Chapitre 22. Code dangereux

> *Que personne ne pense à moi que je suis humble, faible ou passif;*
> *Faites-leur comprendre que je suis d'un autre genre :*
> *dangereux pour mes ennemis, loyal envers mes amis.*
> *À une telle vie appartient la gloire.*
>
> —Euripide, *Médée*

La joie secrète de la programmation de systèmes est que, sous chaque langage sûr et abstraction soigneusement conçue se trouve un maelström tourbillonnant de langage machine extrêmement dangereux et de bricolage de bits. Vous pouvez également 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 entièrement automatiques, grâce aux types, aux durées de vie, aux vérifications des limites, etc. Mais ce genre de raisonnement automatisé a ses limites; il existe de nombreuses techniques précieuses que Rust ne peut pas reconnaître comme sûres.

Le code dangereux vous permet de dire à Rust : « J'opte pour l'utilisation de fonctionnalités dont vous ne pouvez pas garantir la sécurité. » En marquant un bloc ou une fonction comme dangereux, vous acquérez la possibilité d'appeler des fonctions dans la bibliothèque standard, de déréférencer des pointeurs dangereux et d'appeler des fonctions écrites dans d'autres langages comme C et C ++, entre autres pouvoirs. Les autres contrôles de sécurité de Rust s'appliquent toujours: les contrôles de type, les contrôles de durée de vie et les contrôles de limites sur les indices se produisent tous normalement. Le code dangereux permet simplement un petit ensemble de fonctionnalités supplémentaires. `unsafe`

Cette capacité à sortir des limites de Rust en toute sécurité 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 dangereux est ce qui permet au type de gérer efficacement sa mémoire tampon; le module pour parler au système d'exploitation; et les modules et pour fournir des primitives d'accès concurrentiel. `Vec` `std::io` `std::thread` `std::sync`

Ce chapitre couvre l'essentiel de l'utilisation de fonctionnalités dangereuses :

- Les blocs de Rust établissent la frontière entre le code Rust ordinaire et sûr et le code qui utilise des fonctionnalités dangereuses. `unsafe`
- Vous pouvez marquer les fonctions comme , alertant les appelants de la présence de contrats supplémentaires qu'ils doivent suivre pour éviter un comportement indéfini. `unsafe`
- Les pointeurs bruts et leurs méthodes permettent un accès sans contrainte à la mémoire et vous permettent de créer des structures de données que le système de type 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 beaucoup plus graves que le simple fait d'obtenir des résultats incorrects.
- Les caractéristiques dangereuses, analogues aux fonctions, imposent un contrat que chaque implémentation (plutôt que chaque appelant) doit suivre. `unsafe`

Dangereux de quoi?

Au début de ce livre, nous avons montré un programme C qui se bloque de manière surprenante car il ne suit pas l'une des règles prescrites par la norme 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 modifiable à la variable locale, la transforme en pointeur brut de type , puis utilise la méthode pour produire un pointeur trois mots plus loin dans la mémoire. Il se trouve que c'est là que l'adresse de retour de 's est stockée. Le programme écrase l'adresse de retour par une constante, de sorte que le retour de se comporte de manière surprenante. Ce qui rend ce crash possible, c'est l'utilisation incorrecte par le programme de fonctionnalités dangereuses , dans ce cas, la possibilité de déréférencer les pointeurs bruts. `a *mut usize offset main main`

Une fonctionnalité dangereuse est celle qui impose un *contrat* : des 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 et des contrôles de durée de vie habituels, imposant des règles supplémentaires spécifiques à cette caractéristique dangereuse. En règle générale, Rust lui-même ne connaît pas du tout le contrat; c'est juste expliqué dans la documentation de la fonctionnalité. Par exemple, le type de pointeur brut a un contrat vous interdisant de déréférencer un pointeur qui a été avancé au-delà de la fin de son référent d'origine. L'expression de 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 dangereuses, vous, en tant que programmeur, assumez la responsabilité de vérifier que votre code respecte leurs contrats. `*ptr.offset(3) = ...`

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 le voulons dire ici, sauf si les conséquences possibles incluent un comportement indéfini. Un comportement indéfini est un comportement que Rust suppose fermement que votre code ne pourrait jamais présenter. Par exemple, Rust suppose que vous n'écraserez pas l'adresse de retour d'un appel de fonction par autre chose. Le code qui passe les contrôles de sécurité habituels de Rust et se 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 la moitié de votre contrat avec Rust, et Rust refuse d'en prédire les conséquences. Le dragage des messages d'erreur non pertinents des profondeurs des bibliothèques système et le plantage sont une conséquence possible; confier 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 fonction ne revient jamais (peut-être qu'elle appelle à mettre fin au programme plus tôt), l'adresse de retour corrompue n'aura probablement pas d'importance. `main std::process::exit`

Vous ne pouvez utiliser que des fonctionnalités dangereuses au sein d'un bloc ou d'une 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 bloc ou une fonction, Rust s'assure que vous avez reconnu que votre code peut avoir des règles supplémentaires à suivre. `unsafe unsafe unsafe`

Blocs dangereux

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

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

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

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

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

- Vous pouvez appeler des fonctions. Chaque fonction doit spécifier son propre contrat, en fonction de son objet. `unsafe unsafe`

- Vous pouvez déréférencer les pointeurs bruts. Le code sécurisé peut passer 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 « Pointeurs bruts ».
- Vous pouvez accéder aux champs de s, dont le compilateur ne peut pas être sûr qu'ils contiennent des modèles de bits valides pour leurs types respectifs. `union`
- Vous pouvez accéder à des variables modifiables. Comme expliqué dans « Variables globales », Rust ne peut pas être sûr lorsque les threads utilisent des variables mutables, de sorte que leur contrat vous oblige à vous assurer que tous les accès sont correctement synchronisés. `static static`
- Vous pouvez accéder aux fonctions et variables déclarées via l'interface de fonction étrangère de Rust. Ceux-ci sont considérés même lorsqu'ils sont immuables, car ils sont visibles pour le code écrit dans d'autres langages qui peuvent ne pas respecter les règles de sécurité de Rust. `unsafe`

Restreindre les fonctionnalités dangereuses aux blocs ne vous empêche pas vraiment de faire ce que vous voulez. Il est parfaitement possible de simplement coller un 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 un code dont Rust ne peut garantir la sécurité : `unsafe unsafe`

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

Exemple : un type de chaîne ASCII efficace

Voici la définition de , un type de chaîne qui garantit que son contenu est toujours valide ASCII. Ce type utilise une fonctionnalité dangereuse pour fournir une conversion à coût nul en : `Ascii 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, NotAsciiErr
            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 conv
    // 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 type. Le type lui-même est marqué , pour le rendre visible à l'extérieur du module. Mais l'élément du type n'est *pas* public, de sorte que seul le module peut construire une valeur ou faire référence à son élément. Cela laisse le code du module en contrôle total sur ce qui peut ou non y apparaître. Tant que les constructeurs et les méthodes publics s'assurent que les valeurs nouvellement créées sont bien formées et le restent tout au long de leur vie, le reste du programme ne peut pas enfreindre cette règle. Et en effet, le constructeur public vérifie soigneusement le vecteur qui lui est donné avant d'accepter de construire un à partir de celui-ci. 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 valeurs contiennent toujours du texte ASCII approprié, tout comme les méthodes d'a garantissent que son contenu reste bien formé UTF-
8. `Ascii pub my_ascii Vec<u8> my_ascii Ascii Ascii Ascii::fro m_bytes Ascii Ascii String`

