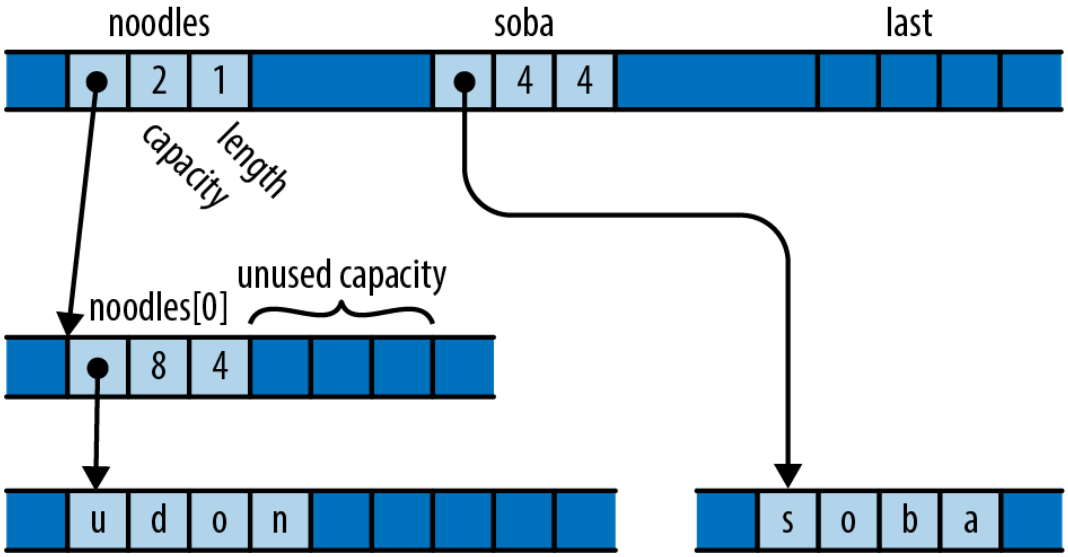

Illustration 22-3. Un vecteur avec une capacité de réserve non initialisée

Le vecteur a la capacité de réserve pour contenir un élément de plus, mais son contenu est indésirable, probablement quel que soit ce que la mémoire contenait auparavant. Supposons que vous exécutiez ensuite ce code :

```
noodles.push(soba);
```

Pousser la chaîne sur le vecteur transforme cette mémoire non initialisée en un nouvel élément, comme illustré à la Figure 22-4 .

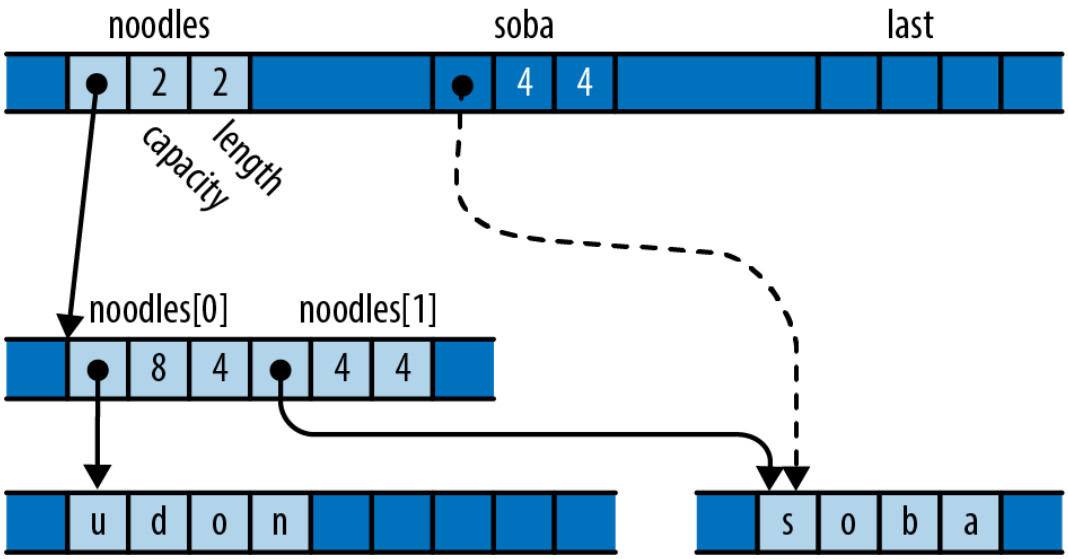

Image 22-4. Après avoir poussé `soba` la valeur de sur le vecteur

Le vecteur a initialisé son espace vide pour posséder la chaîne et incrémenté sa longueur pour le marquer comme un nouvel élément actif. Le vecteur est maintenant le propriétaire de la chaîne ; vous pouvez vous ré-

férer à son deuxième élément, et supprimer le vecteur libérerait les deux chaînes. Et `soba` est maintenant non initialisé.

