Chapitre 17. Chaînes et texte

> *La chaîne est une structure de données austère et partout où elle est transmise, il y a beaucoup de duplication de processus. C'est un véhicule parfait pour cacher des informations.*
>
> —Alan Perlis, épigramme #34

Nous avons utilisé les principaux types textuels de Rust, , et , tout au long du livre. Dans « String Types », nous avons décrit la syntaxe des littéraux de caractères et de chaînes et montré comment les chaînes sont représentées en mémoire. Dans ce chapitre, nous abordons la gestion du texte plus en détail. `String str char`

Dans ce chapitre :

- Nous vous donnons quelques informations sur Unicode qui devraient vous aider à comprendre la conception de la bibliothèque standard.
- Nous décrivons le type, représentant un seul point de code Unicode. `char`
- Nous décrivons les et types, représentant des séquences possédées et empruntées de caractères Unicode. Ceux-ci ont une grande variété de méthodes pour construire, rechercher, modifier et itérer sur leur contenu. `String str`
- Nous couvrons les fonctions de formatage de chaîne de Rust, comme les macros et les macros. Vous pouvez écrire vos propres macros qui fonctionnent avec des chaînes de mise en forme et les étendre pour prendre en charge vos propres types. `println! format!`
- Nous donnons un aperçu du support d'expression régulière de Rust.
- Enfin, nous expliquons pourquoi la normalisation Unicode est importante et montrons comment le faire dans Rust.

Un peu d'arrière-plan Unicode

Ce livre parle de Rust, pas d'Unicode, qui a déjà des livres entiers qui lui sont consacrés. Mais les types de caractères et de chaînes de Rust sont conçus autour d'Unicode. Voici quelques morceaux d'Unicode qui aident à expliquer Rust.

ASCII, Latin-1 et Unicode

Unicode et ASCII correspondent pour tous les points de code ASCII, de à : par exemple, les deux attribuent le caractère au point de code . De même, Unicode attribue les mêmes caractères que le jeu de caractères ISO/IEC 8859-1, un surensemble ASCII de huit bits à utiliser avec les langues d'Europe occidentale. Unicode appelle cette plage de points de code le *bloc de code Latin-1*, nous nous référerons donc à ISO/IEC 8859-1 par le nom plus évocateur *Latin-1*. `0 0x7f * 42 0 0xff`

Étant donné qu'Unicode est un sur-ensemble de Latin-1, la conversion de Latin-1 en Unicode ne nécessite même pas de table :

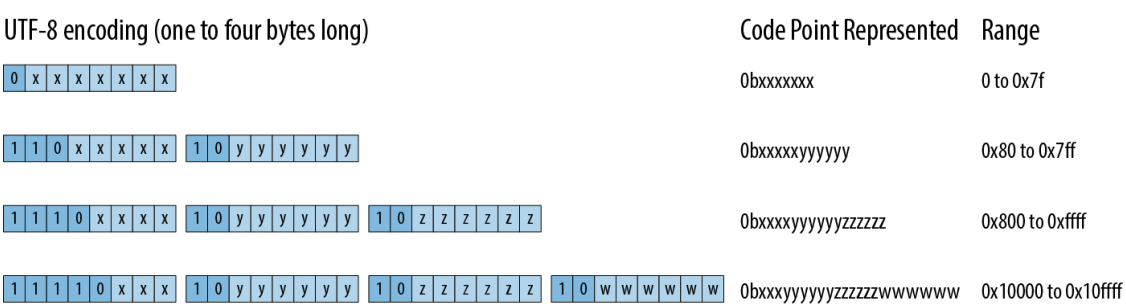
```
fn latin1_to_char(latin1: u8) -> char {
    latin1 as char
}
```

La conversion inverse est également triviale, en supposant que les points de code se situent dans la plage Latin-1:

```
fn char_to_latin1(c: char) -> Option<u8> {
    if c as u32 <= 0xff {
        Some(c as u8)
    } else {
        None
    }
}
```

UTF-8

Rust et types représentent le texte à l'aide du formulaire de codage UTF-8. UTF-8 code un caractère sous la forme d'une séquence d'un à quatre octets (Figure 17-1). `String str`

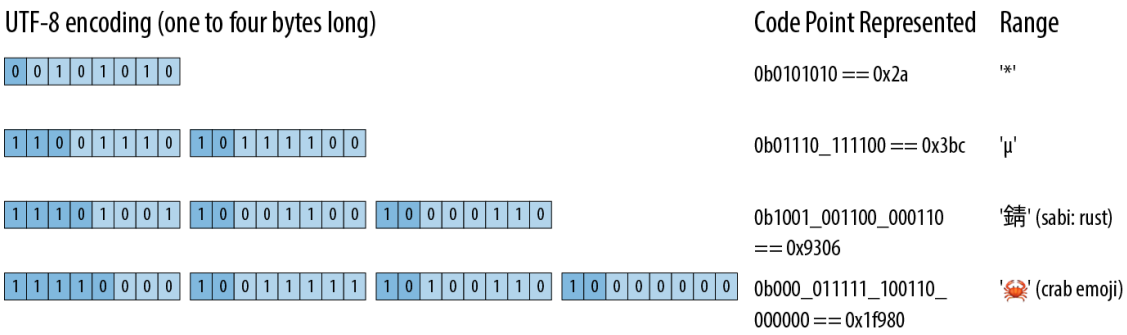


Graphique 17-1. L'encodage UTF-8

Il existe deux restrictions sur les séquences UTF-8 bien formées. Premièrement, seul l'encodage le plus court pour un point de code donné est considéré comme bien formé; vous ne pouvez pas dépenser quatre octets

pour coder un point de code qui tiendrait dans trois. Cette règle garantit qu'il existe exactement un codage UTF-8 pour un point de code donné. Deuxièmement, UTF-8 bien formé ne doit pas encoder les nombres de bout en bout ou au-delà : ceux-ci sont soit réservés à des fins non caractéristiques, soit entièrement en dehors de la plage d'Unicode. `0xd800 0xdfff 0x10ffff`

La figure 17-2 en donne quelques exemples.



Graphique 17-2. Exemples UTF-8

Notez que, même si l'emoji crabe a un encodage dont l'octet principal ne contribue que des zéros au point de code, il a toujours besoin d'un codage de quatre octets: les codages UTF-8 à trois octets ne peuvent transmettre que des points de code de 16 bits et mesurent 17 bits de long. `0x1f980`

Voici un exemple rapide d'une chaîne contenant des caractères avec des codages de longueurs variables :

```
assert_eq!("うどん: udon".as_bytes(),
           &[0xe3, 0x81, 0x86, // う
             0xe3, 0x81, 0xa9, // ど
             0xe3, 0x82, 0x93, // ん
             0x3a, 0x20, 0x75, 0x64, 0x6f, 0x6e // : udon
           ]);
```

La figure 17-2 montre également certaines propriétés très utiles d'UTF-8 :

- Étant donné que UTF-8 code les points de code à travers comme rien de plus que les octets à travers , une plage d'octets contenant du texte ASCII est valide UTF-8. Et si une chaîne UTF-8 inclut uniquement des caractères ASCII, l'inverse est également vrai : le codage UTF-8 est ascii valide. `0 0x7f 0 0x7f`
  Il n'en va pas de même pour Latin-1 : par exemple, Latin-1 code comme l'octet , que UTF-8 interpréterait comme le premier octet d'un codage de trois octets. `é 0xe9`

- En regardant les bits supérieurs de n'importe quel octet, vous pouvez immédiatement dire s'il s'agit du début de l'encodage UTF-8 d'un caractère ou d'un octet au milieu d'un.
- Le premier octet d'un encodage vous indique à lui seul la longueur complète de l'encodage, via ses bits de début.
- Étant donné qu'aucun codage ne dépasse quatre octets, le traitement UTF-8 ne nécessite jamais de boucles illimitées, ce qui est agréable lorsque vous travaillez avec des données non fiables.
- Dans UTF-8 bien formé, vous pouvez toujours dire sans ambiguïté où les codages des caractères commencent et se terminent, même si vous partez d'un point arbitraire au milieu des octets. Les premiers octets UTF-8 et les octets suivants sont toujours distincts, de sorte qu'un encodage ne peut pas démarrer au milieu d'un autre. Le premier octet détermine la longueur totale de l'encodage, de sorte qu'aucun codage ne peut être un préfixe d'un autre. Cela a beaucoup de belles conséquences. Par exemple, la recherche d'un caractère de délimiteur ASCII dans une chaîne UTF-8 ne nécessite qu'une simple analyse de l'octet du délimiteur. Il ne peut jamais apparaître comme une partie d'un encodage multi-octets, il n'est donc pas nécessaire de suivre la structure UTF-8. De même, les algorithmes qui recherchent une chaîne d'octets dans une autre fonctionneront sans modification sur les chaînes UTF-8, même si certains n'examinent même pas chaque octet du texte recherché.

Bien que les codages à largeur variable soient plus compliqués que les codages à largeur fixe, ces caractéristiques rendent UTF-8 plus confortable à utiliser que prévu. La bibliothèque standard gère la plupart des aspects pour vous.

Directionnalité du texte

Alors que des écritures comme le latin, le cyrillique et le thaï sont écrites de gauche à droite, d'autres écritures comme l'hébreu et l'arabe sont écrites de droite à gauche. Unicode stocke les caractères dans l'ordre dans lequel ils seraient normalement écrits ou lus, de sorte que les octets initiaux d'une chaîne contenant, par exemple, du texte hébreu encodent le caractère qui serait écrit à droite:

```
assert_eq!("ערב טוב".chars().next(), Some('ע'));
```

Caractères (char)

Un Rust est une valeur 32 bits contenant un point de code Unicode. A est garanti de tomber dans la plage de à ou dans la gamme à ; toutes les méthodes de création et de manipulation des valeurs garantissent que cela est vrai. Le type implémente et , ainsi que tous les traits habituels pour la comparaison, le hachage et la mise en forme. `char char 0 0xd7ff 0xe000 0x10ffff char char Copy Clone`

Une tranche de chaîne peut produire un itérateur sur ses caractères avec : `slice.chars()`

```
assert_eq!("力二".chars().next(), Some('力'));
```

Dans les descriptions qui suivent, la variable est toujours de type `.ch char`

Classification des caractères

Le type dispose de méthodes pour classer les caractères en quelques catégories courantes, comme indiqué dans le tableau 17-1. Ceux-ci tirent tous leurs définitions d'Unicode. `char`

Tableau 17-1. Méthodes de classification pour le type `char`

Méthode	Description	Exemples

`ch.is_numeric()`	Caractère numérique. Cela inclut les catégories générales Unicode « Nombre; chiffre » et « Nombre; lettre » mais pas « Nombre; autres ».	`'4'.is_numeric()`
		`'小'.is_numeric()`
		`'⑧'.is_numeric()`

`ch.is_alphabetic()`	Un caractère alphabétique : propriété dérivée « Alphabétique » d'Unicode.	`'q'.is_alphabetic()` `'七'.is_alphabetic()`

`ch.is_alphanumeric()`	Numérique ou alphabétique, comme défini précédemment.	`'9'.is_alphanumeric()`
		`'餡'.is_alphanumeric()`
		`!'*'.is_alphanumeric()`

Méthode	Description	Exemples
`ch.is_whitespace()`	Un caractère d'espace blanc : propriété de caractère Unicode « WSpace=Y ».	`' '.is_whitespace()` `'\n'.is_whitespace()` `'\u{A0}'.is_whitespace()`
`ch.is_control()`	Un caractère de contrôle : catégorie générale « Autre, contrôle » d'Unicode.	`'\n'.is_control()` `'\u{85}'.is_control()`

Un ensemble parallèle de méthodes se limite à ASCII uniquement, renvoyant pour tout non-ASCII (Tableau 17-2). `false char`

Table 17-2. ASCII classification methods for `char`

Method	Description	Examples
`ch.is_ascii()`	An ASCII character: one whose code point falls between and inclusive. `0 127`	`'n'.is_ascii()` `!'ñ'.is_ascii()`
`ch.is_ascii_alphabetic()`	An upper- or lowercase ASCII letter, in the range or . `'A'..='Z' 'a'..='z'`	`'n'.is_ascii_alphabetic()` `!'1'.is_ascii_alphabetic()` `!'ñ'.is_ascii_alphabetic()`
`ch.is_ascii_digit()`	Un chiffre ASCII, dans la plage . `'0'..='9'`	`'8'.is_ascii_digit()` `!'-'.is_ascii_digit()` `!'⑧'.is_ascii_digit()`
`ch.is_ascii_hexdigit()`	Tout caractère des plages , , ou . `'0'..='9' 'A'..='F' 'a'..='f'`	
`ch.is_ascii_alphanumeric()`	Un chiffre ASCII ou une lettre majuscule ou minuscule.	`'q'.is_ascii_alphanumeric()` `'0'.is_ascii_alphanumeric()`

Method	Description	Examples
`ch.is_ascii_control()`	Un caractère de contrôle ASCII, y compris 'DEL'.	`'\n'.is_ascii_control()` `'\x7f'.is_ascii_control()`
`ch.is_ascii_graphic()`	Tout caractère ASCII qui laisse de l’encre sur la page : ni un espace ni un caractère de contrôle.	`'Q'.is_ascii_graphic()` `'~'.is_ascii_graphic()` `!' '.is_ascii_graphic()`
`ch.is_ascii_uppercase()`, `ch.is_ascii_lowercase()`	Lettres ASCII majuscules et minuscules.	`'z'.is_ascii_lowercase()` `'Z'.is_ascii_uppercase()`
`ch.is_ascii_punctuation()`	Tout caractère graphique ASCII qui n’est ni alphabétique ni un chiffre.	