Cet arrangement nous permet de mettre en œuvre très efficacement. La fonction dangereuse prend un vecteur octet et en construit un à partir de celui-ci 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 appliquées par le type sont exactement ce dont nous avons besoin pour satisfaire 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-jacent d'un 'est immédiatement prêt à servir de tampon de
.. `From<Ascii> String String::from_utf8_unchecked String Asc ii from_utf8_unchecked Ascii Vec<u8> String`

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 bloc n'est requis pour utiliser . Nous avons mis en place une interface sécurisée utilisant des opérations dangereuses et nous nous sommes arrangés pour respecter leurs contrats en fonction uniquement du code du module, et non du comportement de ses utilisateurs. `unsafe Ascii`

An n'est rien de plus qu'un wrapper autour d'un , 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 commun dans Rust. Le propre type de Rust est défini exactement de la même manière, sauf que son contenu est limité à UTF-8, pas ASCII. En fait, voici la définition de la bibliothèque standard: `Ascii Vec<u8> String String`

```
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, de sorte que la construction d'un nouveau type ne nécessite aucune instruction de la machine. Dans , l'expression considère simplement que la représentation de 'a maintenant une valeur. De même, ne nécessite probablement aucune instruction machine lorsqu'il est inséré: le est maintenant considéré comme un `.Ascii::from_bytes Ascii(bytes) Vec<u8> Ascii String::from_utf8_unchecked Vec<u8> String`

Fonctions dangereuses

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

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

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

```
// 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` directl
    /// rather than a `Result<Ascii, NotAsciiError>` as the `from_byte
    /// constructor does.
    ///
    /// # Safety
    ///
    /// The caller must ensure that `bytes` contains only ASCII
    /// characters: bytes no greater than 0x7f. Otherwise, the effect
    /// undefined.
    pub unsafe fn from_bytes_unchecked(bytes: Vec<u8>) -> Ascii {
        Ascii(bytes)
    }
}
```

Vraisemblablement, l'appel de code 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 insiste pour être effectuée serait une perte de temps, et l'appelant devrait écrire du code pour gérer des résultats dont il sait qu'ils ne se produiront jamais. permet à un tel appelant d'éviter les vérifications et la gestion des erreurs. `Ascii::from_bytes_unchecked Ascii::from_bytes Err As cii::from_bytes_unchecked`

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

Pas tout à fait : remplit ses obligations en les transmettant à son appelant via son contrat. La présence de ce contrat est ce qui rend correct de marquer cette fonction : 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. `from_bytes_unchecked unsafe`

Pouvez-vous vraiment provoquer un comportement indéfini en rompant le contrat de ? Oui. Vous pouvez construire un UTF-8 mal formé comme

suit : `Ascii::from_bytes_unchecked String`

```
// 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 pro
// a `char` that is not a valid Unicode code point. That's undefined
// behavior, so the language doesn't say how this assertion should beh
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 chiffres nous semblent égaux, mais ce n'est pas la faute de Rust; c'est la faute du bloc précédent. Quand nous disons qu'un comportement indéfini conduit à des résultats imprévisibles, c'est le genre de chose que nous voulons dire. `unsafe`

Ceci illustre deux faits critiques sur les bogues et le code dangereux :

- *Les bogues qui se produisent avant le blocage `dangereux` peuvent briser les contrats.* Le fait qu'un bloc provoque un comportement non défini peut dépendre non seulement du code du bloc lui-même, mais également du code qui fournit les valeurs sur lesquelles il opère. Tout ce sur quoi votre code s'appuie pour satisfaire les contrats est essentiel pour la sécurité. La conversion de à basée sur n'est bien définie que si le reste du module maintient correctement les invariants de `.unsafe unsafe Ascii String String::from_utf8_unchecked As cii`

- *Les conséquences de la rupture d'un contrat peuvent apparaître après que vous ayez quitté le bloc* `dangereux`. Le comportement indéfini courtisé par le non-respect du contrat d'une fonctionnalité dangereuse ne se produit souvent pas dans le bloc lui-même. La construction d'un faux comme indiqué précédemment peut ne causer de problèmes que beaucoup plus tard dans l'exécution du programme. `unsafe String`

Essentiellement, le vérificateur de type, le vérificateur d'emprunt et d'autres contrôles statiques de Rust inspectent votre programme et tentent de construire la preuve qu'il ne peut pas présenter un comportement indéfini. Lorsque Rust compile votre programme avec succès, cela signifie qu'il a réussi à prouver le son de votre code. Un bloc est une lacune dans cette preuve : « Ce code », dites-vous à Rust, « va bien, croyez-moi. » La véracité de votre affirmation peut dépendre de n'importe quelle partie du programme qui influence ce qui se passe dans le bloc, et les conséquences d'être faux peuvent apparaître n'importe où influencées par le bloc. Écrire le mot-clé revient à rappeler que vous ne bénéficiez pas pleinement des contrôles de sécurité de la
langue. `unsafe unsafe unsafe unsafe`

Si vous avez le choix, vous devriez naturellement préférer créer des interfaces sûres, 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 non défini. Même si votre implémentation utilise des fonctionnalités dangereuses, 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 transférer les 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 devez déduire les règles vous-même, en fonction de votre expérience et de votre connaissance du comportement du code. Si vous vous êtes déjà demandé si ce que vous faites avec une API C ou C ++ est OK, alors vous savez à quoi cela ressemble.

Bloc dangereux ou fonction dangereuse?

Vous vous demandez peut-être s'il faut utiliser un bloc ou simplement marquer toute la fonction comme dangereuse. L'approche que nous recommandons est de prendre d'abord une décision concernant la fonction: `unsafe`

- S'il est possible d'abuser de la fonction d'une manière qui compile correctement mais provoque toujours un comportement indéfini, vous devez la marquer comme dangereuse. 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 bien tapé ne peut provoquer un comportement indéfini. Il ne doit pas être marqué . `unsafe`

La question de savoir si la fonction utilise des caractéristiques dangereuses dans son corps n'est pas pertinente; ce qui compte, c'est la présence d'un contrat. Auparavant, nous avons montré une fonction dangereuse qui n'utilise aucune fonctionnalité dangereuse et une fonction sécurisée qui utilise des fonctionnalités dangereuses.

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

Comportement non défini

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 étrange tournure de phrase, 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 établir les obligations d'un code dangereux?