Enfin, considérez ce qui se passe lorsque nous extrayons une valeur du vecteur :

```
last = noodles.pop().unwrap();
```

En mémoire, les choses ressemblent maintenant à la Figure 22-5 .

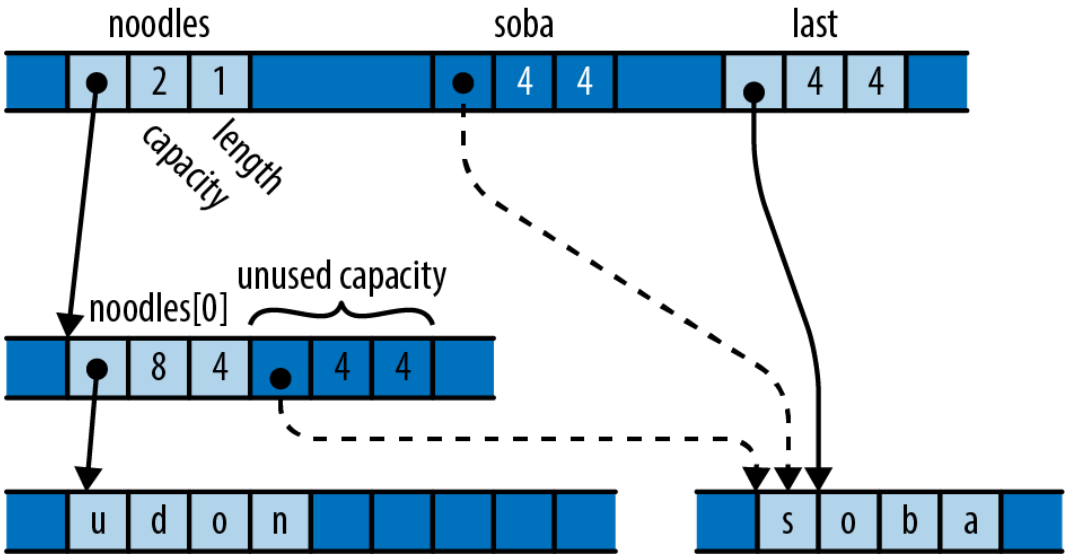

Illustration 22-5. Après avoir inséré un élément du vecteur dans `last`

La variable `last` a pris possession de la chaîne. Le vecteur a décrémenté sa longueur pour indiquer que l'espace qui contenait la chaîne n'est plus initialisé.

Tout comme avec `pot` et `pasta` précédemment, les trois de `soba` , `last` , et l'espace libre du vecteur contiennent probablement des modèles de bits identiques. Mais seul `last` est considéré comme propriétaire de la valeur. Traiter l'un ou l'autre des deux autres emplacements comme étant en direct serait une erreur.

La véritable définition d'une valeur initialisée est celle qui est *traitée comme live* . L'écriture dans les octets d'une valeur est généralement une partie nécessaire de l'initialisation, mais uniquement parce que cela prépare la valeur à être traitée comme active. Un déplacement et une copie ont tous deux le même effet sur la mémoire ; la différence entre les deux est qu'après un déplacement, la source n'est plus traitée comme en direct, alors qu'après une copie, la source et la destination sont en direct.

Rust suit les variables locales actives au moment de la compilation et vous empêche d'utiliser des variables dont les valeurs ont été déplacées ailleurs. Des types tels que `Vec`, `HashMap`, `Box`, etc. suivent leurs tampons de manière dynamique. Si vous implémentez un type qui gère sa propre mémoire, vous devrez faire de même.

Rust fournit deux opérations essentielles pour implémenter de tels types:

`std::ptr::read(src)`

Se déplaceune valeur hors de l'emplacement `src` vers lequel pointe, transférant la propriété à l'appelant. L' `src` argument doit être un `*const T` pointeur brut, où `T` est un type dimensionné. Après avoir appelé cette fonction, le contenu de `*src` n'est pas affecté, mais à moins que ce ne `T` soit `Copy`, vous devez vous assurer que votre programme les traite comme de la mémoire non initialisée.

C'est l'opération derrière `Vec::pop`. Extraire une valeur appelle `read` à déplacer la valeur hors du tampon, puis décrémente la longueur pour marquer cet espace comme capacité non initialisée.

`std::ptr::write(dest, value)`

Se déplace `value` dansl'emplacement `dest` pointe vers, qui doit être une mémoire non initialisée avant l'appel. Le référent est désormais propriétaire de la valeur. Ici, `dest` doit être un `*mut T` pointeur brut et `value` une `T` valeur, où `T` est un type dimensionné.