Method	Description	Examples

`ch.is_ascii_whitespace()`	Un caractère d'espace ASCII : espace, tabulation horizontale, saut de ligne, saut de formulaire ou retour chariot.	`' '.is_ascii_whitespace()`
		`'\n'.is_ascii_whitespace()`
		`!'\u{A0}'.is_ascii_whitespace()`

Tout le... sont également disponibles sur le type d'octet : `is_ascii_ u8`

```
assert!(32u8.is_ascii_whitespace());
assert!(b'9'.is_ascii_digit());
```

Lorsque vous utilisez ces fonctions, faites attention à implémenter une spécification existante comme une norme de langage de programmation ou un format de fichier, car les classifications peuvent différer de manière surprenante. Par exemple, notez que et diffèrent dans leur traitement de certains personnages: `is_whitespace is_ascii_whitespace`

```
let line_tab = '\u{000b}'; // 'line tab', AKA 'vertical tab'
assert_eq!(line_tab.is_whitespace(), true);
assert_eq!(line_tab.is_ascii_whitespace(), false);
```

La fonction implémente une définition d'espace blanc commune à de nombreuses normes Web, alors qu'elle suit la norme Unicode. `char::is_ascii_whitespace char::is_whitespace`

Manipulation des chiffres

Pour gérer les chiffres, vous pouvez utiliser les méthodes suivantes :

`ch.to_digit(radix)`

Décide s'il s'agit d'un chiffre en base . Si c'est le cas, il renvoie , où est un fichier . Sinon, il renvoie . Cela ne reconnaît que les chiffres ASCII, et non la classe plus large de caractères couverts par . Le paramètre peut aller de 2 à 36. Pour les

rayons supérieurs à 10, les lettres ASCII de l'un ou l'autre cas sont considérées comme des chiffres avec des valeurs comprises entre 10 et 35. `ch radix Some(num) num u32 None char::is_numeric radix`

`std::char::from_digit(num, radix)`

Fonction libre qui convertit la valeur numérique en a si possible. Si peut être représenté par un seul chiffre dans , renvoie , où est le chiffre. Lorsqu'il est supérieur à 10, peut être une lettre minuscule. Sinon, il renvoie `. u32 num char num radix from_digit Some(ch) ch radix ch None`

C'est l'inverse de . Si est , alors est . S'il s'agit d'un chiffre ASCII ou d'une lettre minuscule, l'inverse est également valable. `to_digit std::char::from_digit(num, radix) Some(ch) ch.to_digit(radix) Some(num) ch`

`ch.is_digit(radix)`

Renvoie si est un chiffre ASCII en base . Cela équivaut à `. true ch radix ch.to_digit(radix) != None`

Ainsi, par exemple :

```
assert_eq!('F'.to_digit(16), Some(15));
assert_eq!(std::char::from_digit(15, 16), Some('f'));
assert!(char::is_digit('f', 16));
```

Conversion de casse pour les caractères

Pour la gestion de la casse de caractères :

`ch.is_lowercase(), ch.is_uppercase()`

Indiquez s'il s'agit d'un caractère alphabétique minuscule ou majuscule. Ceux-ci suivent les propriétés dérivées en minuscules et en majuscules d'Unicode, de sorte qu'ils couvrent les alphabets non latins comme le grec et le cyrillique et donnent également les résultats attendus pour ASCII. `ch`

`ch.to_lowercase(), ch.to_uppercase()`

Renvoyer des itérateurs qui produisent les caractères des équivalents minuscules et majuscules de , selon les algorithmes de conversion de casse par défaut Unicode : `ch`

```
let mut upper = 's'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);
```

Ces méthodes renvoient un itérateur au lieu d'un seul caractère, car la conversion de casse en Unicode n'est pas toujours un processus un à un :

```
// The uppercase form of the German letter "sharp S" is "SS":
let mut upper = 'ß'.to_uppercase();
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), Some('S'));
assert_eq!(upper.next(), None);

// Unicode says to lowercase Turkish dotted capital 'İ' to 'i'
// followed by `'\u{307}'`, COMBINING DOT ABOVE, so that a
// subsequent conversion back to uppercase preserves the dot.
let ch = 'İ'; // `'\u{130}'`
let mut lower = ch.to_lowercase();
assert_eq!(lower.next(), Some('i'));
assert_eq!(lower.next(), Some('\u{307}'));
assert_eq!(lower.next(), None);
```

Pour plus de commodité, ces itérateurs implémentent le trait, de sorte que vous pouvez les transmettre directement à une ou une macro. `std::fmt::Display println! write!`

Conversions vers et depuis des entiers

L'opérateur de Rust convertira a en n'importe quel type entier, masquant silencieusement tous les bits supérieurs : `as char`

```
assert_eq!('B' as u32, 66);
assert_eq!('饂' as u8, 66);   // upper bits truncated
assert_eq!('二' as i8, -116); // same
```

L'opérateur convertira n'importe quelle valeur en a , et implémente également, mais les types entiers plus larges peuvent représenter des points de code non valides, donc pour ceux que vous devez utiliser , qui renvoie
: `as u8 char char From<u8> std::char::from_u32 Option<char>`

```
assert_eq!(char::from(66), 'B');
assert_eq!(std::char::from_u32(0x9942), Some('餂'));
assert_eq!(std::char::from_u32(0xd800), None); // reserved for UTF-16
```

String et str

Les rouilles et les types de rouille sont garantis pour ne contenir que des UTF-8 bien formés. La bibliothèque garantit cela en limitant les façons dont vous pouvez créer et les valeurs et les opérations que vous pouvez effectuer sur eux, de sorte que les valeurs soient bien formées lorsqu'elles sont introduites et le restent lorsque vous travaillez avec elles. Toutes leurs méthodes protègent cette garantie: aucun fonctionnement sûr sur eux ne peut introduire UTF-8 mal formé. Cela simplifie le code qui fonctionne avec le texte. `String str String str`

Rust place les méthodes de gestion de texte sur l'un ou l'autre ou selon que la méthode a besoin d'un tampon redimensionnable ou qu'il s'agit d'un contenu juste pour utiliser le texte en place. Depuis les déréférences à , chaque méthode définie sur est également directement disponible sur. Cette section présente les méthodes des deux types, regroupées par fonction approximative. `str String String &str str String`

Ces méthodes indexent le texte par décalage d'octets et mesurent sa longueur en octets plutôt qu'en caractères. En pratique, compte tenu de la nature d'Unicode, l'indexation par caractère n'est pas aussi utile qu'il n'y paraît, et les décalages d'octets sont plus rapides et plus simples. Si vous essayez d'utiliser un décalage d'octets qui atterrit au milieu de l'encodage UTF-8 d'un personnage, la méthode panique, de sorte que vous ne pouvez pas introduire UTF-8 mal formé de cette façon.

A est implémenté comme un wrapper autour de a qui garantit que le contenu du vecteur est toujours bien formé UTF-8. Rust ne changera jamais pour utiliser une représentation plus compliquée, vous pouvez donc supposer que les caractéristiques de performance des actions. `String Vec<u8> String String Vec`

Dans ces explications, les variables ont les types [donnés dans le tableau 17-3](#).

Tableau 17-3. Types de variables utilisées dans les explications

Variable	Type présumé
`string`	`String`
`slice`	`&str` ou quelque chose qui fait référence à l'un d'entre eux, comme ou `String Rc<String>`
`ch`	`char`
`n`	`usize`, une longueur
`i, j`	`usize`, un décalage d'octets
`range`	Plage de décalages d'octets, soit entièrement bornés comme , soit partiellement délimités comme , , ou `usize i..j i.. ..j ..`
`patter n`	Tout type de motif : , , , , , ou `char String &str &[char] FnMut(char) -> bool`

Nous décrivons les types de modèles dans « Modèles pour la recherche de texte ».

Création de valeurs de chaîne

Il existe plusieurs façons courantes de créer des valeurs : `String`

`String::new()`

Renvoie une chaîne fraîche et vide. Il n'a pas de tampon alloué au tas, mais en allouera un si nécessaire.

`String::with_capacity(n)`

Renvoie une chaîne vide avec un tampon pré-alloué pour contenir au moins des octets. Si vous connaissez la longueur de la chaîne que vous générez à l'avance, ce constructeur vous permet d'obtenir la taille de la mémoire tampon correctement dès le début, au lieu de redimensionner la mémoire tampon au fur et à mesure que vous construisez la chaîne. La chaîne continuera à développer sa mémoire tampon si nécessaire si sa longueur dépasse les octets. Comme les vecteurs, les chaînes ont , et les méthodes, mais généralement la logique

d'allocation par défaut est
correcte. `n n capacity reserve shrink_to_fit`

`str_slice.to_string()`

Alloue un produit dont le contenu est une copie de . Nous avons utilisé des expressions comme tout au long du livre pour faire des s à partir de littéraux de chaîne. `String str_slice "literal text".to_string() String`

`iter.collect()`

Construit une chaîne en concaténant les éléments d'un itérateur, qui peuvent être , ou des valeurs. Par exemple, pour supprimer tous les espaces d'une chaîne, vous pouvez écrire : `char &str String`

```
let spacey = "man hat tan";
let spaceless: String =
    spacey.chars().filter(|c| !c.is_whitespace()).collect();
assert_eq!(spaceless, "manhattan");
```

L'utilisation de cette méthode tire parti de la mise en œuvre du trait de caractère. `collect String std::iter::FromIterator`

`slice.to_owned()`

Renvoie une copie de la tranche sous la forme d'un fichier . Le type ne peut pas implémenter : le trait nécessiterait sur a de renvoyer une valeur, mais n'est pas dimensionné. Cependant, implémente , ce qui permet à l'implémenteur de spécifier son équivalent
propre. `String str Clone clone &str str str &str ToOwned`

Simple Inspection

Ces méthodes obtiennent des informations de base à partir de tranches de chaîne :

`slice.len()`

Longueur de , en octets. `slice`

`slice.is_empty()`

True si . `slice.len() == 0`

`slice[range]`

Renvoie une tranche empruntant la partie donnée de . Les plages partiellement délimitées et non bornées sont OK ; par exemple: `slice`

```
let full = "bookkeeping";
assert_eq!(&full[..4], "book");
assert_eq!(&full[5..], "eeping");
assert_eq!(&full[2..4], "ok");
assert_eq!(full[..].len(), 11);
assert_eq!(full[5..].contains("boo"), false);
```

Notez que vous ne pouvez pas indexer une tranche de chaîne avec une seule position, comme . La récupération d'un seul caractère à un décalage d'octet donné est un peu maladroite : vous devez produire un itérateur sur la tranche et lui demander d'analyser l'UTF-8 d'un caractère : `slice[i] chars`

```
let parenthesized = "Rust (饂)";
assert_eq!(parenthesized[6..].chars().next(), Some('饂'));
```

Cependant, vous devriez rarement avoir besoin de le faire. Rust a des façons beaucoup plus agréables d'itérer sur les tranches, que nous décrivons dans « Itérer sur le texte ».

`slice.split_at(i)`

Renvoie un tuple de deux tranches partagées empruntées à : la portion jusqu'au décalage d'octets et la partie qui la suit. En d'autres termes, cela renvoie `.slice i (slice[..i], slice[i..])`

`slice.is_char_boundary(i)`

True si le décalage d'octet se situe entre les limites de caractères et convient donc comme décalage dans . `i slice`

Naturellement, les tranches peuvent être comparées pour l'égalité, ordonnées et hachées. La comparaison ordonnée traite simplement la chaîne comme une séquence de points de code Unicode et les compare dans l'ordre lexicographique.