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 signifie de dire que deux programmes dans des langues aussi complètement différentes 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 lorsqu'ils sont exécutés : ils font 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 pouvez pas dire 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 , nous pouvons voir qu'il ne reçoit qu'une référence partagée à , de sorte que l'appel ne peut pas changer la valeur de . Puisque la valeur transmise à sera toujours , Rust peut traduire ce code en langage machine comme si nous avions écrit : `very_trustworthy i i println! 1000`

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

Cette version transformée a le même comportement visible que l'original, et c'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 elles étaient définies comme suit ? `very_trustworthy`

```
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 pour les références partagées : il change la valeur de en , même s'il doit être gelé car il est emprunté pour le partage.

En conséquence, la transformation que nous avons faite à l'appelant a maintenant un effet très visible: si Rust transforme le code, le programme imprime; s'il laisse le code seul et utilise la nouvelle valeur de , il imprime . Enfreindre les règles pour les références partagées signifie que les références partagées ne se comporteront pas comme prévu chez leurs appelants. `i 20 i 1000 i 2000 very_trustworthy`

Ce genre de problème se pose avec presque tous les types de transformation que Rust pourrait tenter. Même l'insertion d'une fonction dans son site d'appel suppose, entre autres, que lorsque l'appelé se termine, le flux de contrôle retourne 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 en un programme préserve sa signification à moins qu'il ne puisse faire confiance aux caractéristiques fondamentales du langage pour se comporter comme prévu. Et qu'ils le fassent ou non peut dépendre non seulement du code à portée de main, mais d'autres parties potentiellement éloignées du programme. Afin de 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 Rust pour les programmes bien élevés:

- Le programme ne doit pas lire la mémoire non initialisée.
- Le programme ne doit pas créer de valeurs primitives non valides :
  - Références, zones ou pointeurs qui sont `fn null`
  - `bool` valeurs qui ne sont pas a ou `0 1`
  - `enum` valeurs avec des valeurs discriminantes non valides
  - `char` valeurs qui ne sont pas valides, points de code Unicode non substitution
  - `str` valeurs qui ne sont pas bien formées UTF-8
  - Pointeurs de graisse avec des vtables/longueurs de tranche non valides
  - Toute valeur du type « jamais », écrite , pour les fonctions qui ne renvoient pas `!`
- Les règles de référence expliquées au chapitre 5 doivent être suivies. 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 null, mal alignés ou pendants.

- Le programme ne doit pas utiliser de pointeur pour accéder à la mémoire en dehors de l'allocation à laquelle le pointeur est associé. Nous expliquerons cette règle en détail dans « Déréférencement des pointeurs bruts en toute sécurité ».
- Le programme doit être exempt de courses de données. Une course de données se produit lorsque deux threads accèdent au même emplacement de mémoire sans synchronisation et qu'au moins un des accès est une écriture.
- Le programme ne doit pas se dérouler sur un appel effectué à partir d'une autre langue, via l'interface de fonction étrangère, comme expliqué dans « Déroulement ».
- Le programme doit être conforme aux contrats des fonctions standard de la bibliothèque.

Comme nous n'avons pas encore de modèle complet de la sémantique de Rust pour le code, cette liste évoluera probablement avec le temps, mais celles-ci resteront probablement interdites. `unsafe`

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 UTF-8 mal formé à , peut-être que 2097151 n'est pas si égal à 2097151 après tout. `String::from_utf8_unchecked`

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

En C et C++, le fait que votre programme 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 respectés qui maintiennent leur code à des normes élevées présentent un comportement indéfini dans la pratique.

Traits dangereux

Un *trait dangereux est un trait* qui a un contrat que Rust ne peut pas vérifier ou appliquer que les implémenteurs doivent satisfaire pour éviter un

comportement indéfini. Pour implémenter un caractère dangereux, vous devez marquer l'implémentation comme dangereuse. 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 dangereux est généralement une fonction qui utilise elle-même des entités dangereuses et satisfait leurs contrats uniquement en fonction du contrat du caractère dangereux. Une implémentation incorrecte du trait pourrait entraîner un comportement indéfini d'une telle fonction.

`std::marker::Send` et sont les exemples classiques de traits dangereux. Ces traits ne définissent aucune méthode, ils sont donc triviaux à mettre en œuvre pour tout type que vous aimez. Mais ils ont des contrats : exige que les implémenteurs soient en sécurité pour passer à un autre thread, et exige qu'ils soient sûrs à partager entre les threads via des références partagées. La mise en œuvre pour un type inapproprié, par exemple, ne serait plus à l'abri des courses de données. `std::marker::Sync Send Sync Send std::sync::Mutex`

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

`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 type : `Zeroable`

```
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 avec la capacité requise, puis appelle à remplir le tampon inoccupé avec des zéros. (La fonction traite comme un nombre d'éléments, et non comme un nombre d'octets, de sorte que cet appel remplit la totalité de la mémoire tampon.) La méthode d'un vecteur change de longueur sans rien faire au tampon ; ceci n'est pas sûr, car vous devez vous assurer que l'espace tampon nouvellement fermé contient effectivement des valeurs de type correctement initialisées . Mais c'est exactement ce que la liaison établit : un bloc de zéro octet représente une valeur valide. Notre utilisation de était sûre. `Vec write_bytes write_byte len T set_len T T: Zeroable T set_len`

Ici, nous l'avons mis à profit:

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

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

```
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 est censé signifier, il ne peut donc pas dire quand il est mis en œuvre pour un type inapproprié. Comme pour toute autre fonctionnalité dangereuse, c'est à vous de comprendre et d'adhérer au contrat d'un trait dangereux. `Zeroable`

Notez que le code dangereux ne doit pas dépendre de caractéristiques ordinaires et sûres correctement implémentées. Par exemple, supposons qu'il y ait une implémentation du trait qui renvoie simplement une valeur de hachage aléatoire, sans rapport avec les valeurs hachées. Le trait exige que le hachage des mêmes bits deux fois produise la même valeur de hachage, mais cette implémentation ne répond pas à cette exigence ; c'est tout simplement incorrect. Mais parce que ce n'est pas un trait dangereux, le code dangereux ne doit pas présenter un comportement indéfini lorsqu'il utilise ce hasher. Le type est soigneusement écrit pour respecter les contrats des fonctionnalités dangereuses qu'il utilise, quel que soit le comportement du hasher. 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. `std::hash::Hasher Hasher std::collections::HashMap`

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 bloc. `unsafe`

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

ces langages.

Il existe deux types de pointeurs bruts :

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

(Il n'y a pas de type simple ; vous devez toujours spécifier l'un ou l'autre .) `*T const 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 bruts peuvent être nuls, comme en C ou en C++ : `NULL nullptr`

```
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 blocs : la création de pointeurs bruts, leur transmission et leur comparaison sont tous sûrs. Seul le déréférencement d'un pointeur brut n'est pas sûr. `unsafe`