C'est l'opération derrière `Vec::push`. Pousser une valeur appelle `write` à déplacer la valeur dans l'espace disponible suivant, puis incrémente la longueur pour marquer cet espace comme un élément valide.

Les deux sont des fonctions libres, pas des méthodes sur les types de pointeurs bruts.

Notez que vous ne pouvez pas faire ces choses avec l'un des types de pointeurs sûrs de Rust. Ils exigent tous que leurs référents soient initialisés à tout moment, donc transformer une mémoire non initialisée en une valeur, ou vice versa, est hors de leur portée. Les pointeurs bruts font l'affaire.

La bibliothèque standard fournit également des fonctions pour déplacer des tableaux de valeurs d'un bloc de mémoire à un autre :

`std::ptr::copy(src, dst, count)`

Se déplacele tableau de `count` valeurs en mémoire commençant à `src` jusqu'à la mémoire à `dst` , comme si vous aviez écrit une boucle d' appels `read` et pour les déplacer un par un. `write` La mémoire destination doit être désinitialisée avant l'appel, et ensuite la mémoire source reste non initialisée. Les arguments `src` et doivent être des pointeurs bruts et doivent être un . `dest *const T *mut T count usize`

ptr. `copy_to` (dst, compter)

Un plusune version pratique de `copy` qui déplace le tableau de `count` valeurs en mémoire de `ptr` à `dst` , plutôt que de prendre son point de départ comme argument.

`std::ptr::copy_nonoverlapping(src, dst, count)`

Commel'appel correspondant à `copy` , sauf que son contrat exige en outre que les blocs de mémoire source et destination ne doivent pas se chevaucher. Cela peut être légèrement plus rapide que d'appeler `copy` .

`ptr.copy_to_nonoverlapping(dst, count)`

Un plusversion pratique de `copy_nonoverlapping` , comme `copy_to` .

Il existe deux autres familles de fonctions `read` et `write` , également dans le `std::ptr` module :

`read_unaligned`, `write_unaligned`

Cesles fonctions sont comme `read` et `write` , sauf que le pointeur n'a pas besoin d'être aligné comme normalement requis pour le type référent. Ces fonctions peuvent être plus lentes que les fonctions plain `read` et `write` .

`read_volatile`, `write_volatile`

Cesles fonctions sont l'équivalent des lectures et écritures volatilesen C ou C++.

Exemple : GapBuffer

Voici un exemplequi utilise les fonctions de pointeur brutes décrites ci-dessus.

Supposons que vous écriviez un éditeur de texte et que vous recherchiez un type pour représenter le texte. Vous pouvez choisir `String` et utiliser

les méthodes `insert` et `remove` pour insérer et supprimer des caractères au fur et à mesure que l'utilisateur tape. Mais s'ils éditent du texte au début d'un fichier volumineux, ces méthodes peuvent être coûteuses : l'insertion d'un nouveau caractère implique de décaler tout le reste de la chaîne vers la droite en mémoire, et la suppression la décale entièrement vers la gauche. Vous aimeriez que ces opérations courantes soient moins chères.

L'éditeur de texte Emacs utilise une structure de données simple appelée *tampon* d'espacement qui peut insérer et supprimer des caractères en temps constant. Tandis que a `String` garde toute sa capacité de réserve à la fin du texte, ce qui rend `push` et `pop` bon marché, un tampon d'écart garde sa capacité de réserve au milieu du texte, à l'endroit où l'édition a lieu. Cette capacité de réserve s'appelle l' *écart* . L'insertion ou la suppression d'éléments au niveau de l'espace est bon marché : il vous suffit de réduire ou d'agrandir l'espace selon vos besoins. Vous pouvez déplacer l'espace à n'importe quel endroit de votre choix en déplaçant le texte d'un côté de l'espace à l'autre. Lorsque l'espace est vide, vous migrez vers un tampon plus grand.

Alors que l'insertion et la suppression dans un tampon d'espace sont rapides, la modification de la position à laquelle elles ont lieu implique le déplacement de l'espace vers la nouvelle position. Le déplacement des éléments nécessite un temps proportionnel à la distance parcourue. Heureusement, une activité d'édition typique implique de faire un tas de changements dans un quartier du tampon avant de partir et de jouer avec du texte ailleurs.

Dans cette section, nous allons implémenter un tampon d'écart dans Rust. Pour éviter d'être distrait par UTF-8, nous allons faire en sorte que nos `char` valeurs de stockage tampon soient directement, mais les principes de fonctionnement seraient les mêmes si nous stockions le texte sous une autre forme.