Ajout et insertion de texte

Les méthodes suivantes ajoutent du texte à un : `String`

`string.push(ch)`

Ajoute le caractère à la fin . `ch string`

`string.push_str(slice)`

Ajoute le contenu complet de . `slice`

string.extend(iter)

Ajoute les éléments produits par l'itérateur à la chaîne. L'itérateur peut produire , ou des valeurs. Ce sont les implémentations de : `iter` `char` `str` `String` `String` `std::iter::Extend`

```
let mut also_spaceless = "con".to_string();
also_spaceless.extend("tri but ion".split_whitespace());
assert_eq!(also_spaceless, "contribution");
```

string.insert(i, ch)

Insère le caractère unique au décalage d'octet dans . Cela implique de déplacer tous les caractères après pour faire de la place pour , de sorte que la construction d'une chaîne de cette façon peut nécessiter un temps quadratique dans la longueur de la chaîne. `ch` `i` `string` `i` `ch`

string.insert_str(i, slice)

Cela fait la même chose pour , avec la même mise en garde de performance. `slice`

`String` implémente , ce qui signifie que les et les macros peuvent ajouter du texte mis en forme à s : `std::fmt::Write` `write!` `writeln!` `String`

```
use std::fmt::Write;

let mut letter = String::new();
writeln!(letter, "Whose {} these are I think I know", "rutabagas")?;
writeln!(letter, "His house is in the village though;")?;
assert_eq!(letter, "Whose rutabagas these are I think I know\n\
                    His house is in the village though;\n");
```

Puisque et sont conçus pour écrire dans des flux de sortie, ils renvoient un , ce que Rust se plaint si vous ignorez. Ce code utilise l'opérateur pour le gérer, mais écrire sur un est en fait infaillible, donc dans ce cas, l'appel serait OK aussi. `write!` `writeln!` `Result` `?` `String` `.unwrap()`

Puisque implémente et , vous pouvez écrire du code comme ceci: `String` `Add<&str>` `AddAssign<&str>`

```
let left = "partners".to_string();
let mut right = "crime".to_string();
assert_eq!(left + " in " + &right, "partners in crime");
```

```
right += " doesn't pay";
assert_eq!(right, "crime doesn't pay");
```

Lorsqu'il est appliqué à des chaînes, l'opérateur prend son opérande gauche par valeur, de sorte qu'il peut réellement le réutiliser à la suite de l'ajout. Par conséquent, si la mémoire tampon de l'opérande gauche est suffisamment grande pour contenir le résultat, aucune allocation n'est nécessaire. `+ String`

Dans un manque regrettable de symétrie, l'opérande gauche de ne peut pas être un , vous ne pouvez donc pas écrire: `+ &str`

```
let parenthetical = "(" + string + ")";
```

Vous devez plutôt écrire :

```
let parenthetical = "(".to_string() + &string + ")";
```

Cependant, cette restriction décourage la construction de chaînes à partir de la fin vers l'arrière. Cette approche fonctionne mal car le texte doit être déplacé à plusieurs reprises vers la fin de la mémoire tampon.

Construire des chaînes du début à la fin en ajoutant de petits morceaux, cependant, est efficace. A se comporte comme un vecteur, doublant toujours au moins la taille de son tampon lorsqu'il a besoin de plus de capacité. Cela permet de conserver les frais généraux de recopie proportionnellement à la taille finale. Néanmoins, l'utilisation de la création de chaînes avec la bonne taille de tampon pour commencer évite le redimensionnement et peut réduire le nombre d'appels à l'allocateur de tas. `String String::with_capacity`

Suppression et remplacement de texte

`String` dispose de quelques méthodes pour supprimer du texte (celles-ci n'affectent pas la capacité de la chaîne; utilisez si vous avez besoin de libérer de la mémoire): `shrink_to_fit`

`string.clear()`

Réinitialise la chaîne vide. `string`

`string.truncate(n)`

Supprime tous les caractères après le décalage d'octets , en laissant avec une longueur d'au plus . Si est plus court que les octets, cela n'a aucun effet. `n string n string n`

`string.pop()`

Supprime le dernier caractère de , le cas échéant, et le renvoie sous la forme d'un fichier . `string Option<char>`

`string.remove(i)`

Supprime le caractère au décalage d'octet et le renvoie, en déplaçant tous les caractères suivants vers l'avant. Cela prend du temps linéaire dans le nombre de caractères suivants. `i string`

`string.drain(range)`

Renvoie un itérateur sur la plage donnée d'index d'octets et supprime les caractères une fois l'itérateur supprimé. Caractères après que la plage est décalée vers l'avant :

```
let mut choco = "chocolate".to_string();
assert_eq!(choco.drain(3..6).collect::<String>(), "col");
assert_eq!(choco, "choate");
```

Si vous souhaitez simplement supprimer la plage, vous pouvez simplement laisser tomber l'itérateur immédiatement, sans en tirer d'éléments:

```
let mut winston = "Churchill".to_string();
winston.drain(2..6);
assert_eq!(winston, "Chill");
```

`string.replace_range(range, replacement)`

Remplace la plage donnée par la tranche de chaîne de remplacement donnée. La tranche n'a pas besoin d'avoir la même longueur que la plage remplacée, mais à moins que la plage remplacée ne se termine par , cela nécessitera de déplacer tous les octets après la fin de la plage : `string string`

```
let mut beverage = "a piña colada".to_string();
beverage.replace_range(2..7, "kahlua"); // 'ñ' is two bytes!
assert_eq!(beverage, "a kahlua colada");
```

Conventions de recherche et d'itération

Les fonctions de bibliothèque standard de Rust pour la recherche de texte et l'itération sur le texte suivent certaines conventions de dénomination pour les rendre plus faciles à retenir:

r

La plupart des opérations traitent le texte du début à la fin, mais les opérations dont les noms commencent par **`r`** fonctionnent de bout en début. Par exemple, est la version de bout en bout de . Dans certains cas, le changement de direction peut affecter non seulement l'ordre dans lequel les valeurs sont produites, mais aussi les valeurs elles-mêmes. Voir le diagramme de la figure 17-3 pour un exemple de ceci. `rsplit split`

n

Les itérateurs dont les noms se terminent par **`n`** se limitent à un nombre donné de correspondances.

_indices

Les itérateurs dont les noms se terminent par produisent, avec leurs valeurs d'itération habituelles, les décalages d'octets dans la tranche à laquelle ils apparaissent. `_indices`

La bibliothèque standard ne fournit pas toutes les combinaisons pour chaque opération. Par exemple, de nombreuses opérations n'ont pas besoin d'une variante, car il est assez facile de simplement mettre fin à l'itération plus tôt. `n`

Modèles de recherche de texte

Lorsqu'une fonction de bibliothèque standard doit rechercher, faire correspondre, diviser ou découper du texte, elle accepte plusieurs types différents pour représenter ce qu'il faut rechercher :

```
let haystack = "One fine day, in the middle of the night";

assert_eq!(haystack.find(','), Some(12));
assert_eq!(haystack.find("night"), Some(35));
assert_eq!(haystack.find(char::is_whitespace), Some(3));
```

Ces types sont appelés *modèles* et la plupart des opérations les prennent en charge :

```
assert_eq!("## Elephants"
           .trim_start_matches(|ch: char| ch == '#' || ch.is_whitespac
           "Elephants");
```

La bibliothèque standard prend en charge quatre principaux types de modèles :

- A en tant que motif correspond à ce caractère. `char`
- A ou ou comme un motif correspond à une sous-chaîne égale au motif. `String &str &&str`
- Une fermeture en tant que motif correspond à un caractère unique pour lequel la fermeture renvoie true. `FnMut(char) -> bool`
- A en tant que modèle (pas un , mais une tranche de valeurs) correspond à n'importe quel caractère unique qui apparaît dans la liste. Notez que si vous écrivez la liste en tant que littéral de tableau, vous devrez peut-être appeler pour obtenir le bon type : `&[char] &str char as_ref()`

  ```
  let code = "\t    function noodle() { ";
  assert_eq!(code.trim_start_matches([' ', '\t'].as_ref()),
             "function noodle() { ");
  // Shorter equivalent: &[' ', '\t'][..]
  ```

  Sinon, Rust sera confondu par le type de tableau à taille fixe , qui n'est malheureusement pas un type de modèle. `&[char; 2]`

Dans le code de la bibliothèque, un modèle est un type qui implémente le trait. Les détails de ne sont pas encore stables, vous ne pouvez donc pas l'implémenter pour vos propres types dans Rust stable, mais la porte est ouverte pour permettre des expressions régulières et d'autres modèles sophistiqués à l'avenir. Rust garantit que les types de modèles pris en charge aujourd'hui continueront à fonctionner à l'avenir. `std::str::Pattern Pattern`

Recherche et remplacement

Rust a quelques méthodes pour rechercher des motifs dans les tranches et éventuellement les remplacer par du nouveau texte:

`slice.contains(pattern)`

Renvoie true si contient une correspondance pour . `slice pattern`

slice.starts_with(pattern), slice.ends_with(pattern)

Renvoyer true si le texte initial ou final de ' correspond à
: `slice pattern`

```
assert!("2017".starts_with(char::is_numeric));
```

slice.find(pattern), slice.rfind(pattern)

Renvoyer si contient une correspondance pour , où est le décalage d'octet auquel le motif apparaît. La méthode renvoie la première correspondance, la dernière
: `Some(i) slice pattern i find rfind`

```
let quip = "We also know there are known unknowns";
assert_eq!(quip.find("know"), Some(8));
assert_eq!(quip.rfind("know"), Some(31));
assert_eq!(quip.find("ya know"), None);
assert_eq!(quip.rfind(char::is_uppercase), Some(0));
```

slice.replace(pattern, replacement)

Renvoie un nouveau formé en remplaçant avec empressement tous les matchs par : `String pattern replacement`

```
assert_eq!("The only thing we have to fear is fear itself"
           .replace("fear", "spin"),
           "The only thing we have to spin is spin itself");

assert_eq!("`Borrow` and `BorrowMut`"
           .replace(|ch:char| !ch.is_alphanumeric(), ""),
           "BorrowandBorrowMut");
```

Parce que le remplacement est fait avec empressement, le comportement de 'sur les matchs qui se chevauchent peut être surprenant. Ici, il y a quatre instances du modèle, mais la deuxième et la quatrième ne correspondent plus après que la première et la troisième sont remplacées: `.replace() "aba"`

```
assert_eq!("cabababababbage"
           .replace("aba", "***"),
           "c***b***babbage")
```

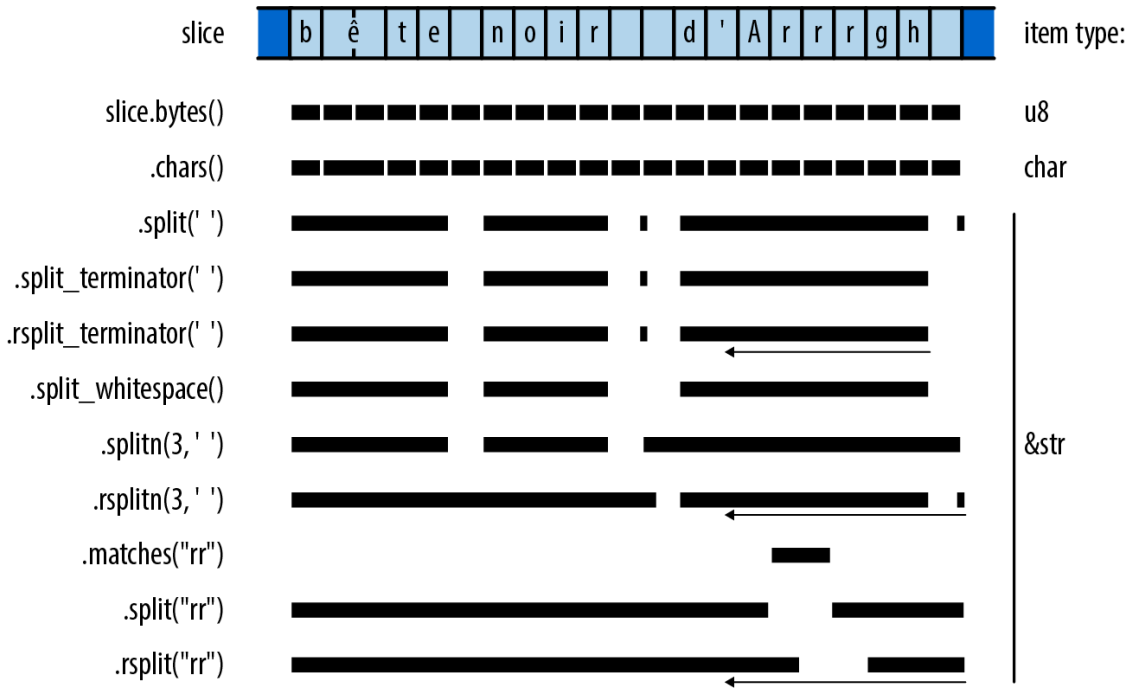
slice.replacen(pattern, replacement, n)

Cela fait de même, mais remplace tout au plus les premiers matchs. `n`

Itération sur le texte

La bibliothèque standard offre plusieurs façons d'itérer sur le texte d'une tranche. La figure 17-3 en montre des exemples.