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

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

- L'opérateur ne déréférence pas implicitement un pointeur brut ; vous devez écrire ou `. . (*raw).field (*raw).method(...)`
- Les pointeurs bruts n'implémentent pas , de sorte que les coercitions deref ne s'appliquent pas à eux. `Deref`
- Les opérateurs aiment et comparent les pointeurs bruts en tant qu'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, et non la valeur de son référent. `== <`
- Les traits de mise en forme tels que suivre les références automatiquement, mais ne gèrent pas du tout les pointeurs bruts. Les exceptions sont et , qui affichent les pointeurs bruts sous forme d'adresses hexadécimales, sans les déréférencerencement. `std::fmt::Display std::fmt::Debug std::fmt::Pointer`

Contrairement à l'opérateur en C et C++, Rust ne gère pas les pointeurs bruts, mais vous pouvez effectuer l'arithmétique des pointeurs via leurs méthodes et méthodes, ou les méthodes plus pratiques , , et . 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 de mémoire (la même, par exemple): `+ + offset wrapping_offset add 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 et ; il suffit de spécifier le type. Rust contraint implicitement les références à des pointeurs bruts (mais pas l'inverse, bien sûr). `first last`

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

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

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

Soyez très prudent lorsque vous faites cela: une référence produite de cette manière a une durée de vie sans contrainte: il n'y a pas de limite à la durée de vie, car le pointeur brut ne donne rien à Rust sur lequel fonder une telle décision. Dans « A Safe Interface to libgit2 » plus loin dans ce chapitre, nous montrons plusieurs exemples de la façon de limiter correctement les durées de vie.

De nombreux types ont et méthodes qui renvoient un pointeur brut à 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 , , et avoir et des fonctions qui convertissent vers et à partir de pointeurs bruts. Certains contrats de ces méthodes imposent des exigences surprenantes, alors vérifiez leur documentation avant de les
utiliser. `as_ptr as_mut_ptr Box Rc Arc into_raw from_raw`

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 d'un pointeur en premier lieu. « Exemple: RefWithFlag » utilise des pointeurs bruts de cette façon.

Contrairement aux références, les pointeurs bruts ne sont ni ni . 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 bloc pour les déréférencer. Mais étant donné les rôles que jouent généralement les pointeurs bruts, les concepteurs de langage ont considéré ce comportement comme la valeur par défaut la plus utile. Nous avons déjà discuté de la façon de mettre en œuvre et vous-même dans « Traits dangereux ». `Send` `Sync` `unsafe` `Send` `Sync`

Déréférencement des pointeurs bruts en toute sécurité

Voici quelques directives de bon sens pour utiliser les pointeurs bruts en toute sécurité :

- Le déréférencement de pointeurs nuls ou de pointeurs pendants est un comportement indéfini, tout comme la mémoire non initialisée ou les valeurs qui sont sorties de la portée.
- Le déréférencement des pointeurs qui ne sont pas correctement alignés pour leur type de référent n'est pas un comportement défini.
- Vous ne pouvez emprunter des valeurs à partir d'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 enfreindre 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 d'un produit produit un point de code Unicode approprié et non aurigué. `*const char`
- Vous pouvez utiliser les méthodes et sur les pointeurs bruts uniquement pour pointer vers des octets dans la variable ou le bloc de mémoire alloué au tas auquel le pointeur d'origine faisait référence, ou vers le premier octet au-delà d'une telle région. `offset` `wrapping_offset`
  Si vous effectuez de l'arithmétique de pointeur en convertissant le pointeur en entier, en effectuant de l'arithmétique sur l'entier, puis en

le reconvertissant en pointeur, le résultat doit être un pointeur que les règles de la méthode vous auraient permis de produire. `offset`

- 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 un pointage vers un octet de a , vous ne pouvez stocker que des valeurs dans ce qui laisse le maintien UTF-8 bien formé. `*mut u8 String u8 String`

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 de ne pas violer les invariants des types doit être claire. De nombreux types standard de Rust utilisent du code dangereux dans leur mise en œuvre, 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 conduire à 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 exemple de la façon de prendre un classique[1] hack au niveau des bits rendu possible par des pointeurs bruts et l'envelopper comme un type Rust complètement sûr. Ce module définit un type, , qui contient à la fois a et a , comme le tuple et qui parvient tout de même à n'occuper qu'un seul mot machine au lieu de deux. Ce type de technique est régulièrement utilisé dans les garbage collectors 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 : `RefWithFlag<'a, T> &'a T bool (&'a T, bool)`

```
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 who
        /// 2⁰-bit you didn't want to steal, well, now you can do it safel
        /// ("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 nul, nous pouvons y stocker autre chose et ensuite reconstruire de manière fiable l'adresse d'origine simplement en masquant le bit inférieur. Tous les types ne sont pas admissibles; par exemple, les types et peuvent être placés à n'importe quelle adresse. Mais nous pouvons vérifier l'alignement du type sur la construction et refuser les types qui ne fonctionneront pas. `u8` `(bool, [i8; 2])`

Vous pouvez utiliser comme ceci: `RefWithFlag`

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

La méthode extrait le composant d'un fichier . C'est simple: il suffit de masquer le bit inférieur et de vérifier s'il n'est pas
nul. `get_flag bool RefWithFlag`

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

En mémoire, a ressemble à un : puisqu'il s'agit d'un type de taille nulle, le champ ne prend aucune place dans la structure. Mais il est nécessaire pour Rust de savoir comment traiter les durées de vie dans le code qui utilise . Imaginez à quoi ressemblerait le type sans le champ
: `RefWithFlag usize PhantomData behaves_like PhantomData Ref WithFlag behaves_like`

```
// 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 indications pendantes. La structure doit respecter les restrictions qui s'appliquent à ses champs. Cela s'applique certainement à : dans l'exemple de code que nous venons de regarder, ne doit pas survivre, puisqu'il renvoie une référence à celui-ci. Mais notre type réduit ne contient aucune référence et n'utilise jamais son paramètre de durée de vie. C'est juste un fichier . Comment Rust devrait-il savoir que des restrictions s'appliquent à la durée de vie de ? L'inclusion d'un champ indique à Rust de traiter *comme s'il* contenait un , sans affecter réellement la représentation de la structure. `RefWithFlag flagged vec flagged.get_ref() RefWithFlag 'a usize flagged PhantomData<&'a T> RefWithFlag<'a, T> &'a T`

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

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

Pointeurs Nullable

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

Il existe plusieurs façons de vérifier si un pointeur brut est nul. La plus simple est la méthode, mais la méthode peut être plus pratique : elle prend un pointeur et renvoie un , transformant un pointeur null en un . De même, la méthode convertit les pointeurs en valeurs. `is_null as_ref *const T Option<&'a T> None as_mut *mut T Option<&'a mut T>`

Tailles de type et alignements

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

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

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

L'alignement de tout 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 ne nécessite que cinq octets, est , parce que est . Cela garantit que si vous avez un tableau, la taille du type d'élément reflète toujours l'espacement entre un élément et le suivant. `(f32, u8) size_of::<(f32, u8)>() 8 align_of::<(f32, u8)>() 4`