Tout d'abord, nous allons montrer un tampon d'écart en action. Ce code crée un `GapBuffer` , y insère du texte, puis déplace le point d'insertion juste avant le dernier mot :

```
let mut buf = GapBuffer::new();
buf.insert_iter("Lord of the Rings".chars());
buf.set_position(12);
```

Après avoir exécuté ce code, le tampon ressemble à celui illustré à la Figure 22-6 .

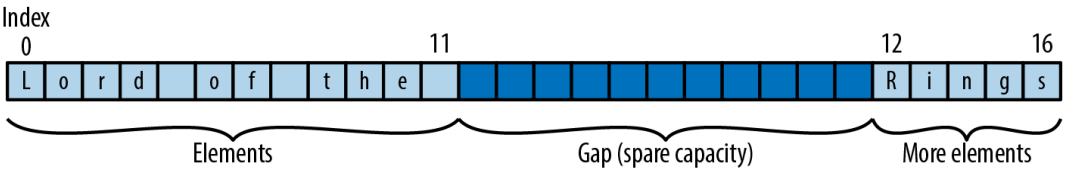

Image 22-6. Un tampon vide contenant du texte

L'insertion consiste à combler le vide avec un nouveau texte. Ce code ajoute un mot et ruine le film :

```
buf.insert_iter("Onion ".chars());
```

Il en résulte l'état illustré à la Figure 22-7 .

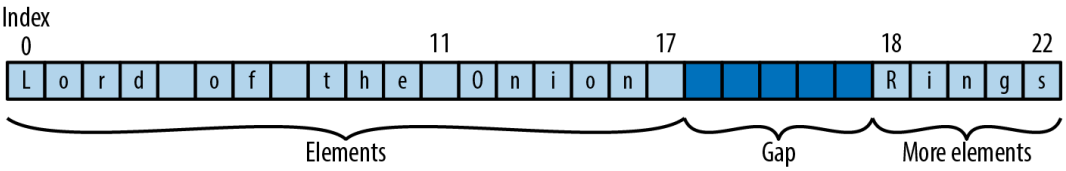

Image 22-7. Un tampon vide contenant du texte supplémentaire

Voici notre `GapBuffer` type :

```
use std;
use std:: ops::Range;

pub struct GapBuffer<T> {
    // Storage for elements. This has the capacity we need, but its lengt
    // always remains zero. GapBuffer puts its elements and the gap in th
    // `Vec`'s "unused" capacity.
    storage:Vec<T>,

    // Range of uninitialized elements in the middle of `storage`.
    // Elements before and after this range are always initialized.
    gap:Range<usize>
}
```

`GapBuffer` utilise son `storage` champ d'une manière étrange. [2] Il ne stocke jamais réellement d'éléments dans le vecteur, ou pas tout à fait. Il appelle simplement `Vec::with_capacity(n)` pour obtenir un bloc de mémoire suffisamment grand pour contenir des `n` valeurs, obtient des pointeurs bruts vers cette mémoire via les vecteurs `as_ptr` et les `as_mut_ptr` méthodes, puis utilise le tampon directement à ses propres

fins. La longueur du vecteur reste toujours nulle. Lorsque le `Vec` est supprimé, le `Vec` n'essaie pas de libérer ses éléments, car il ne sait pas qu'il en a, mais il libère le bloc de mémoire. C'est ce qui `GapBuffer` veut; il a sa propre `Drop` implémentation qui sait où se trouvent les éléments actifs et les supprime correctement.

`GapBuffer` Les méthodes les plus simples correspondent à ce que vous attendez :

```
impl<T> GapBuffer<T> {
    pub fn new() -> GapBuffer<T> {
        GapBuffer { storage: Vec:: new(), gap:0..0 }
    }

    /// Return the number of elements this GapBuffer could hold without
    /// reallocation.
    pub fn capacity(&self) ->usize {
        self.storage.capacity()
    }

    /// Return the number of elements this GapBuffer currently holds.
    pub fn len(&self) ->usize {
        self.capacity() - self.gap.len()
    }

    /// Return the current insertion position.
    pub fn position(&self) ->usize {
        self.gap.start
    }

    ...
}
```