Vous pouvez considérer les familles comme étant complémentaires les unes des autres: les divisions sont les fourchettes entre les matchs. `split match`



Graphique 17-3. Quelques façons d'itérer sur une tranche

La plupart de ces méthodes renvoient des itérateurs réversibles (c'est-à-dire qu'ils implémentent) : l'appel de leur méthode d'adaptateur vous donne un itérateur qui produit les mêmes éléments, mais dans l'ordre inverse. `DoubleEndedIterator .rev()`

`slice.chars()`

Renvoie un itérateur sur les caractères de .' `slice`

`slice.char_indices()`

Renvoie un itérateur sur les caractères de et leurs décalages d'octets : `slice`

```
assert_eq!("élan".char_indices().collect::<Vec<_>>(),
           vec![(0, 'é'), // has a two-byte UTF-8 encoding
                (2, 'l'),
                (3, 'a'),
                (4, 'n')]);
```

Notez que cela n'est pas équivalent à , car il fournit le décalage d'octet de chaque caractère dans la tranche, au lieu de simplement numéroter les caractères. `.chars().enumerate()`

`slice.bytes()`

Renvoie un itérateur sur les octets individuels de , exposant le codage UTF-8 : `slice`

```
assert_eq!("élan".bytes().collect::<Vec<_>>(),
           vec![195, 169, b'l', b'a', b'n']);
```

`slice.lines()`

Renvoie un itérateur sur les lignes de . Les lignes sont terminées par ou . Chaque article produit est un emprunt auprès de . Les éléments n'incluent pas les caractères de fin des lignes. `slice "\n" "\r\n" &str slice`

`slice.split(pattern)`

Renvoie un itérateur sur les parties de séparées par des correspondances de . Cela produit des chaînes vides entre les correspondances immédiatement adjacentes, ainsi que pour les correspondances au début et à la fin de . `slice pattern slice`

L'itérateur renvoyé n'est pas réversible si est un . De tels modèles peuvent produire différentes séquences de correspondances en fonction de la direction à partir de laquelle vous scannez, ce que les itérateurs réversibles sont interdits de faire. Au lieu de cela, vous pourrez peut-être utiliser la méthode décrite ci-dessous. `pattern &str rsplit`

`slice.rsplit(pattern)`

Cette méthode est la même, mais scanne de bout en bout, produisant des correspondances dans cet ordre. `slice`

`slice.split_terminator(pattern)`,
`slice.rsplit_terminator(pattern)`

Ceux-ci sont similaires, sauf que le motif est traité comme un terminateur, pas un séparateur : si les correspondances à la toute fin de , les itérateurs ne produisent pas de tranche vide représentant la chaîne vide entre cette correspondance et la fin de la tranche, comme et le font. Par exemple: `pattern slice split rsplit`

```
    // The ':' characters are separators here. Note the final "".
    assert_eq!("jimb:1000:Jim Blandy:".split(':').collect::<Vec<_>>(),
               vec!["jimb", "1000", "Jim Blandy", ""]);

    // The '\n' characters are terminators here.
    assert_eq!("127.0.0.1  localhost\n\
                127.0.0.1  www.reddit.com\n"
               .split_terminator('\n').collect::<Vec<_>>(),
               vec!["127.0.0.1  localhost",
                    "127.0.0.1  www.reddit.com"]);
                    // Note, no final ""!
```

`slice.splitn(n, pattern)`, `slice.rsplitn(n, pattern)`

Ceux-ci sont comme et , sauf qu'ils divisent la chaîne en au plus des tranches, à la première ou à la dernière correspondance pour . `split rsplit n n-1 pattern`

`slice.split_whitespace()`, `slice.split_ascii_whitespace()`

Renvoyer un itérateur sur les parties séparées par des espaces blancs de . Une série de plusieurs caractères d'espace blanc est considérée comme un séparateur unique. L'espace blanc de fin est ignoré. `slice`

La méthode utilise la définition Unicode d'espace blanc, telle qu'implémentée par la méthode sur . La méthode utilise à la place, ce qui ne reconnaît que les caractères d'espace ASCII. `split_whitespace is_whitespace char split_ascii_whitespace char::is_ascii_whitespace`

```
    let poem = "This  is  just  to say\n\
                I have eaten\n\
                the plums\n\
                again\n";

    assert_eq!(poem.split_whitespace().collect::<Vec<_>>(),
               vec!["This", "is", "just", "to", "say",
                    "I", "have", "eaten", "the", "plums",
                    "again"]);
```

`slice.matches(pattern)`

Renvoie un itérateur sur les correspondances pour dans la tranche. est le même, mais itére de bout en bout. `pattern slice.rmatches(pattern)`

`slice.match_indices(pattern)`,
`slice.rmatch_indices(pattern)`

Ceux-ci sont similaires, sauf que les éléments produits sont des paires, où est le décalage d'octets auquel la correspondance commence et est la tranche correspondante. `(offset, match) offset match`

Coupe

Couper une chaîne revient à supprimer du texte, généralement des espaces blancs, du début ou de la fin de la chaîne. Il est souvent utile pour nettoyer les entrées lues à partir d'un fichier où l'utilisateur peut avoir mis en retrait du texte pour plus de lisibilité ou accidentellement laissé un espace blanc de fin sur une ligne.

`slice.trim()`

Renvoie une sous-section qui omet tout espace blanc de début et de fin. omet uniquement l'espace blanc de début, uniquement l'espace de fin : `slice slice.trim_start() slice.trim_end()`

```
assert_eq!("\t*.rs  ".trim(), "*.rs");
assert_eq!("\t*.rs  ".trim_start(), "*.rs  ");
assert_eq!("\t*.rs  ".trim_end(), "\t*.rs");
```

`slice.trim_matches(pattern)`

Renvoie une sous-section qui omet toutes les correspondances du début et de la fin. Les méthodes et font de même pour les matchs de début ou de fin uniquement
: `slice pattern trim_start_matches trim_end_matches`

```
assert_eq!("001990".trim_start_matches('0'), "1990");
```

`slice.strip_prefix(pattern)`, *`slice.strip_suffix(pattern)`*

Si commence par , renvoie la colonne en maintenant la tranche enfoncée avec le texte correspondant supprimé. Sinon, il renvoie . La méthode est similaire, mais vérifie une correspondance à la fin de la chaîne. `slice pattern strip_prefix Some None strip_suffix`

Ceux-ci sont comme et , sauf qu'ils renvoient un , et qu'une seule copie de est

supprimée: `trim_start_matches trim_end_matches Option pattern`

```
let slice = "banana";
assert_eq!(slice.strip_suffix("na"),
           Some("bana"))
```

Conversion de casse pour les chaînes

Les méthodes et renvoyer une chaîne fraîchement allouée contenant le texte de converti en majuscules ou minuscules. Le résultat peut ne pas être de la même longueur que ; voir « Conversion de casse pour les caractères » pour plus de détails. `slice.to_uppercase() slice.to_lowercase() slice slice`

Analyse d'autres types à partir de chaînes

Rust fournit des caractéristiques standard pour analyser les valeurs des chaînes et produire des représentations textuelles des valeurs.

Si un type implémente le trait, il fournit un moyen standard d'analyser une valeur à partir d'une tranche de chaîne : `std::str::FromStr`

```
pub trait FromStr: Sized {
    type Err;
    fn from_str(s: &str) -> Result<Self, Self::Err>;
}
```

Tous les types de machines habituels implémentent : `FromStr`

```
use std::str::FromStr;

assert_eq!(usize::from_str("3628800"), Ok(3628800));
assert_eq!(f64::from_str("128.5625"), Ok(128.5625));
assert_eq!(bool::from_str("true"), Ok(true));

assert!(f64::from_str("not a float at all").is_err());
assert!(bool::from_str("TRUE").is_err());
```

Le type implémente également , pour les chaînes avec un seul caractère : `char FromStr`

```
assert_eq!(char::from_str("é"), Ok('é'));
assert!(char::from_str("abcdefg").is_err());
```

Le type, une détention d'une adresse Internet IPv4 ou IPv6, implémente également : `std::net::IpAddr enum FromStr`

```
use std::net::IpAddr;

let address = IpAddr::from_str("fe80::0000:3ea9:f4ff:fe34:7a50")?;
assert_eq!(address,
           IpAddr::from([0xfe80, 0, 0, 0, 0x3ea9, 0xf4ff, 0xfe34, 0x7a
```

Les tranches de chaîne ont une méthode qui analyse la tranche dans le type de votre choix, en supposant qu'elle implémente . Comme avec , vous devrez parfois préciser quel type vous voulez, donc ce n'est pas toujours beaucoup plus lisible que d'appeler directement: `parse FromStr Iterator::collect parse from_str`

```
let address = "fe80::0000:3ea9:f4ff:fe34:7a50".parse::<IpAddr>()?;
```

Conversion d'autres types en chaînes

Il existe trois façons principales de convertir des valeurs non textuelles en chaînes :

- Les types qui ont un formulaire imprimé naturel lisible par l'homme peuvent implémenter le trait, ce qui vous permet d'utiliser le spécificateur de format dans la macro : `std::fmt::Display {} format!`

  ```
  assert_eq!(format!("{}, wow", "doge"), "doge, wow");
  assert_eq!(format!("{}", true), "true");
  assert_eq!(format!("({:.3}, {:.3})", 0.5, f64::sqrt(3.0)/2.0),
             "(0.500, 0.866)");

  // Using `address` from above.
  let formatted_addr: String = format!("{}", address);
  assert_eq!(formatted_addr, "fe80::3ea9:f4ff:fe34:7a50");
  ```

  Tous les types numériques de machine de Rust implémentent , tout comme les caractères, les chaînes et les tranches. Les types de pointeurs intelligents , , et implémenter si lui-même le fait: leur forme affichée est simplement celle de leur référent. Les conteneurs aiment et ne sont pas

implémentés, car il n'existe pas de forme naturelle lisible par l'homme pour ces types. `Display Box<T> Rc<T> Arc<T> Display T Vec HashMap Display`

- Si un type implémente , la bibliothèque standard implémente automatiquement le trait pour cela, dont la seule méthode peut être plus pratique lorsque vous n'avez pas besoin de la flexibilité de : `Display std::str::ToString to_string format!`

  ```
  // Continued from above.
  assert_eq!(address.to_string(), "fe80::3ea9:f4ff:fe34:7a50");
  ```

  Le trait est antérieur à l'introduction et est moins flexible. Pour vos propres types, vous devez généralement implémenter au lieu de . `ToString Display Display ToString`
- Chaque type public de la bibliothèque standard implémente , ce qui prend une valeur et la formate sous forme de chaîne d'une manière utile aux programmeurs. Le moyen le plus simple d'utiliser pour produire une chaîne consiste à utiliser le spécificateur de format de la macro : `std::fmt::Debug Debug format! {:?}`

  ```
  // Continued from above.
  let addresses = vec![address,
                       IpAddr::from_str("192.168.0.1")?];
  assert_eq!(format!("{:?}", addresses),
             "[fe80::3ea9:f4ff:fe34:7a50, 192.168.0.1]");
  ```

  Cela tire parti d'une implémentation générale de pour , pour tout ce qui implémente lui-même . Tous les types de collection de Rust ont de telles implémentations. `Debug Vec<T> T Debug`
  Vous devez également implémenter pour vos propres types. Habituellement, il est préférable de laisser Rust dériver une implémentation, comme nous l'avons fait pour le type dans le chapitre 12: `Debug Complex`

  ```
  #[derive(Copy, Clone, Debug)]
  struct Complex { re: f64, im: f64 }
  ```

Les traits de mise en forme ne sont que deux parmi plusieurs que la macro et ses parents utilisent pour mettre en forme les valeurs sous forme de texte. Nous couvrirons les autres et expliquerons comment les

implémenter tous dans « Formatage des valeurs ». `Display Debug format!`

Emprunt en tant qu'autres types textuels

Vous pouvez emprunter le contenu d'une tranche de plusieurs manières différentes :

- Les tranches et s implémentent , , , et . De nombreuses fonctions de bibliothèque standard utilisent ces traits comme limites sur leurs types de paramètres, de sorte que vous pouvez leur transmettre directement des tranches et des chaînes, même lorsqu'elles veulent vraiment un autre type. Voir « AsRef et AsMut » pour une explication plus détaillée. `String AsRef<str> AsRef<[u8]> AsRef<Path> AsRef<OsStr>`
- Les tranches et les chaînes implémentent également le trait. et utiliser pour faire fonctionner s bien comme des clés dans une table. Voir « Borrow and BorrowMut » pour plus de détails. `std::borrow::Borrow<str> HashMap BTreeMap Borrow String`