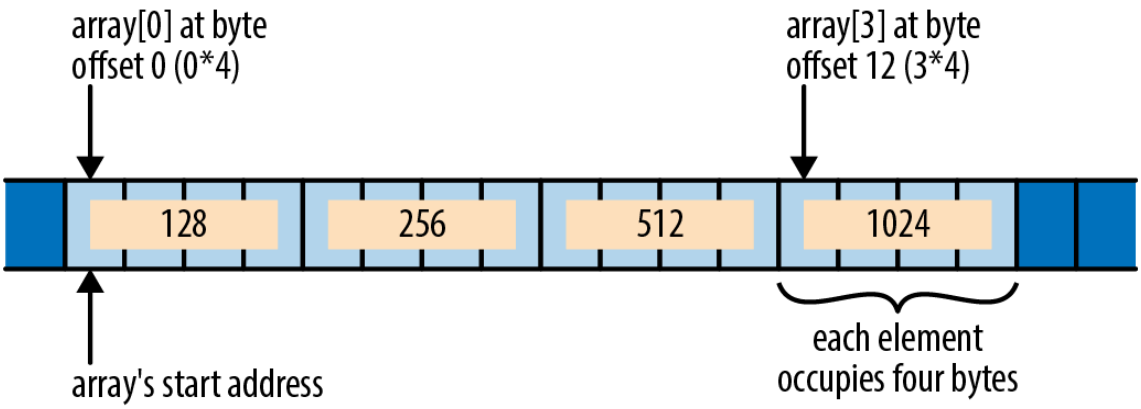
Pour les types non dimensionnés, la taille et l'alignement dépendent de la valeur à portée de main. Étant donné une référence à une valeur non dimensionnée, les fonctions et renvoient la taille et l'alignement de la valeur. Ces fonctions peuvent fonctionner sur des références à des types à la fois et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

```
// 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

Rust présente 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 des octets, le ème élément commence par le ème octet. `size i i * size`



Graphique 22-1. Une baie 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 , alors un pointeur brut vers le th élément est inférieur à un pointeur brut vers le ème élément. Cela rend les pointeurs bruts utiles en tant que limites sur les traversées de tableau. En fait, l'itérateur simple de la bibliothèque standard sur une tranche a été défini à l'origine comme suit : `i < j i j`

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

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

Une autre conséquence intéressante de la disposition du tableau: si est un pointeur brut vers le ème élément d'un tableau, alors est un pointeur brut vers le ème élément. Sa définition est équivalente à ceci : `element_ptr *const T *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 fonction renvoie la taille du type en octets. Étant donné qu'il est, par définition, assez grand pour contenir une adresse, vous pouvez convertir le pointeur de base en , faire de l'arithmétique sur cette valeur, puis re-convertir le résultat en pointeur. `std::mem::size_of:: <T> T isize isize`

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

Toutefois, il s'agit d'un comportement indéfini à utiliser pour produire un 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 quand est positif et que quand est négatif. Cette hypothèse semble sûre, mais elle peut ne pas tenir si l'arithmétique dans déborde une valeur. Si est contraint de rester dans le même tableau que , aucun débordement ne peut se produire : après tout, le tableau lui-même ne dé-

passe pas les limites de l'espace d'adressage. (Pour sécuriser les pointeurs vers le premier octet après la fin, Rust ne place jamais de valeurs à l'extrémité supérieure de l'espace d'adressage.) `offset ptr.offset(i) > ptr i ptr.offset(i) < ptr i offset isize i ptr`

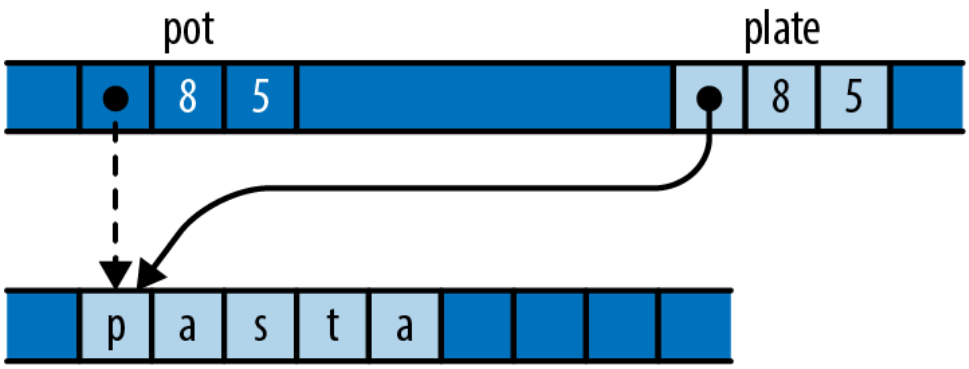
Si vous devez décaler les pointeurs au-delà des limites du tableau auquel ils sont associés, vous pouvez utiliser la méthode. Ceci est équivalent à , mais Rust ne fait aucune hypothèse sur l'ordre relatif de et lui-même. Bien sûr, vous ne pouvez toujours pas déréférencer ces pointeurs à moins qu'ils ne tombent dans le tableau. `wrapping_offset offset ptr.wrapping_offset(i) ptr`

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.



Graphique 22-2. Déplacement d'une chaîne d'une variable locale à une autre

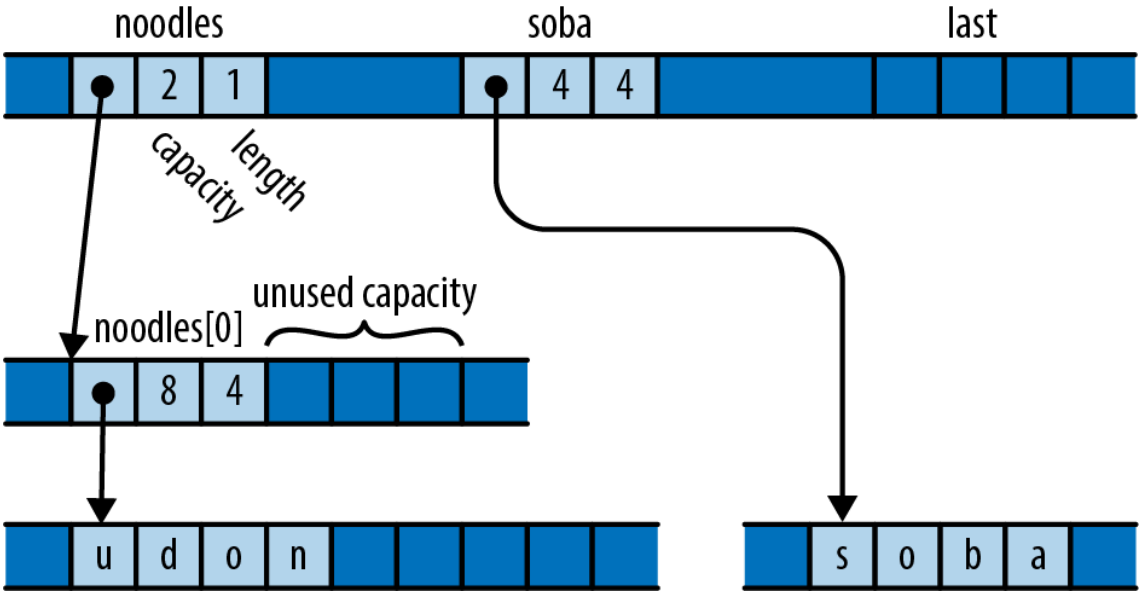
Après l'affectation, n'est pas initialisé et est le propriétaire de la chaîne. `pot plate`

Au niveau de la machine, il n'est pas spécifié ce qu'un déplacement fait à la source, mais dans la pratique, il ne fait généralement rien du tout. L'affectation laisse probablement toujours un pointeur, une capacité et une longueur pour la chaîne. Naturellement, il serait désastreux de traiter cela comme une valeur réelle, et Rust s'assure que vous ne le faites pas. `pot`

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.

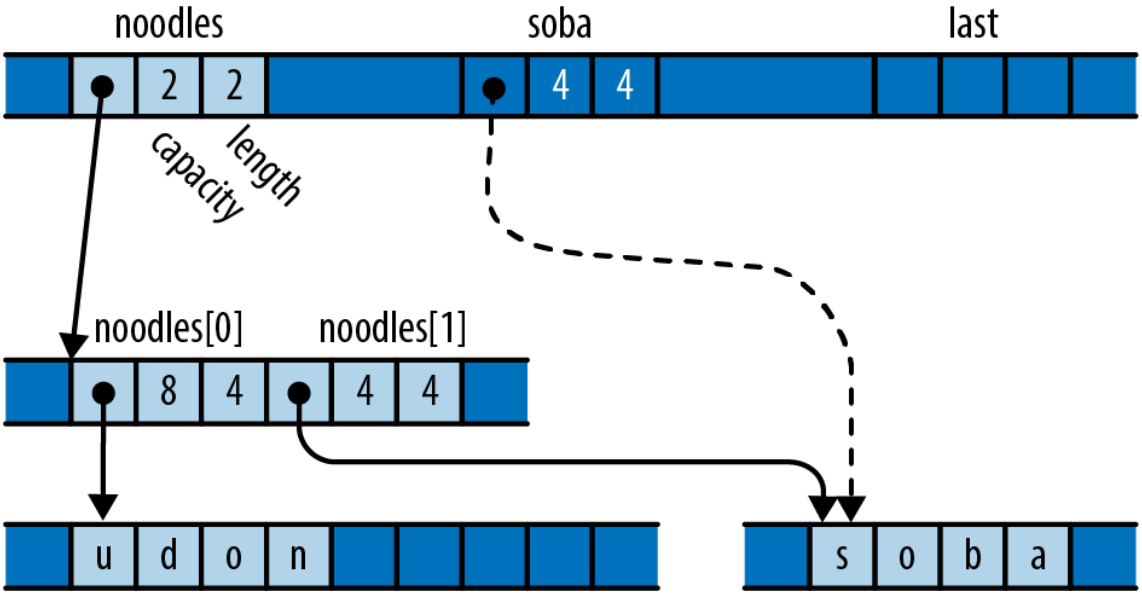


Graphique 22-3. Un vecteur avec une capacité inutilisée non initialisée

Le vecteur a la capacité de réserve pour contenir un élément de plus, mais son contenu est indésirable, probablement quelle que soit la mémoire détenue précédemment. 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.



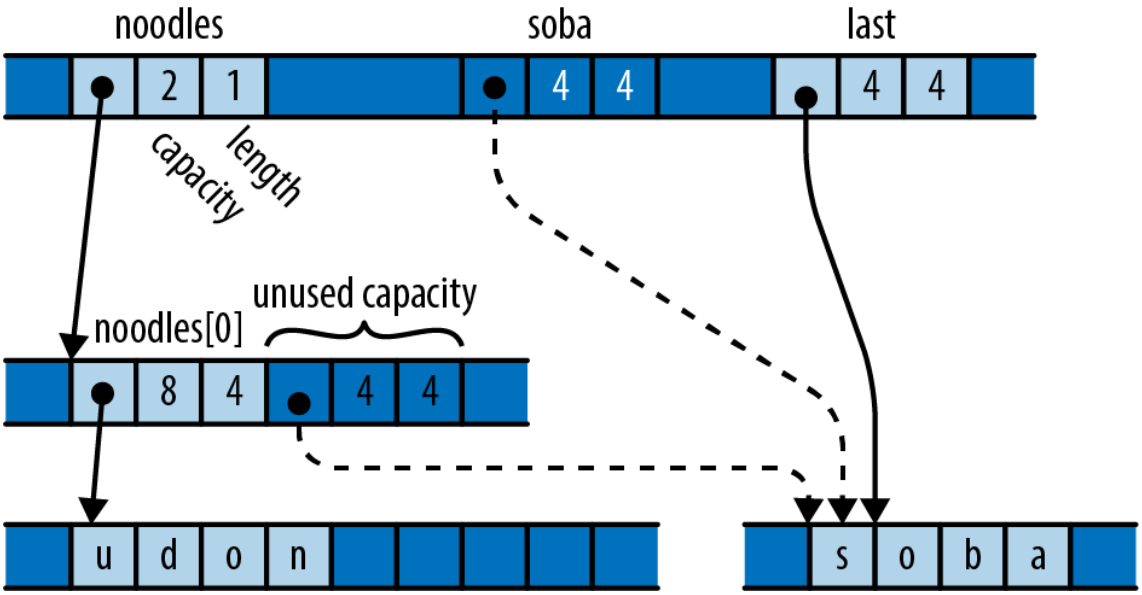
Graphique 22-4. Après avoir poussé la valeur de 'sur le vecteur `soba`

Le vecteur a initialisé son espace vide pour posséder la chaîne et incrémenté sa longueur pour la marquer comme un nouvel élément actif. Le vecteur est maintenant le propriétaire de la chaîne ; vous pouvez vous référer à son deuxième élément, et la suppression du vecteur libérerait les deux chaînes. Et n'est plus initialisé. `soba`

Enfin, considérons ce qui se passe lorsque nous faisons apparaître une valeur à partir du vecteur :

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

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



Graphique 22-5. Après avoir fait apparaître un élément du vecteur dans `last`

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

tialisé. `last`

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

La véritable définition d'une valeur initialisée est celle qui est *traitée comme vivante*. L'écriture sur 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 que, après un déplacement, la source n'est plus traitée comme vivante, alors qu'après une copie, la source et la destination sont en direct.