Il nettoie la plupart des fonctions suivantes pour avoir une méthode utilitaire qui renvoie un pointeur brut vers l'élément buffer à un index donné. Ceci étant Rust, nous finissons par avoir besoin d'une méthode pour les `mut` pointeurs et d'une autre pour `const` . Contrairement aux méthodes précédentes, celles-ci ne sont pas publiques. Continuation de ce `impl` bloc :

```
/// Return a pointer to the `index`th element of the underlying storage,
/// regardless of the gap.
///
/// Safety: `index` must be a valid index into `self.storage`.
unsafe fn space(&self, index: usize) ->*const T {
```

```
        self.storage.as_ptr().offset(index as isize)
    }

    /// Return a mutable pointer to the `index`th element of the underlying
    /// storage, regardless of the gap.
    ///
    /// Safety: `index` must be a valid index into `self.storage`.
    unsafe fn space_mut(&mut self, index: usize) ->*mut T {
        self.storage.as_mut_ptr().offset(index as isize)
    }
```

Pour trouver l'élément à un index donné, vous devez déterminer si l'index tombe avant ou après l'écart et ajuster en conséquence :

```
    /// Return the offset in the buffer of the `index`th element, taking
    /// the gap into account. This does not check whether index is in range,
    /// but it never returns an index in the gap.
    fn index_to_raw(&self, index: usize) ->usize {
        if index < self.gap.start {
            index
        } else {
            index + self.gap.len()
        }
    }

    /// Return a reference to the `index`th element,
    /// or `None` if `index` is out of bounds.
    pub fn get(&self, index: usize) ->Option<&T> {
        let raw = self.index_to_raw(index);
        if raw < self.capacity() {
            unsafe {
                // We just checked `raw` against self.capacity(),
                // and index_to_raw skips the gap, so this is safe.
                Some(&*self.space(raw))
            }
        } else {
            None
        }
    }
```

Lorsque nous commençons à faire des insertions et des suppressions dans une autre partie du tampon, nous devons déplacer l'espace vers le nouvel emplacement. Déplacer l'espace vers la droite implique de déplacer des éléments vers la gauche, et vice versa, tout comme la bulle d'un niveau à bulle se déplace dans un sens lorsque le fluide s'écoule dans l'autre :

```
    /// Set the current insertion position to `pos`.
    /// If `pos` is out of bounds, panic.
    pub fn set_position(&mut self, pos:usize) {
        if pos > self.len() {
            panic!("index {} out of range for GapBuffer", pos);
        }

        unsafe {
            let gap = self.gap.clone();
            if pos > gap.start {
                // `pos` falls after the gap. Move the gap right
                // by shifting elements after the gap to before it.
                let distance = pos - gap.start;
                std:: ptr:: copy(self.space(gap.end),
                                 self.space_mut(gap.start),
                                 distance);
            } else if pos < gap.start {
                // `pos` falls before the gap. Move the gap left
                // by shifting elements before the gap to after it.
                let distance = gap.start - pos;
                std:: ptr::copy(self.space(pos),
                                 self.space_mut(gap.end - distance),
                                 distance);
            }

            self.gap = pos .. pos + gap.len();
        }
    }
```

Cette fonction utilise la `std::ptr::copy` méthode pour décaler les éléments ; `copy` nécessite que la destination soit non initialisée et laisse la source non initialisée. Les plages source et destination peuvent se chevaucher, mais `copy` gère ce cas correctement. Étant donné que l'espace est une mémoire non initialisée avant l'appel et que la fonction ajuste la position de l'espace pour couvrir l'espace libéré par la copie, le `copy` contrat de la fonction est satisfait.

L'insertion et le retrait d'éléments sont relativement simples. L'insertion prend un espace à partir de l'espace pour le nouvel élément, tandis que la suppression déplace une valeur vers l'extérieur et agrandit l'espace pour couvrir l'espace qu'il occupait auparavant :

```
    /// Insert `elt` at the current insertion position,
    /// and leave the insertion position after it.
    pub fn insert(&mut self, elt:T) {
        if self.gap.len() == 0 {
```

```
            self.enlarge_gap();
        }

        unsafe {
            let index = self.gap.start;
            std:: ptr::write(self.space_mut(index), elt);
        }
        self.gap.start += 1;
    }

    /// Insert the elements produced by `iter` at the current insertion
    /// position, and leave the insertion position after them.
    pub fn insert_iter<I>(&mut self, iterable: I)
        where I:IntoIterator<Item=T>
    {
        for item in iterable {
            self.insert(item)
        }
    }

    /// Remove the element just after the insertion position
    /// and return it, or return `None` if the insertion position
    /// is at the end of the GapBuffer.
    pub fn remove(&mut self) ->Option<T> {
        if self.gap.end == self.capacity() {
            return None;
        }

        let element = unsafe {
            std:: ptr::read(self.space(self.gap.end))
        };
        self.gap.end += 1;
        Some(element)
    }
```

Semblable à la façon dont `Vec` utilise `std::ptr::write` pour pousser et `std::ptr::read` pour `pop`, `GapBuffer` utilise `write` pour `insert` et `read` pour `remove`. Et tout comme `Vec` doit ajuster sa longueur pour maintenir la frontière entre les éléments initialisés et la capacité de réserve, `GapBuffer` ajuste son écart.