Accès au texte en UTF-8

Il existe deux façons principales d'obtenir les octets représentant le texte, selon que vous souhaitez vous approprier les octets ou simplement les emprunter :

`slice.as_bytes()`

Emprunte les octets de . Comme il ne s'agit pas d'une référence modifiable, on peut supposer que ses octets resteront bien formés UTF-8. `slice & [u8] slice`

`string.into_bytes()`

Prend possession et renvoie un octet de la chaîne par valeur. Il s'agit d'une conversion bon marché, car elle remet simplement ce que la chaîne utilisait comme tampon. Comme il n'existe plus, il n'est pas nécessaire que les octets continuent d'être bien formés UTF-8, et l'appelant est libre de le modifier à sa guise. `string Vec<u8> Vec<u8> string Vec<u8>`

Production de texte à partir de données UTF-8

Si vous avez un bloc d'octets qui, selon vous, contient des données UTF-8, vous disposez de quelques options pour les convertir en s ou en tranches,

selon la façon dont vous souhaitez gérer les erreurs : `String`

`str::from_utf8(byte_slice)`

Prend une tranche d'octets et renvoie un : soit s'il contient utf-8 bien formé, soit une erreur dans le cas contraire. `&[u8] Result Ok(&str) byte_slice`

`String::from_utf8(vec)`

Tente de construire une chaîne à partir d'une valeur transmise par. S'il contient UTF-8 bien formé, renvoie , où a pris possession de pour une utilisation comme tampon. Aucune allocation de tas ou copie du texte n'a lieu. `Vec<u8> vec from_utf8 Ok(string) string vec`

Si les octets ne sont pas valides UTF-8, cela renvoie , où est une valeur d'erreur. L'appel vous redonne le vecteur d'origine , afin qu'il ne soit pas perdu lorsque la conversion échoue `:Err(e) e FromUtf8Error e.into_bytes() vec`

```
let good_utf8: Vec<u8> = vec![0xe9, 0x8c, 0x86];
assert_eq!(String::from_utf8(good_utf8).ok(), Some("錆".to_string(

let bad_utf8:  Vec<u8> = vec![0x9f, 0xf0, 0xa6, 0x80];
let result = String::from_utf8(bad_utf8);
assert!(result.is_err());
// Since String::from_utf8 failed, it didn't consume the original
// vector, and the error value hands it back to us unharmed.
assert_eq!(result.unwrap_err().into_bytes(),
           vec![0x9f, 0xf0, 0xa6, 0x80]);
```

`String::from_utf8_lossy(byte_slice)`

Tente de construire un ou à partir d'une tranche d'octets partagée. Cette conversion réussit toujours, remplaçant tout UTF-8 mal formé par des caractères de remplacement Unicode. La valeur renvoyée est une valeur qui emprunte un directement à s'il contient utf-8 bien formé ou possède un nouveau alloué avec des caractères de remplacement substitués aux octets mal formés. Par conséquent, lorsqu'il est bien formé, aucune allocation ou copie de tas n'a lieu. Nous discutons plus en détail dans « Différer l'allocation ». `String &str & [u8] Cow<str> &str byte_slice String byte_slice Cow<str >`

`String::from_utf8_unchecked(vec)`

Si vous savez pertinemment que votre contient UTF-8 bien formé, alors vous pouvez appeler cette fonction dangereuse. Cela se termine simplement par un et

le renvoie, sans examiner les octets du tout. Vous êtes responsable de vous assurer que vous n'avez pas introduit UTF-8 mal formé dans le système, c'est pourquoi cette fonction est marquée . `Vec<u8> vec String unsafe`

`str::from_utf8_unchecked(byte_slice)`

De même, cela prend un et le renvoie sous la forme d'un , sans vérifier s'il contient UTF-8 bien formé. Comme pour , vous êtes responsable de vous assurer que c'est sûr. `&[u8] &str String::from_utf8_unchecked`

Report de l'allocation

Supposons que vous souhaitiez que votre programme accueille l'utilisateur. Sous Unix, vous pouvez écrire :

```
fn get_name() -> String {
    std::env::var("USER") // Windows uses "USERNAME"
        .unwrap_or("whoever you are".to_string())
}

println!("Greetings, {}!", get_name());
```

Pour les utilisateurs d'Unix, cela les accueille par nom d'utilisateur. Pour les utilisateurs de Windows et les personnes tragiquement anonymes, il fournit un texte de stock alternatif.

La fonction renvoie un — et a de bonnes raisons de le faire que nous n'aborderons pas ici. Mais cela signifie que le texte de stock alternatif doit également être retourné en tant que fichier . C'est décevant : lorsqu'une chaîne statique renvoie une chaîne statique, aucune allocation ne devrait être nécessaire. `std::env::var String String get_name`

Le nœud du problème est que parfois la valeur de retour de devrait être une propriété, parfois elle devrait être un , et nous ne pouvons pas savoir lequel ce sera jusqu'à ce que nous exécutions le programme. Ce caractère dynamique est l'indice à considérer pour utiliser , le type de clone sur écriture qui peut contenir des données possédées ou empruntées. `get_name String &'static str std::borrow::Cow`

Comme expliqué dans « Borrow and ToOwned at Work: The Humble Cow », est un enum avec deux variantes: et . détient une référence , et détient la version propriétaire de : pour , pour , et ainsi de suite. Que ce soit ou , un peut toujours produire un pour vous d'utiliser. En fait, déréférence à , se comportant comme une sorte de pointeur

intelligent. `Cow<'a, T> Owned Borrowed Borrowed &'a T Owned &T String &str Vec<i32> & [i32] Owned Borrowed Cow<'a, T> &T Cow<'a, T> &T`

Modification pour renvoyer un résultat dans les éléments suivants : `get_name Cow`

```
use std::borrow::Cow;

fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| Cow::Owned(v))
        .unwrap_or(Cow::Borrowed("whoever you are"))
}
```

Si cela réussit à lire la variable d'environnement, le renvoie le résultat sous la forme d'un fichier . En cas d'échec, le renvoie sa statique sous la forme d'un fichier . L'appelant peut rester inchangé : `"USER" map String Cow::Owned unwrap_or &str Cow::Borrowed`

```
println!("Greetings, {}!", get_name());
```

Tant qu'il implémente le trait, l'affichage d'un produit les mêmes résultats que l'affichage d'un . `T std::fmt::Display Cow<'a, T> T`

`Cow` est également utile lorsque vous avez besoin ou non de modifier un texte que vous avez emprunté. Lorsqu'aucun changement n'est nécessaire, vous pouvez continuer à l'emprunter. Mais le comportement homonyme de clone sur écriture peut vous donner une copie propre et mutable de la valeur à la demande. s'assure que is , en appliquant l'implémentation de la valeur si nécessaire, puis renvoie une référence modifiable à la valeur. `Cow Cow to_mut Cow Cow::Owned ToOwned`

Donc, si vous constatez que certains de vos utilisateurs, mais pas tous, ont des titres par lesquels ils préféreraient être adressés, vous pouvez dire:

```
fn get_title() -> Option<&'static str> { ... }

let mut name = get_name();
if let Some(title) = get_title() {
    name.to_mut().push_str(", ");
    name.to_mut().push_str(title);
}
```

```
println!("Greetings, {}!", name);
```

Cela peut produire une sortie comme suit :

```
$ cargo run
Greetings, jimb, Esq.!
$
```

Ce qui est bien ici, c'est que, si renvoie une chaîne statique et renvoie , le porte simplement la chaîne statique jusqu'au . Vous avez réussi à reporter l'allocation à moins que ce ne soit vraiment nécessaire, tout en écrivant du code simple. `get_name()` `get_title` `None` `Cow` `println!`

Étant donné qu'elle est fréquemment utilisée pour les chaînes, la bibliothèque standard dispose d'un support spécial pour . Il fournit et convertit à la fois et , afin que vous puissiez écrire de manière plus laconique: `Cow` `Cow<'a, str>` `From` `Into` `String` `&str` `get_name`

```
fn get_name() -> Cow<'static, str> {
    std::env::var("USER")
        .map(|v| v.into())
        .unwrap_or("whoever you are".into())
}
```

`Cow<'a, str>` implémente également et , donc pour ajouter le titre au nom, vous pouvez écrire: `std::ops::Add` `std::ops::AddAssign`

```
if let Some(title) = get_title() {
    name += ", ";
    name += title;
}
```

Ou, puisque a peut être la destination d'une macro : `String` `write!`

```
use std::fmt::Write;

if let Some(title) = get_title() {
    write!(name.to_mut(), ", {}", title).unwrap();
}
```

Comme précédemment, aucune allocation ne se produit tant que vous n'avez pas essayé de modifier le fichier . `Cow`

Gardez à l'esprit que tout ne doit pas être : vous pouvez utiliser pour emprunter du texte préalablement calculé jusqu'au moment où une copie devient nécessaire. `Cow<..., str> 'static Cow`

Chaînes en tant que collections génériques

`String` implémente les deux et : renvoie une chaîne vide et peut ajouter des caractères, des tranches de chaîne, des s ou des chaînes à la fin d'une chaîne. Il s'agit de la même combinaison de traits implémentés par les autres types de collection de Rust comme et pour les modèles de construction génériques tels que et
`.std::default::Default std::iter::Extend default extend Cow <..., str> Vec HashMap collect partition`

Le type implémente également , renvoyant une tranche vide. C'est pratique dans certains cas de coin; par exemple, il vous permet de dériver pour des structures contenant des tranches de chaîne. `&str Default Default`

Mise en forme des valeurs

Tout au long du livre, nous avons utilisé des macros de mise en forme de texte comme : `println!`

```
println!("{:.3}µs: relocated {} at {:#x} to {:#x}, {} bytes",
         0.84391, "object",
         140737488346304_usize, 6299664_usize, 64);
```

Cet appel produit la sortie suivante :

```
0.844µs: relocated object at 0x7ffffffdcc0 to 0x602010, 64 bytes
```

Le littéral de chaîne sert de modèle pour la sortie : chacun dans le modèle est remplacé par la forme formatée de l'un des arguments suivants. La chaîne de modèle doit être une constante afin que Rust puisse la comparer aux types d'arguments au moment de la compilation. Chaque argument doit être utilisé ; Rust signale une erreur de compilation dans le cas contraire. `{...}`

Plusieurs fonctionnalités de bibliothèque standard partagent ce petit langage pour la mise en forme des chaînes :

- La macro l'utilise pour construire s. `format! String`
- Les macros et les macros écrivent du texte formaté dans le flux de sortie standard. `println! print!`
- Les macros et l'écrivent dans un flux de sortie désigné. `writeln! write!`
- La macro l'utilise pour construire une expression (idéalement informative) de la consternation terminale. `panic!`

Les installations de formatage de Rust sont conçues pour être ouvertes. Vous pouvez étendre ces macros pour prendre en charge vos propres types en implémentant les traits de mise en forme du module. Et vous pouvez utiliser la macro et le type pour créer vos propres fonctions et macros prenant en charge le langage de formatage. `std::fmt format_args! std::fmt::Arguments`

Les macros de mise en forme empruntent toujours des références partagées à leurs arguments ; ils ne s'en approprient jamais et ne les transforment jamais.

Les formulaires du modèle sont *appelés paramètres de format* et ont le formulaire . Les deux parties sont facultatives; est fréquemment utilisé. `{...} {which:how} {}`

La valeur *`qui`* sélectionne l'argument suivant le modèle doit prendre la place du paramètre. Vous pouvez sélectionner des arguments par index ou par nom. Les paramètres *sans valeur* sont simplement associés à des arguments de gauche à droite.

La valeur *`how`* indique comment l'argument doit être mis en forme : combien de remplissage, à quelle précision, dans quel rayon numérique, etc. Si *`comment`* est présent, le côlon avant qu'il ne soit nécessaire. Le tableau 17-4 présente quelques exemples.