Rust suit les variables locales qui sont en ligne au moment de la compilation et vous empêche d'utiliser des variables dont les valeurs ont été déplacées ailleurs. Des types comme , , , et ainsi de suite suivent leurs tampons dynamiquement. Si vous implémentez un type qui gère sa propre mémoire, vous devrez faire de même. `Vec HashMap Box`

Rust fournit deux opérations essentielles pour la mise en œuvre de tels types:

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

Déplace une valeur hors de l'emplacement pointe vers, transférant la propriété à l'appelant. L'argument doit être un pointeur brut, où est un type de taille. Après avoir appelé cette fonction, le contenu de n'est pas affecté, mais sauf si c'est , vous devez vous assurer que votre programme les traite comme de la mémoire non initialisée. `src src *const T T *src T Copy`

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

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

Se déplace dans l'emplacement pointe vers, qui doit être de la mémoire non initialisée avant l'appel. Le référent est maintenant pro-

priétaire de la valeur. Ici, doit être un pointeur brut et une valeur, où est un type de taille. `value dest dest *mut T value T T`

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

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 de sécurité de Rust. Ils exigent tous que leurs référents soient initialisés à tout moment, de sorte que la transformation de la mémoire non initialisée en valeur, ou vice versa, est hors de leur portée. Les pointeurs bruts correspondent à la facture.

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

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

Déplace le tableau de valeurs en mémoire à partir de la mémoire à , comme si vous aviez écrit une boucle de et appelle pour les déplacer une à la fois. La mémoire de destination doit être non initialisée avant l'appel, puis la mémoire source n'est pas initialisée. Les arguments et doivent être des pointeurs bruts, et doivent être un `.count src dst read write src dest *const T *mut T count usize`

ptr. (dst, compte) `copy_to`

Une version plus pratique de cela déplace le tableau de valeurs en mémoire à partir de , plutôt que de prendre son point de départ comme argument. `copy count ptr dst`

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

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

`ptr.copy_to_nonoverlapping(dst, count)`

Une version plus pratique de , comme . `copy_nonoverlapping copy_to`

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

`read_unaligned`, `write_unaligned`

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

`read_volatile`, `write_volatile`

Ces fonctions sont l'équivalent de lectures et d'écritures volatiles en C ou C++.

Exemple : GapBuffer

Voici un exemple qui met à profit les fonctions de pointeur brutes que nous venons de décrire.

Supposons que vous écrivez un éditeur de texte et que vous recherchez un type pour représenter le texte. Vous pouvez choisir et utiliser les méthodes et pour insérer et supprimer des caractères au fur et à mesure que l'utilisateur tape. Mais s'ils modifient du texte au début d'un fichier volumineux, ces méthodes peuvent être coûteuses : l'insertion d'un nouveau caractère implique de déplacer tout le reste de la chaîne vers la droite dans la mémoire, et la suppression déplace tout cela vers la gauche. Vous aimeriez que ces opérations courantes soient moins chères. `String insert remove`

L'éditeur de texte Emacs utilise une structure de données simple appelée *tampon d'écart* qui peut insérer et supprimer des caractères en temps constant. Alors qu'un garde toute sa capacité inutilisée à la fin du texte, ce qui fait et bon marché, un tampon d'écart maintient sa capacité inutilisée au milieu du texte, au moment où l'édition a lieu. Cette capacité inutilisée s'appelle *l'écart*. Insérer ou supprimer des éléments à l'écart est bon marché: il vous suffit de réduire ou d'agrandir l'espace selon vos besoins. Vous pouvez déplacer l'espace vers n'importe quel emplacement 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. `String push pop`

Bien que l'insertion et la suppression dans un tampon d'écart soient rapides, changer la position à laquelle elles ont lieu implique de déplacer l'écart vers la nouvelle position. Le déplacement des éléments nécessite un temps proportionnel à la distance déplacée. Heureusement, l'activité d'édition typique consiste à apporter un tas de modifications dans un

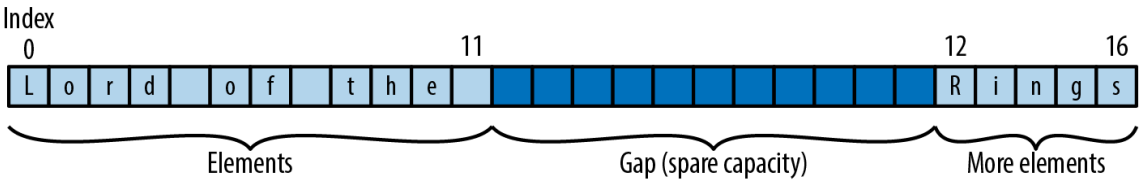
quartier de la mémoire tampon avant de partir et de jouer avec du texte ailleurs.

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

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

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

Après avoir exécuté ce code, la mémoire tampon ressemble à ce qui est illustré à la figure 22-6.

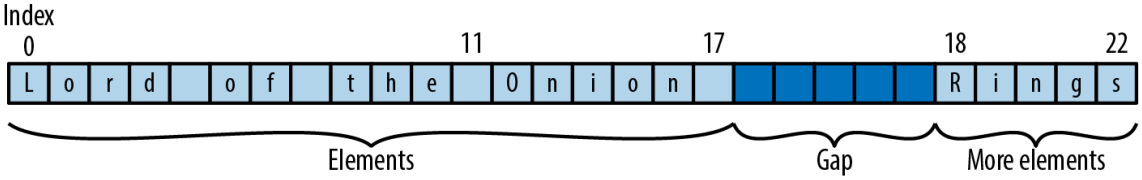


Graphique 22-6. Un tampon d'espace contenant du texte

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.



Graphique 22-7. Un tampon d'espace contenant un peu plus de texte

Voici notre type: `GapBuffer`

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