Lorsque l'espace a été comblé, la `insert` méthode doit agrandir la mémoire tampon pour acquérir plus d'espace libre. La `enlarge_gap` méthode (la dernière du `impl` bloc) gère ceci :

```
    /// Double the capacity of `self.storage`.
    fn enlarge_gap(&mut self) {
```

```
        let mut new_capacity = self.capacity() * 2;
        if new_capacity == 0 {
            // The existing vector is empty.
            // Choose a reasonable starting capacity.
            new_capacity = 4;
        }

        // We have no idea what resizing a Vec does with its "unused"
        // capacity. So just create a new vector and move over the elements.
        let mut new = Vec::with_capacity(new_capacity);
        let after_gap = self.capacity() - self.gap.end;
        let new_gap = self.gap.start .. new.capacity() - after_gap;

        unsafe {
            // Move the elements that fall before the gap.
            std:: ptr::copy_nonoverlapping(self.space(0),
                                           new.as_mut_ptr(),
                                           self.gap.start);

            // Move the elements that fall after the gap.
            let new_gap_end = new.as_mut_ptr().offset(new_gap.end as isize);
            std:: ptr::copy_nonoverlapping(self.space(self.gap.end),
                                           new_gap_end,
                                           after_gap);
        }

        // This frees the old Vec, but drops no elements,
        // because the Vec's length is zero.
        self.storage = new;
        self.gap = new_gap;
    }
```

Alors que `set_position` doit utiliser `copy` pour déplacer des éléments d'avant en arrière dans l'espace, `enlarge_gap` peut utiliser `copy_nonoverlapping`, car il déplace des éléments vers un tout nouveau tampon.

Déplacer le nouveau vecteur dans `self.storage` les gouttes de l'ancien vecteur. Comme sa longueur est nulle, l'ancien vecteur pense qu'il n'a aucun élément à supprimer et libère simplement son tampon. Soigneusement, `copy_nonoverlapping` laisse sa source non initialisée, donc l'ancien vecteur a raison dans cette croyance : tous les éléments appartiennent maintenant au nouveau vecteur.

Enfin, nous devons nous assurer que déposer a `GapBuffer` supprime tous ses éléments :

```
impl<T> Drop for GapBuffer<T> {
    fn drop(&mut self) {
        unsafe {
            for i in 0 .. self.gap.start {
                std:: ptr:: drop_in_place(self.space_mut(i));
            }
            for i in self.gap.end .. self.capacity() {
                std:: ptr::drop_in_place(self.space_mut(i));
            }
        }
    }
}
```

Les éléments se trouvent avant et après l'écart, nous parcourons donc chaque région et utilisons la `std::ptr::drop_in_place` fonction pour supprimer chacun d'eux. La `drop_in_place` fonction est un utilitaire qui se comporte comme `drop(std::ptr::read(ptr))` , mais ne prend pas la peine de déplacer la valeur vers son appelant (et fonctionne donc sur des types non dimensionnés). Et tout comme dans `enlarge_gap` , au moment où le vecteur `self.storage` est supprimé, son tampon n'est vraiment pas initialisé.

Comme les autres types que nous avons montrés dans ce chapitre, `GapBuffer` garantit que ses propres invariants sont suffisants pour s'assurer que le contrat de chaque fonctionnalité non sécurisée qu'il utilise est respecté, de sorte qu'aucune de ses méthodes publiques n'a besoin d'être marquée comme non sécurisée. `GapBuffer` implémente une interface sécurisée pour une fonctionnalité qui ne peut pas être écrite efficacement dans un code sécurisé.

Sécurité anti-panique dans un code dangereux

A Rust, paniquene peut généralement pas provoquer un comportement indéfini ; la `panic!` macro n'est pas une fonctionnalité dangereuse. Mais lorsque vous décidez de travailler avec un code non sécurisé, la sécurité anti-panique fait partie de votre travail.

Considérez la `GapBuffer::remove` méthode de la section précédente :

```
pub fn remove(&mut self) ->Option<T> {
    if self.gap.end == self.capacity() {
        return None;
    }
```

```
        let element = unsafe {
            std:: ptr::read(self.space(self.gap.end))
        };
        self.gap.end += 1;
        Some(element)
    }
```

L'appel à `read` déplace l'élément suivant immédiatement l'espace hors du tampon, laissant derrière lui un espace non initialisé. À ce stade, le `GapBuffer` est dans un état incohérent : nous avons rompu l'invariant selon lequel tous les éléments en dehors de l'espace doivent être initialisés. Heureusement, la toute prochaine déclaration agrandit l'écart pour couvrir cet espace, donc au moment où nous revenons, l'invariant tient à nouveau.