Tableau 17-4. Exemples de chaînes mises en forme

Chaîne de modèle	Liste d'arguments	Résultat
`"number of {}: {}"`	`"elephants", 19`	`"number of elephants: 19"`
`"from {1} to {0}"`	`"the grave", "the cradle"`	`"from the cradle to the grave"`
`"v = {:?}"`	`vec![0,1,2,5,12,29]`	`"v = [0, 1, 2, 5, 12, 29]"`
`"name = {:?}"`	`"Nemo"`	`"name = \"Nemo\""`
`"{:8.2} km/s"`	`11.186`	`"   11.19 km/s"`
`"{:20} {:02x} {:02x}"`	`"adc #42", 105, 42`	`"adc #42 69 2a"`
`"{1:02x} {2:02x}  {0}"`	`"adc #42", 105, 42`	`"69 2a  adc #42"`
`"{lsb:02x} {msb:02x}  {insn}"`	`insn="adc #42", lsb=105, msb=42`	`"69 2a  adc #42"`
`"{:02?}"`	`[110, 11, 9]`	`"[110, 11, 09]"`
`"{:02x?}"`	`[110, 11, 9]`	`"[6e, 0b, 09]"`

Si vous souhaitez inclure ou des caractères dans votre sortie, doublez les caractères du modèle : `{ }`

```
assert_eq!(format!("{{a, c}} ⊂ {{a, b, c}}"),
           "{a, c} ⊂ {a, b, c}");
```

Mise en forme des valeurs de texte

Lors de la mise en forme d'un type textuel comme ou (est traité comme une chaîne à un seul caractère), la valeur *how* d'un paramètre comporte

plusieurs parties, toutes facultatives : `&str String char`

- Limite *de longueur de texte*. Rust tronque votre argument s'il est plus long que cela. Si vous ne spécifiez aucune limite, Rust utilise le texte intégral.
- Largeur *de champ minimale*. Après toute troncature, si votre argument est plus court que cela, Rust le tamponne à droite (par défaut) avec des espaces (par défaut) pour créer un champ de cette largeur. S'il est omis, Rust ne répond pas à votre argumentation.
- Un *alignement*. Si votre argument doit être rembourré pour respecter la largeur minimale du champ, cela indique où votre texte doit être placé dans le champ. , et placez votre texte au début, au milieu et à la fin, respectivement. `< ^ >`
- Caractère *de remplissage* à utiliser dans ce processus de remplissage. S'il est omis, Rust utilise des espaces. Si vous spécifiez le caractère de remplissage, vous devez également spécifier l'alignement.

Le tableau 17-5 illustre quelques exemples montrant comment écrire les choses et leurs effets. Tous utilisent le même argument à huit caractères, . `"bookends"`

Tableau 17-5. Mettre en forme des directives de chaîne pour le texte

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Faire défaut	`"{}"`	`"bookends"`
Largeur minimale du champ	`"{:4}"`	`"bookends"`
	`"{:12}"`	`"bookends "`
Limite de longueur de texte	`"{:.4}"`	`"book"`
	`"{:.12}"`	`"bookends"`
Largeur du champ, limite de longueur	`"{:12.20}"`	`"bookends "`
	`"{:4.20}"`	`"bookends"`
	`"{:4.6}"`	`"booken"`
	`"{:6.4}"`	`"book  "`
Aligné à gauche, largeur	`"{:<12}"`	`"bookends "`
Centré, largeur	`"{:^12}"`	`" bookends "`
Aligné à droite, largeur	`"{:>12}"`	`" booken ds"`
Pad avec , centré, largeur `'='`	`"{:=^12}"`	`"==bookends ==="`
Pad , aligné à droite, largeur, limite `'*'`	`"{:*>12.4}"`	`"********bo ok"`

Le formateur de Rust a une compréhension naïve de la largeur: il suppose que chaque caractère occupe une colonne, sans égard pour combiner des caractères, des katakana demi-largeur, des espaces de largeur nulle ou les autres réalités désordonnées d'Unicode. Par exemple:

```
assert_eq!(format!("{:4}", "th\u{e9}"),   "th\u{e9} ");
assert_eq!(format!("{:4}", "the\u{301}"), "the\u{301}");
```

Bien qu'Unicode indique que ces chaînes sont toutes deux équivalentes à , le formateur de Rust ne sait pas que des caractères comme , COMBINING ACUTE ACCENT, ont besoin d'un traitement spécial. Il tamponne correctement la première chaîne, mais suppose que la seconde a quatre colonnes de large et n'ajoute aucun rembourrage. Bien qu'il soit facile de voir comment Rust pourrait s'améliorer dans ce cas spécifique, la véritable mise en forme de texte multilingue pour tous les scripts Unicode est une tâche monumentale, mieux gérée en s'appuyant sur les boîtes à outils de l'interface utilisateur de votre plate-forme, ou peut-être en générant HTML et CSS et en faisant en sorte qu'un navigateur Web règle tout. Il y a une caisse populaire, , qui gère certains aspects de cela. `"thé" '\u{301}' unicode-width`

Avec et , vous pouvez également passer des types de pointeurs intelligents de macros de mise en forme avec des référents textuels, comme ou , sans cérémonie. `&str String Rc<String> Cow<'a, str>`

Étant donné que les chemins de noms de fichiers ne sont pas nécessairement utF-8 bien formés, ce n'est pas tout à fait un type textuel; vous ne pouvez pas passer un macro directement à une macro de mise en forme. Cependant, la méthode d'a renvoie une valeur que vous pouvez mettre en forme et qui règle les choses d'une manière adaptée à la plate-forme : `std::path::Path std::path::Path Path display`

```
println!("processing file: {}", path.display());
```

Mise en forme des nombres

Lorsque l'argument de mise en forme a un type numérique comme ou , *la valeur du* paramètre comporte les parties suivantes, toutes facultatives : `usize f64`

- Un *rembourrage* et un *alignement*, qui fonctionnent comme ils le font avec les types textuels.
- Un caractère, demandant que le signe du numéro soit toujours affiché, même lorsque l'argument est positif. +
- Caractère demandant un préfixe radix explicite comme ou . Voir la puce « notation » qui conclut cette liste. `# 0x 0b`
- Caractère demandant que la largeur minimale du champ soit satisfaite en incluant des zéros de début dans le nombre, au lieu de l'approche de remplissage habituelle. `0`
- Largeur *de champ minimale*. Si le nombre formaté n'est pas au moins aussi large, Rust le tamponne à gauche (par défaut) avec des espaces (par défaut) pour créer un champ de la largeur donnée.
- *Précision pour* les arguments à virgule flottante, indiquant le nombre de chiffres que Rust doit inclure après la virgule. La rouille arrondit ou s'étend à zéro selon les besoins pour produire exactement autant de chiffres fractionnaires. Si la précision est omise, Rust essaie de représenter avec précision la valeur en utilisant le moins de chiffres possible. Pour les arguments de type entier, la précision est ignorée.
- Une *notation*. Pour les types entiers, cela peut être pour binaire, pour octal, ou pour hexadécimal avec des lettres minuscules ou majuscules. Si vous avez inclus le caractère, il s'agit d'un préfixe radix explicite de style Rust, , , , ou . Pour les types à virgule flottante, un radix de ou demande une notation scientifique, avec un coefficient normalisé, en utilisant ou pour l'exposant. Si vous ne spécifiez aucune notation, Rust met en forme les nombres en décimal. `b o x X # 0b 0o 0x 0X e E e E`

Le tableau 17-6 montre quelques exemples de mise en forme de la valeur `.i32 1234`

Tableau 17-6. Mettre en forme des directives de chaîne pour les entiers

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Faire défaut	`"{}"`	`"1234"`
Signe forcé	`"{:+}"`	`"+1234"`
Largeur minimale du champ	`"{:12}"`	`"        1234"`
	`"{:2}"`	`"1234"`
Signe, largeur	`"{:+12}"`	`"       +1234"`
Zéros de début, largeur	`"{:012}"`	`"000000001234"`
Signe, zéros, largeur	`"{:+012}"`	`"+000000001234"`
Aligné à gauche, largeur	`"{:<12}"`	`"1234        "`
Centré, largeur	`"{:^12}"`	`"    1234    "`
Aligné à droite, largeur	`"{:>12}"`	`"        1234"`
Aligné à gauche, signe, largeur	`"{:<+12}"`	`"+1234       "`
Centré, signe, largeur	`"{:^+12}"`	`"   +1234    "`
Aligné à droite, signe, largeur	`"{:>+12}"`	`"       +1234"`

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Rembourré avec , centré, largeur `'='`	`"{:=^12}"`	`"====1234====="`
Notation binaire	`"{:b}"`	`"10011010010"`
Largeur, notation octale	`"{:12o}"`	`"        2322"`
Signe, largeur, notation hexadécimale	`"{:+12x}"`	`"        +4d2"`
Signe, largeur, hexagone avec chiffres majuscules	`"{:+12X}"`	`"        +4D2"`
Signe, préfixe radix explicite, largeur, hexadécimal	`"{:+#12x}"`	`"      +0x4d2"`
Signe, radix, zéros, largeur, hexagonal	`"{:+#012x}"`	`"+0x00004d2"`
	`"{:+#06x}"`	`"+0x4d2"`

Comme le montrent les deux derniers exemples, la largeur minimale du champ s'applique au nombre entier, au signe, au préfixe radix et à tout.

Les nombres négatifs incluent toujours leur signe. Les résultats sont similaires à ceux présentés dans les exemples de « signes forcés ».

Lorsque vous demandez des zéros de début, les caractères d'alignement et de remplissage sont simplement ignorés, car les zéros développent le nombre pour remplir le champ entier.

En utilisant l'argument , nous pouvons montrer des effets spécifiques aux types à virgule flottante (tableau 17-7). `1234.5678`

Tableau 17-7. Mettre en forme des directives de chaîne pour les nombres à virgule flottante

Fonctionnalités utilisées	Chaîne de modèle	Résultat
Faire défaut	`"{}"`	`"1234.5678"`
Précision	`"{:.2}"`	`"1234.57"`
	`"{:.6}"`	`"1234.567800"`
Largeur minimale du champ	`"{:12}"`	`"  1234.5678"`
Minimum, précision	`"{:12.2}"`	`"     1234.57"`
	`"{:12.6}"`	`" 1234.567800"`
Zéros de début, minimum, précision	`"{:012.6}"`	`"01234.567800"`
Scientifique	`"{:e}"`	`"1.2345678e3"`
Scientifique, précision	`"{:.3e}"`	`"1.235e3"`
Scientifique, minimum, précision	`"{:12.3e}"`	`"    1.235e3"`
	`"{:12.3E}"`	`"    1.235E3"`

Mise en forme d’autres types

Au-delà des chaînes et des nombres, vous pouvez mettre en forme plusieurs autres types de bibliothèque standard :

- Les types d'erreur peuvent tous être formatés directement, ce qui facilite leur inclusion dans les messages d'erreur. Chaque type d'erreur doit implémenter le trait, ce qui étend le trait de mise en forme par défaut . En conséquence, tout type qui implémente est prêt à formater. `std::error::Error` `std::fmt::Display` `Error`
- Vous pouvez formater des types d'adresses de protocole Internet comme et . `std::net::IpAddr` `std::net::SocketAddr`
- Le booléen et les valeurs peuvent être mis en forme, bien que ce ne soient généralement pas les meilleures chaînes à présenter directement aux utilisateurs finaux. `true` `false`

Vous devez utiliser les mêmes types de paramètres de format que pour les chaînes. Les contrôles de limite de longueur, de largeur de champ et d'alignement fonctionnent comme prévu.

Mise en forme des valeurs pour le débogage

Pour faciliter le débogage et la journalisation, le paramètre formate tout type public dans la bibliothèque standard Rust d'une manière destinée à être utile aux programmeurs. Vous pouvez l'utiliser pour inspecter des vecteurs, des tranches, des tuples, des tables de hachage, des threads et des centaines d'autres types. `{:?}`

Par exemple, vous pouvez écrire ce qui suit :

```
use std::collections::HashMap;
let mut map = HashMap::new();
map.insert("Portland", (45.5237606,-122.6819273));
map.insert("Taipei",   (25.0375167, 121.5637));
println!("{:?}", map);
```

Cette impression est :

```
{"Taipei": (25.0375167, 121.5637), "Portland": (45.5237606, -122.68192
```

Les et types savent déjà comment se formater, sans effort de votre part. `HashMap` `(f64, f64)`

Si vous incluez le caractère dans le paramètre format, Rust imprimera joliment la valeur. Changer ce code pour dire conduit à cette sortie: `#` `println!("{:#?}", map)`

```
{
    "Taipei": (
        25.0375167,
        121.5637
    ),
    "Portland": (
        45.5237606,
        -122.6819273
    )
}
```

Ces formes exactes ne sont pas garanties et changent parfois d'une version Rust à l'autre.