```
pub struct GapBuffer<T> {
    // Storage for elements. This has the capacity we need, but its le
    // always remains zero. GapBuffer puts its elements and the gap ir
    // `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 champ d'une manière étrange. `storage` [2] Il ne stocke jamais réellement d'éléments dans le vecteur , ou pas tout à fait. Il appelle simplement à obtenir un bloc de mémoire suffisamment grand pour contenir des valeurs, obtient des pointeurs bruts vers cette mémoire via le vecteur et les méthodes, puis utilise le tampon directement à ses propres fins. La longueur du vecteur reste toujours nulle. Lorsque le est abandonné, le 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 que veut; il a sa propre implémentation qui sait où se trouvent les éléments vivants et les laisse tomber
correctement. `Vec::with_capacity(n) n as_ptr as_mut_ptr Vec V ec GapBuffer Drop`

`GapBuffer` les méthodes les plus simples sont ce à quoi vous 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 withou
    /// 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 de nombreuses fonctions suivantes pour avoir une méthode utilitaire qui renvoie un pointeur brut vers l'élément tampon à un index donné. Ceci étant Rust, nous finissons par avoir besoin d'une méthode pour les pointeurs et d'une pour . Contrairement aux méthodes précédentes, celles-ci ne sont pas publiques. Suite de ce bloc : `mut const impl`

```
/// Return a pointer to the `index`th element of the underlying storag
/// 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 underlyin
/// 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 indice donné, vous devez déterminer si l'indice tombe avant ou après l'écart et ajuster de manière appropriée :