Mais considérez ce qui se passerait si, après l'appel à `read` mais avant l'ajustement à `self.gap.end`, ce code tentait d'utiliser une fonctionnalité susceptible de paniquer, par exemple l'indexation d'une tranche. Quitter la méthode brusquement n'importe où entre ces deux actions laisserait le `GapBuffer` avec un élément non initialisé en dehors de l'espace. Le prochain appel à `remove` pourrait essayer à `read` nouveau ; même simplement laisser tomber le `GapBuffer` essaierait de le laisser tomber. Les deux ont un comportement indéfini, car ils accèdent à une mémoire non initialisée.

Il est presque inévitable que les méthodes d'un type relâchent momentanément les invariants du type pendant qu'elles font leur travail, puis remettent tout en ordre avant leur retour. Une méthode intermédiaire de panique pourrait raccourcir ce processus de nettoyage, laissant le type dans un état incohérent.

Si le type utilise uniquement du code sécurisé, cette incohérence peut entraîner un mauvais comportement du type, mais elle ne peut pas introduire de comportement indéfini. Mais le code utilisant des fonctionnalités non sécurisées compte généralement sur ses invariants pour respecter les contrats de ces fonctionnalités. Les invariants rompus conduisent à des contrats rompus, qui conduisent à un comportement indéfini.

Lorsque vous travaillez avec des fonctionnalités non sécurisées, vous devez faire particulièrement attention à identifier ces régions sensibles du code où les invariants sont temporairement relâchés, et vous assurer qu'ils ne font rien qui pourrait paniquer..

Réinterpréter la mémoire avec les syndicats

Rouillerfournit de nombreuses abstractions utiles, mais en fin de compte, le logiciel que vous écrivez ne fait que pousser des octets. Les unions sont l'une des fonctionnalités les plus puissantes de Rust pour manipuler ces octets et choisir comment ils sont interprétés. Par exemple, toute collection de 32 bits - 4 octets - peut être interprétée comme un entier ou comme un nombre à virgule flottante. L'une ou l'autre interprétation est valide, bien que l'interprétation des données destinées à l'une comme l'autre entraînera probablement un non-sens.

Une union représentant une collection d'octets pouvant être interprétés comme un entier ou un nombre à virgule flottante s'écrirait comme suit :

```
union FloatOrInt {
    f: f32,
    i:i32,
}
```

C'est une union avec deux champs, `f` et `i`. Ils peuvent être affectés à la même manière que les champs d'une structure, mais lors de la construction d'une union, contrairement à une structure, vous devez en choisir exactement un. Là où les champs d'une structure font référence à différentes positions en mémoire, les champs d'une union font référence à différentes interprétations de la même séquence de bits. Attribuer à un champ différent signifie simplement écraser certains ou tous ces bits, conformément à un type approprié. Ici, `one` fait référence à une seule étendue de mémoire de 32 bits, qui stocke d'abord `1` codée sous la forme d'un entier simple, puis `1.0` sous la forme d'un nombre à virgule flottante IEEE 754. Dès que `f` est écrit dans , la valeur précédemment écrite dans `FloatOrInt` est écrasée :

```
let mut one = FloatOrInt { i:1 };
assert_eq!(unsafe { one.i }, 0x00_00_00_01);
one.f = 1.0;
assert_eq!(unsafe { one.i }, 0x3F_80_00_00);
```

Pour la même raison, la taille d'une union est déterminée par son plus grand champ. Par exemple, cette union a une taille de 64 bits, même si ce `SmallOrLarge::s` n'est qu'un `bool` :

```
union SmallOrLarge {
    s: bool,
    l:u64
}
```

Bien que la construction d'une union ou l'affectation à ses champs soit totalement sûre, la lecture à partir de n'importe quel champ d'une union est toujours dangereuse :

```
let u = SmallOrLarge { l:1337 };
println!("{}", unsafe {u.l}); // prints 1337
```

En effet, contrairement aux énumérations, les unions n'ont pas de balise. Le compilateur n'ajoute aucun bit supplémentaire pour différencier les variantes. Il n'y a aucun moyen de savoir au moment de l'exécution si a `SmallOrLarge` doit être interprété comme a `u64` ou a `bool` , à moins que le programme n'ait un contexte supplémentaire.