Le débogage de la mise en forme imprime généralement des nombres en décimal, mais vous pouvez placer un ou avant le point d'interrogation pour demander hexadécimal à la place. La syntaxe de zéro et de largeur de champ est également respectée. Par exemple, vous pouvez écrire : `x X`

```
println!("ordinary: {:02?}",  [9, 15, 240]);
println!("hex:      {:02x?}", [9, 15, 240]);
```

Cette impression est :

```
ordinary: [09, 15, 240]
hex:      [09, 0f, f0]
```

Comme nous l'avons mentionné, vous pouvez utiliser la syntaxe pour faire fonctionner vos propres types avec : `#[derive(Debug)] {:?}`

```
#[derive(Copy, Clone, Debug)]
struct Complex { re: f64, im: f64 }
```

Avec cette définition en place, nous pouvons utiliser un format pour imprimer des valeurs : `{:?} Complex`

```
let third = Complex { re: -0.5, im: f64::sqrt(0.75) };
println!("{:?}", third);
```

Cette impression est :

```
Complex { re: -0.5, im: 0.8660254037844386 }
```

C’est bien pour le débogage, mais ce serait peut-être bien si vous pouviez les imprimer sous une forme plus traditionnelle, comme . Dans « Formater vos propres types », nous allons montrer comment faire exactement cela. `{} -0.5 + 0.8660254037844386i`

Mise en forme des pointeurs pour le débogage

Normalement, si vous passez n’importe quel type de pointeur à une macro de mise en forme (une référence, un , un), la macro suit simplement le pointeur et met en forme son référent ; le pointeur lui-même n’est pas intéressant. Mais lorsque vous déboguez, il est parfois utile de voir le pointeur : une adresse peut servir de « nom » approximatif pour une valeur individuelle, ce qui peut être éclairant lors de l’examen de structures avec des cycles ou du partage. `Box` `Rc`

La notation met en forme les références, les zones et d’autres types de type pointeur en tant qu’adresses : `{:p}`

```
use std::rc::Rc;
let original = Rc::new("mazurka".to_string());
let cloned = original.clone();
let impostor = Rc::new("mazurka".to_string());
println!("text:     {}, {}, {}",       original, cloned, impostor);
println!("pointers: {:p}, {:p}, {:p}", original, cloned, impostor);
```

Ce code imprime :

```
text:     mazurka, mazurka, mazurka
pointers: 0x7f99af80e000, 0x7f99af80e000, 0x7f99af80e030
```

Bien sûr, les valeurs de pointeur spécifiques varient d’une exécution à l’autre, mais même ainsi, la comparaison des adresses indique clairement que les deux premières sont des références à la même , tandis que la troisième pointe vers une valeur distincte. `String`

Les adresses ont tendance à ressembler à de la soupe hexadécimale, donc des visualisations plus raffinées peuvent en valoir la peine, mais le style peut toujours être une solution efficace rapide et sale. `{:p}`

Référence aux arguments par index ou par nom

Un paramètre de format peut sélectionner explicitement l'argument qu'il utilise. Par exemple:

```
assert_eq!(format!("{1},{0},{2}", "zeroth", "first", "second"),
           "first,zeroth,second");
```

Vous pouvez inclure des paramètres de format après deux points :

```
assert_eq!(format!("{2:#06x},{1:b},{0:=>10}", "first", 10, 100),
           "0x0064,1010,=====first");
```

Vous pouvez également sélectionner des arguments par nom. Cela rend les modèles complexes avec de nombreux paramètres beaucoup plus lisibles. Par exemple:

```
assert_eq!(format!("{description:.<25}{quantity:2} @ {price:5.2}",
                   price=3.25,
                   quantity=3,
                   description="Maple Turmeric Latte"),
           "Maple Turmeric Latte..... 3 @  3.25");
```

(Les arguments nommés ici ressemblent à des arguments de mots-clés en Python, mais il ne s'agit que d'une caractéristique spéciale des macros de mise en forme, qui ne fait pas partie de la syntaxe d'appel de fonction de Rust.)

Vous pouvez mélanger des paramètres indexés, nommés et positionnels (c'est-à-dire sans index ni nom) en une seule utilisation de macro de mise en forme. Les paramètres positionnels sont associés à des arguments de gauche à droite comme si les paramètres indexés et nommés n'étaient pas là :

```
assert_eq!(format!("{mode} {2} {} {}",
                   "people", "eater", "purple", mode="flying"),
           "flying purple people eater");
```

Les arguments nommés doivent apparaître à la fin de la liste.

Largeurs et précisions dynamiques

La largeur minimale de champ, la limite de longueur de texte et la précision numérique d'un paramètre ne doivent pas toujours être des valeurs

fixes ; vous pouvez les choisir au moment de l'exécution.

Nous avons examiné des cas comme cette expression, qui vous donne la chaîne justifiée à droite dans un champ de 20 caractères de large : `content`

```
format!("{:>20}", content)
```

Mais si vous souhaitez choisir la largeur du champ au moment de l'exécution, vous pouvez écrire :

```
format!("{:>1$}", content, get_width())
```

L'écriture pour la largeur minimale du champ indique d'utiliser la valeur du deuxième argument comme largeur. L'argument cité doit être un fichier . Vous pouvez également faire référence à l'argument par son nom : `1$ format! usize`

```
format!("{:>width$}", content, width=get_width())
```

La même approche fonctionne également pour la limite de longueur du texte :

```
format!("{:>width$.limit$}", content,
        width=get_width(), limit=get_limit())
```

Au lieu de la limite de longueur de texte ou de la précision en virgule flottante, vous pouvez également écrire , ce qui indique de prendre l'argument positionnel suivant comme précision. Les clips suivants à au plus des caractères : `* content get_limit()`

```
format!("{:.*}", get_limit(), content)
```

L'argument pris comme précision doit être un . Il n'existe pas de syntaxe correspondante pour la largeur du champ. `usize`

Formatage de vos propres types

Les macros de mise en forme utilisent un ensemble de traits définis dans le module pour convertir les valeurs en texte. Vous pouvez faire en sorte

que les macros de formatage de Rust formatent vos propres types en implémentant vous-même un ou plusieurs de ces traits. `std::fmt`

La notation d'un paramètre de format indique le trait que le type de son argument doit implémenter, comme illustré dans le tableau 17-8.

Tableau 17-8. Notation de directive de chaîne de format

Notation	Exemple	Trait	But
aucun	`{}`	`std::fmt::Display`	Texte, chiffres, erreurs : le trait fourre-tout
`b`	`{bits:#b}`	`std::fmt::Binary`	Nombres en binaire
`o`	`{:#5o}`	`std::fmt::Octal`	Nombres en octal
`x`	`{:4x}`	`std::fmt::LowerHex`	Nombres en chiffres hexadécimaux en minuscules
`X`	`{:016X}`	`std::fmt::UpperHex`	Nombres en chiffres hexadécimaux, majuscules
`e`	`{:.3e}`	`std::fmt::LowerExp`	Nombres à virgule flottante en notation scientifique
`E`	`{:.3E}`	`std::fmt::UpperExp`	Idem, majuscule `E`
`?`	`{:#?}`	`std::fmt::Debug`	Vue de débogage, pour les développeurs
`p`	`{:p}`	`std::fmt::Pointer`	Pointeur comme adresse, pour les développeurs

Lorsque vous placez l'attribut sur une définition de type afin de pouvoir utiliser le paramètre format, vous demandez simplement à Rust d'implémenter le trait pour vous. `#[derive(Debug)]` `{:?}` `std::fmt::Debug`

Les traits de mise en forme ont tous la même structure, ne différant que par leurs noms. Nous utiliserons en tant que représentant : `std::fmt::Display`

```
trait Display {
    fn fmt(&self, dest: &mut std::fmt::Formatter)
        -> std::fmt::Result;
}
```

Le travail de la méthode consiste à produire une représentation correctement formatée et à écrire ses caractères sur . En plus de servir de flux de sortie, l'argument contient également des détails analysés à partir du paramètre de format, tels que l'alignement et la largeur minimale du champ. `fmt self dest dest`

Par exemple, plus haut dans ce chapitre, nous avons suggéré qu'il serait bien que les valeurs s'impriment elles-mêmes sous la forme habituelle. Voici une implémentation qui fait cela : `Complex a + bi Display`

```
use std::fmt;

impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let im_sign = if self.im < 0.0 { '-' } else { '+' };
        write!(dest, "{} {} {}i", self.re, im_sign, f64::abs(self.im))
    }
}
```

Cela tire parti du fait qu'il s'agit lui-même d'un flux de sortie, de sorte que la macro peut faire la majeure partie du travail pour nous. Avec cette implémentation en place, nous pouvons écrire ce qui suit: `Formatter write!`

```
let one_twenty = Complex { re: -0.5, im: 0.866 };
assert_eq!(format!("{}", one_twenty),
           "-0.5 + 0.866i");

let two_forty = Complex { re: -0.5, im: -0.866 };
assert_eq!(format!("{}", two_forty),
           "-0.5 - 0.866i");
```

Il est parfois utile d'afficher des nombres complexes sous forme polaire : si vous imaginez une ligne tracée sur le plan complexe de l'origine au

nombre, la forme polaire donne la longueur de la ligne et son angle dans le sens des aiguilles d'une montre par rapport à l'axe des x positif. Le caractère d'un paramètre de format sélectionne généralement une autre forme d'affichage ; la mise en œuvre pourrait le traiter comme une demande d'utilisation de la forme polaire : `# Display`

```
impl fmt::Display for Complex {
    fn fmt(&self, dest: &mut fmt::Formatter) -> fmt::Result {
        let (re, im) = (self.re, self.im);
        if dest.alternate() {
            let abs = f64::sqrt(re * re + im * im);
            let angle = f64::atan2(im, re) / std::f64::consts::PI * 18
            write!(dest, "{} ∠ {}°", abs, angle)
        } else {
            let im_sign = if im < 0.0 { '-' } else { '+' };
            write!(dest, "{} {} {}i", re, im_sign, f64::abs(im))
        }
    }
}
```

À l'aide de cette implémentation :

```
let ninety = Complex { re: 0.0, im: 2.0 };
assert_eq!(format!("{}", ninety),
           "0 + 2i");
assert_eq!(format!("{:#}", ninety),
           "2 ∠ 90°");
```

Bien que les méthodes des traits de mise en forme renvoient une valeur (un type typique spécifique au module), vous ne devez propager les échecs qu'à partir des opérations sur le , comme le fait l'implémentation avec ses appels à ; vos fonctions de formatage ne doivent jamais générer d'erreurs elles-mêmes. Cela permet aux macros de renvoyer simplement un au lieu d'un , car l'ajout du texte formaté à un n'échoue jamais. Il garantit également que toutes les erreurs que vous obtenez ou reflètent des problèmes réels du flux d'E/S sous-jacent, et non des problèmes de formatage. `fmt fmt::Result Result Formatter fmt::Display writ e! format! String Result<String, ...> String write! writeln!`

`Formatter` a beaucoup d'autres méthodes utiles, y compris certaines pour gérer les données structurées comme les cartes, les listes, etc., que

nous ne couvrirons pas ici; consultez la documentation en ligne pour tous les détails.

Utilisation du langage de mise en forme dans votre propre code

Vous pouvez écrire vos propres fonctions et macros qui acceptent les modèles de format et les arguments en utilisant la macro de Rust et le type. Par exemple, supposons que votre programme doit consigner les messages d'état au fur et à mesure de son exécution et que vous souhaitez utiliser le langage de formatage de texte de Rust pour les produire. Ce qui suit serait un début: `format_args! std::fmt::Arguments`

```
fn logging_enabled() -> bool { ... }

use std::fs::OpenOptions;
use std::io::Write;

fn write_log_entry(entry: std::fmt::Arguments) {
    if logging_enabled() {
        // Keep things simple for now, and just
        // open the file every time.
        let mut log_file = OpenOptions::new()
            .append(true)
            .create(true)
            .open("log-file-name")
            .expect("failed to open log file");

        log_file.write_fmt(entry)
            .expect("failed to write to log");
    }
}
```

Vous pouvez appeler comme ça: `write_log_entry`

```
write_log_entry(format_args!("Hark! {:?}\n", mysterious_value));
```

Au moment de la compilation, la macro analyse la chaîne de modèle et la compare aux types des arguments, signalant une erreur en cas de problème. Au moment de l'exécution, il évalue les arguments et crée une valeur contenant toutes les informations nécessaires à la mise en forme du texte : une forme prédéfinie du modèle, ainsi que des références partagées aux valeurs de l'argument. `format_args! Arguments`

Construire une valeur est bon marché: c'est juste rassembler quelques conseils. Aucun travail de formatage n'a encore lieu, seulement la collecte des informations nécessaires pour le faire plus tard. Cela peut être important : si la journalisation n'est pas activée, tout temps passé à convertir des nombres en décimales, à remplir des valeurs, etc. serait gaspillé. `Arguments`

Le type implémente le trait, dont la méthode prend un et effectue la mise en forme. Il écrit les résultats dans le flux sous-jacent. `File std::io::Write write_fmt Argument`

Cet appel à n'est pas joli. C'est là qu'une macro peut vous aider : `write_log_entry`

```
macro_rules! log { // no ! needed after name in macro definitions
    ($format:tt, $($arg:expr),*) => (
        write_log_entry(format_args!($format, $($arg),*))
    )
}
```

Nous couvrons les macros en détail au chapitre 21. Pour l'instant, croyez que cela définit une nouvelle macro qui transmet ses arguments à et appelle ensuite votre fonction sur la valeur résultante. Les macros de formatage comme , et sont toutes à peu près la même idée. `log! format_args! write_log_entry Arguments println! writeln! format!`

Vous pouvez utiliser comme suit: `log!`

```
log!("O day and night, but this is wondrous strange! {:?}\n",
     mysterious_value);
```

Idéalement, cela a l'air un peu mieux.