```
/// Return the offset in the buffer of the `index`th element, taking
/// the gap into account. This does not check whether index is in rang
/// 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 effectuer des insertions et des suppressions dans une autre partie de la mémoire tampon, nous devons déplacer l'espace vers le nouvel emplacement. Déplacer l'espace vers la droite implique de déplacer les éléments vers la gauche, et vice versa, tout comme la bulle dans un niveau d'esprit se déplace dans une direction 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 méthode pour déplacer les éléments ; exige que la destination ne soit pas initialisée et laisse la source non initialisée. Les plages source et de destination peuvent se chevaucher, mais gèrent correctement ce cas. Étant donné que l'écart est une mémoire non initialisée avant l'appel et que la fonction ajuste la position de l'écart pour couvrir l'espace libéré par la copie, le contrat de la fonction est satisfait. `std::ptr::copy copy copy copy`

L'insertion et le retrait des éléments sont relativement simples. L'insertion prend plus d'un espace de l'espace pour le nouvel élément, tandis que la suppression déplace une valeur 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 les utilisations pour push et pour , utilise pour et pour . Et tout comme doit ajuster sa longueur pour maintenir la frontière entre les éléments initialisés et la capacité de réserve, ajuste son écart. `Vec std::ptr::write std::ptr::read pop GapBuffer write insert read remove Vec GapBuffer`

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

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

Le déplacement du nouveau vecteur dans laisse tomber l'ancien vecteur. Comme sa longueur est nulle, l'ancien vecteur croit qu'il n'a pas d'éléments à laisser tomber et libère simplement son tampon. Soigneusement, laisse sa source non initialisée, de sorte que l'ancien vecteur est correct dans cette croyance: tous les éléments sont maintenant la propriété du nouveau vecteur. `self.storage` `copy_nonoverlapping`

Enfin, nous devons nous assurer que la chute d'un laisse tomber tous ses éléments: `GapBuffer`

```
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 itérons donc sur chaque région et utilisons la fonction pour laisser tomber chacune d'elles. La fonction est un utilitaire qui se comporte comme , mais ne prend pas la peine de déplacer la valeur vers son appelant (et fonctionne donc sur les types non dimensionnés). Et tout comme dans , au moment où le vecteur est abandonné, son tampon n'est vraiment pas initialisé. `std::ptr::drop_in_place drop_in_place drop(std::ptr ::read(ptr)) enlarge_gap self.storage`

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

Sécurité de panique dans le code dangereux

Dans Rust, les paniques ne peuvent généralement pas provoquer un comportement indéfini; la macro n'est pas une fonctionnalité dangereuse. Mais lorsque vous décidez de travailler avec un code dangereux, la sécurité panique fait partie de votre travail. `panic!`

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

```
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)
}
```

Appel à déplacer l'élément immédiatement après l'espace hors de la mémoire tampon, laissant derrière lui un espace non initialisé. À ce stade, le est dans un état incohérent: nous avons brisé l'invariant que tous les éléments en dehors de l'écart doivent être initialisés. Heureusement, la déc-

laration suivante agrandit l'écart pour couvrir cet espace, de sorte qu'au moment où nous revenons, l'invariant tient à nouveau. `read GapBuffer`

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

Il est presque inévitable que les méthodes d'un type détendent momentanément les invariants du type pendant qu'ils font leur travail, puis remettent tout en place avant leur retour. Une méthode de panique au milieu pourrait couper court à 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 un comportement indéfini. Mais le code utilisant des fonctionnalités non sécurisées compte généralement sur ses invariants pour répondre aux contrats de ces fonctionnalités. Les invariants brisés conduisent à des contrats rompus, ce qui conduit à un comportement indéfini.

Lorsque vous travaillez avec des fonctionnalités dangereuses, vous devez prendre des précautions particulières pour identifier ces régions sensibles du code où les invariants sont temporairement détendus, et vous assurer qu'ils ne font rien qui pourrait paniquer.

Réinterprétation de la mémoire avec les unions

Rust fournit 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 de données destinées à l'un comme à l'autre entraînera probablement des absurdités.

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

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

Il s'agit d'une union avec deux champs, et . Ils peuvent être affectés à tout comme les champs d'une structure, mais lors de la construction d'une union, contrairement à une structure, vous devez en choisir exactement une. Là où les champs d'une struct se réfèrent à différentes positions en mémoire, les champs d'une union se réfèrent à différentes interprétations de la même séquence de bits. Affecter à un champ différent signifie simplement écraser tout ou partie de ces bits, conformément à un type approprié. Ici, fait référence à une seule plage de mémoire 32 bits, qui stocke d'abord codée sous la forme d'un simple entier, puis sous la forme d'un nombre à virgule flottante IEEE 754. Dès qu'elle est écrite, la valeur précédemment écrite dans le est écrasée : `f i one 1 1.0 f FloatOrInt`

```
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 elle n'est qu'un : `SmallOrLarge::s 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 enums, les syndicats n'ont pas d'étiquette. Le compilateur n'ajoute aucun bit supplémentaire pour distinguer les variantes. Il n'y a aucun moyen de dire au moment de l'exécution si a est destiné à être interprété comme un ou un , à moins que le programme n'ait un contexte supplémentaire. `SmallOrLarge u64 bool`

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

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 peut facilement être utilisé pour imprimer les bits individuels d'un nombre à virgule flottante, même s'il n'implémente pas le formateur
: `FloatOrInt f32 Binary`

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

Bien que ces exemples simples fonctionneront 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 définition indiquant au compilateur comment disposer les données en mémoire. L'ajout de l'attribut garantit que tous les champs commencent au décalage 0, plutôt que là où le compilateur le souhaite. 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 : `union #[repr(C)]`

```
#[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. Étant donné que les processeurs x86 sont peu endiens, l'ordre de ces octets est inversé ; l'octet le plus significatif n'est pas , mais . Normalement, ce n'est pas quelque chose que le code Rust doit gérer, mais parce que ce code fonctionne directement avec la représentation en mémoire du , ces détails de bas niveau deviennent importants. `bytes[0]` `bytes[7]` `i64`

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

Syndicats correspondants

La correspondance sur une union rust est similaire à la correspondance sur 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 bras qui correspond à une variante d'union sans spécifier de valeur réussira toujours. Le code suivant provoquera un comportement non défini si le dernier champ écrit de était : `match u u.i`

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

Syndicats emprunteurs

Emprunter un champ d'un syndicat emprunte l'ensemble du syndicat. Cela signifie que, conformément aux règles d'emprunt normales, emprunter un champ comme mutable empêche tout emprunt supplémentaire sur celui-ci ou sur d'autres champs, et emprunter un champ comme immuable signifie qu'il ne peut y avoir aucun emprunt mutable sur aucun champ.

Comme nous le verrons dans le chapitre suivant, Rust vous aide à créer des interfaces sécurisées non seulement pour votre propre code dangereux, mais également pour le code écrit dans d'autres langages. Dangereux est, comme son nom l'indique, lourd, mais utilisé avec soin, il peut vous permettre de construire un code hautement performant qui conserve les garanties dont jouissent 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 type de la caisse interne du compilateur, mais cette caisse est toujours instable. `RawVec alloc`