Il n'y a pas non plus de garantie intégrée que la configuration binaire d'un champ donné est valide. Par exemple, écrire dans le champ d'une `SmallOrLarge` valeur `l` écrasera son `s` champ, créant un motif de bits qui ne signifie certainement rien d'utile et qui n'est probablement pas valide `bool` . Par conséquent, bien que l'écriture dans les champs union soit sûre, chaque lecture nécessite `unsafe` . La lecture de `u.s` n'est autorisée que lorsque les bits du `s` champ forment un `bool` ; sinon, il s'agit d'un comportement indéfini.

Avec ces restrictions à l'esprit, les unions peuvent être un moyen utile de réinterpréter temporairement certaines données, en particulier lors de calculs sur la représentation des valeurs plutôt que sur les valeurs elles-mêmes. Par exemple, le type mentionné précédemment `FloatOrInt` peut facilement être utilisé pour imprimer les bits individuels d'un nombre à virgule flottante, même s'il `f32` n'implémente pas le `Binary` formateur :

```
let float = FloatOrInt { f:31337.0 };
// prints 10001101111010011010010000000000
println!("{:b}", unsafe { float.i });
```

Bien que ces exemples simples fonctionnent presque certainement comme prévu sur n'importe quelle version du compilateur, il n'y a aucune garantie qu'un champ commence à un endroit spécifique à moins

qu'un attribut ne soit ajouté à la `union` définition indiquant au compilateur comment disposer les données en mémoire. L'ajout de l'attribut `#[repr(C)]` garantit que tous les champs commencent à l'offset 0, plutôt qu'à l'endroit souhaité par le compilateur. Avec cette garantie en place, le comportement d'écrasement peut être utilisé pour extraire des bits individuels, comme le bit de signe d'un entier :

```
#[repr(C)]
union SignExtractor {
    value: i64,
    bytes:[u8; 8]
}

fn sign(int: i64) -> bool {
    let se = SignExtractor { value:int};
    println!( "{:b} ({:?})", unsafe { se.value }, unsafe { se.bytes });
    unsafe { se.bytes[7] >= 0b10000000 }
}

assert_eq!(sign(-1), true);
assert_eq!(sign(1), false);
assert_eq!(sign(i64:: MAX), false);
assert_eq!(sign(i64::MIN), true);
```

Ici, le bit de signe est le bit le plus significatif de l'octet le plus significatif. Comme les processeurs x86 sont little-endian, l'ordre de ces octets est inversé ; l'octet le plus significatif n'est pas `bytes[0]`, mais `bytes[7]`. Normalement, ce n'est pas quelque chose que le code Rust doit gérer, mais comme ce code travaille directement avec la représentation en mémoire du `i64`, ces détails de bas niveau deviennent importants.

Étant donné que les syndicats ne peuvent pas dire comment supprimer leur contenu, tous leurs champs doivent être `Copy`. Cependant, si vous devez simplement stocker a `String` dans une union, il existe une solution de contournement ; consultez la documentation de la bibliothèque standard pour `std::mem::ManuallyDrop`.

Unions correspondantes

Correspondant àsur une union Rust, c'est comme faire correspondre une structure, sauf que chaque motif doit spécifier exactement un champ :

```
unsafe {
    match u {
        SmallOrLarge { s: true } => { println!("boolean true"); }
        SmallOrLarge { l:2 } => { println!("integer 2"); }
        _ => { println!("something else"); }
    }
}
```

Un `match` bras qui correspond à une variante d'union sans spécifier de valeur réussira toujours. Le code suivant provoquera un comportement indéfini si le dernier champ écrit de `u` était `u.i` :

```
// Undefined behavior!
unsafe {
    match u {
        FloatOrInt { f } => { println!("float {}", f) },
        // warning: unreachable pattern
        FloatOrInt { i } => { println!("int {}", i) }
    }
}
```

Syndicats d'emprunt

Empruntun champ d'une union emprunte toute l'union. Cela signifie que, conformément aux règles d'emprunt normales, emprunter un champ comme mutable exclut tout emprunt supplémentaire sur celui-ci ou sur d'autres champs, et emprunter un champ comme immuable signifie qu'il ne peut y avoir d'emprunt mutable sur aucun champ.

Comme nous le verrons dans le chapitre suivant, Rust vous aide à construire des interfaces sûres non seulement pour votre propre code non sûr, mais aussi pour du code écrit dans d'autres langages.. Unsafe est, comme son nom l'indique, lourd, mais utilisé avec précaution, il peut vous permettre de créer un code hautement performant qui conserve les garanties dont bénéficient les programmeurs Rust.

[1] Eh bien, c'est un classique d'où nous venons.

[2] Il existe de meilleures façons de gérer cela en utilisant le `RawVec` type du crate interne au compilateur `alloc`, mais ce crate est toujours instable.