Expressions régulières

La caisse externe est la bibliothèque d'expressions régulières officielle de Rust. Il fournit les fonctions habituelles de recherche et de correspondance. Il a un bon support pour Unicode, mais il peut également rechercher des chaînes d'octets. Bien qu'il ne prenne pas en charge certaines fonctionnalités que vous trouverez souvent dans d'autres packages d'expression régulière, comme les backreferences et les modèles de

recherche, ces simplifications permettent de s'assurer que les recherches prennent du temps linéaire dans la taille de l'expression et dans la longueur du texte recherché. Ces garanties, entre autres, rendent l'utilisation sûre même avec des expressions non fiables recherchant du texte non fiable. `regex regex regex`

Dans ce livre, nous ne fournirons qu'un aperçu de ; vous devriez consulter sa documentation en ligne pour plus de détails. `regex`

Bien que la caisse ne soit pas dans , elle est maintenue par l'équipe de la bibliothèque Rust, le même groupe responsable de . Pour utiliser , placez la ligne suivante dans la section du fichier *Cargo.toml* de votre caisse : `regex std std regex [dependencies]`

```
regex = "1"
```

Dans les sections suivantes, nous supposerons que vous avez ce changement en place.

Utilisation de base de Regex

Une valeur représente une expression régulière analysée prête à l'emploi. Le constructeur tente d'analyser a en tant qu'expression régulière et renvoie un : `Regex Regex::new &str Result`

```
use regex::Regex;

// A semver version number, like 0.2.1.
// May contain a pre-release version suffix, like 0.2.1-alpha.
// (No build metadata suffix, for brevity.)
//
// Note use of r"..." raw string syntax, to avoid backslash blizzard.
let semver = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")?;

// Simple search, with a Boolean result.
let haystack = r#"regex = "0.2.5""#;
assert!(semver.is_match(haystack));
```

La méthode recherche la première correspondance dans une chaîne et renvoie une valeur contenant des informations de correspondance pour chaque groupe de l'expression : `Regex::captures regex::Captures`

```
// You can retrieve capture groups:
let captures = semver.captures(haystack)
    .ok_or("semver regex should have matched")?;
assert_eq!(&captures[0], "0.2.5");
assert_eq!(&captures[1], "0");
assert_eq!(&captures[2], "2");
assert_eq!(&captures[3], "5");
```

L'indexation d'une valeur panique si le groupe demandé ne correspond pas. Pour tester si un groupe particulier correspond, vous pouvez appeler , qui renvoie un fichier . Une valeur enregistre la correspondance d'un seul groupe
: `Captures Captures::get Option<regex::Match> Match`

```
assert_eq!(captures.get(4), None);
assert_eq!(captures.get(3).unwrap().start(), 13);
assert_eq!(captures.get(3).unwrap().end(), 14);
assert_eq!(captures.get(3).unwrap().as_str(), "5");
```

Vous pouvez itérer sur toutes les correspondances d'une chaîne :

```
let haystack = "In the beginning, there was 1.0.0. \
                For a while, we used 1.0.1-beta, \
                but in the end, we settled on 1.2.4.";

let matches: Vec<&str> = semver.find_iter(haystack)
    .map(|match_| match_.as_str())
    .collect();
assert_eq!(matches, vec!["1.0.0", "1.0.1-beta", "1.2.4"]);
```

L'itérateur produit une valeur pour chaque correspondance sans chevauchement de l'expression, en travaillant du début à la fin de la chaîne. La méthode est similaire, mais produit des valeurs enregistrant tous les groupes de capture. La recherche est plus lente lorsque les groupes de capture doivent être signalés, donc si vous n'en avez pas besoin, il est préférable d'utiliser l'une des méthodes qui ne les renvoie pas. `find_iter Match captures_iter Captures`

Construire les valeurs Regex paresseusement

Le constructeur peut être coûteux : la construction d'une expression régulière de 1 200 caractères peut prendre près d'une milliseconde sur

une machine de développement rapide, et même une expression triviale prend des microsecondes. Il est préférable de garder la construction hors des boucles de calcul lourdes; au lieu de cela, vous devriez construire votre une fois, puis réutiliser le même. `Regex::new Regex Regex Regex`

La caisse offre un bon moyen de construire des valeurs statiques paresseusement la première fois qu'elles sont utilisées. Pour commencer, notez la dépendance dans votre fichier *Cargo.toml* : `lazy_static`

```
[dependencies]
lazy_static = "1"
```

Cette caisse fournit une macro pour déclarer ces variables :

```
use lazy_static::lazy_static;

lazy_static! {
    static ref SEMVER: Regex
        = Regex::new(r"(\d+)\.(\d+)\.(\d+)(-[-.[:alnum:]]*)?")
              .expect("error parsing regex");
}
```

La macro se développe à une déclaration d'une variable statique nommée , mais son type n'est pas exactement . Au lieu de cela, il s'agit d'un type généré par macro qui implémente et expose donc toutes les mêmes méthodes qu'un fichier . La première fois est déréférencée, l'initialiseur est évalué et la valeur est enregistrée pour une utilisation ultérieure. Puisqu'il s'agit d'une variable statique, et pas seulement d'une variable locale, l'initialiseur s'exécute au plus une fois par exécution de programme. `SEMVER Regex Deref<Target=Regex> Regex SEMVER SEMVER`

Avec cette déclaration en place, l'utilisation est simple: `SEMVER`

```
use std::io::BufRead;

let stdin = std::io::stdin();
for line_result in stdin.lock().lines() {
    let line = line_result?;
    if let Some(match_) = SEMVER.find(&line) {
        println!("{}", match_.as_str());
    }
}
```

Vous pouvez placer la déclaration dans un module, ou même à l'intérieur de la fonction qui utilise le , si c'est l'étendue la plus appropriée. L'expression régulière n'est toujours compilée qu'une seule fois par exécution de programme. `lazy_static! Regex`

Normalisation

La plupart des utilisateurs considéreraient que le mot Français pour le thé, *thé*, est long de trois caractères. Cependant, Unicode a en fait deux façons de représenter ce texte :

- Dans la forme *composée*, *« thé »* comprend les trois caractères , , et , où est un seul caractère Unicode avec point de code . `'t' 'h' 'é' 'é' 0xe9`
- Dans la forme *décomposée*, *« thé »* comprend les quatre caractères , , , et , où le est le caractère ASCII simple, sans accent, et le point de code est le caractère « COMBINING ACUTE ACCENT », qui ajoute un accent aigu à n'importe quel caractère qu'il suit. `'t' 'h' 'e' '\u{301}' 'e' 0x301`

Unicode ne considère ni la forme composée ni la forme décomposée de *é* comme la « bonne » ; elle les considère plutôt comme des représentations équivalentes du même caractère abstrait. Unicode indique que les deux formulaires doivent être affichés de la même manière, et les méthodes de saisie de texte sont autorisées à produire l'un ou l'autre, de sorte que les utilisateurs ne sauront généralement pas quel formulaire ils consultent ou tapent. (Rust vous permet d'utiliser des caractères Unicode directement dans les littéraux de chaîne, de sorte que vous pouvez simplement écrire si vous ne vous souciez pas de l'encodage que vous obtenez. Ici, nous allons utiliser les échappements pour plus de clarté.) `"thé" \u`

Cependant, considéré comme rouille ou valeurs, et sont complètement distincts. Ils ont des longueurs différentes, se comparent comme inégaux, ont des valeurs de hachage différentes et s'ordonnent différemment par rapport aux autres chaînes : `&str String "th\u{e9}" "the\u{301}"`

```
assert!("th\u{e9}" != "the\u{301}");
assert!("th\u{e9}" >  "the\u{301}");

// A Hasher is designed to accumulate the hash of a series of values,
// so hashing just one is a bit clunky.
```

```
use std::hash::{Hash, Hasher};
use std::collections::hash_map::DefaultHasher;
fn hash<T: ?Sized + Hash>(t: &T) -> u64 {
    let mut s = DefaultHasher::new();
    t.hash(&mut s);
    s.finish()
}

// These values may change in future Rust releases.
assert_eq!(hash("th\u{e9}"),   0x53e2d0734eb1dff3);
assert_eq!(hash("the\u{301}"), 0x90d837f0a0928144);
```

De toute évidence, si vous avez l'intention de comparer le texte fourni par l'utilisateur ou de l'utiliser comme clé dans une table de hachage ou un arbre B, vous devrez d'abord mettre chaque chaîne sous une forme canonique.

Heureusement, Unicode spécifie des formulaires *normalisés* pour les chaînes. Chaque fois que deux chaînes doivent être traitées comme équivalentes selon les règles d'Unicode, leurs formes normalisées sont identiques caractère par caractère. Lorsqu'ils sont codés en UTF-8, ils sont identiques octet pour octet. Cela signifie que vous pouvez comparer des chaînes normalisées avec , les utiliser comme clés dans un ou , et ainsi de suite, et vous obtiendrez la notion d'égalité
d'Unicode. `== HashMap HashSet`

L'échec de la normalisation peut même avoir des conséquences sur la sécurité. Par exemple, si votre site Web normalise les noms d'utilisateur dans certains cas mais pas dans d'autres, vous pourriez vous retrouver avec deux utilisateurs distincts nommés , que certaines parties de votre code traitent comme le même utilisateur, mais d'autres distinguent, ce qui entraîne l'extension incorrecte des privilèges de l'un à l'autre. Bien sûr, il existe de nombreuses façons d'éviter ce genre de problème, mais l'histoire montre qu'il existe également de nombreuses façons de ne pas le faire. `bananasflambé`

Formulaires de normalisation

Unicode définit quatre formes normalisées, chacune étant appropriée pour différentes utilisations. Il y a deux questions auxquelles il faut répondre :

- Tout d'abord, préférez-vous que les personnages soient aussi *composés* que possible ou aussi *décomposés* que possible ?
  Par exemple, la représentation la plus composée du mot vietnamien *Phở* est la chaîne à trois caractères , où la marque tonale ̉et la marque voyelle ̛sont appliquées au caractère de base « o » dans un seul caractère Unicode, , qu'Unicode nomme consciencieusement PETITE LETTRE LATINE O AVEC CORNE ET CROCHET CI-
  DESSUS. `"Ph\u{1edf}" '\u{1edf}'`
  La représentation la plus décomposée divise la lettre de base et ses deux marques en trois caractères Unicode distincts : , (COMBINAISON DE CORNE) et (COMBINAISON DE CROCHET CI-DESSUS), ce qui donne . (Chaque fois que les marques combinées apparaissent en tant que caractères distincts, plutôt que dans le cadre d'un caractère composé, toutes les formes normalisées spécifient un ordre fixe dans lequel elles doivent apparaître, de sorte que la normalisation est bien spécifiée même lorsque les caractères ont plusieurs
  accents.) `'o' '\u{31b}' '\u{309}' "Pho\u{31b}\u{309}"`
  La forme composée a généralement moins de problèmes de compatibilité, car elle correspond plus étroitement aux représentations que la plupart des langues utilisaient pour leur texte avant l'établissement d'Unicode. Il peut également mieux fonctionner avec des fonctionnalités de formatage de chaîne naïves comme la macro de Rust. La forme décomposée, en revanche, peut être meilleure pour l'affichage du texte ou la recherche, car elle rend la structure détaillée du texte plus
  explicite. `format!`
- La deuxième question est la suivante : si deux séquences de caractères représentent le même texte fondamental mais diffèrent dans la façon dont le texte doit être formaté, voulez-vous les traiter comme équivalentes ou les garder distinctes ?
  Unicode a des caractères distincts pour le chiffre ordinaire, le chiffre en exposant (ou) et le chiffre encerclé (ou), mais déclare que les trois sont *équivalents* à la compatibilité. De même, Unicode a un seul caractère pour la ligature *ffi* (), mais déclare que c'est une compatibilité équivalente à la séquence à trois caractères
  `.5 ⁵ '\u{2075}' ⑤ '\u{2464}' '\u{fb03}' ffi`
  L'équivalence de compatibilité est logique pour les recherches : une recherche pour , en utilisant uniquement des caractères ASCII, doit correspondre à la chaîne , qui utilise la ligature *ffi*. L'application de la décomposition de compatibilité à cette dernière chaîne remplacerait la

ligature par les trois lettres simples, ce qui faciliterait la recherche. Mais la normalisation du texte à une forme équivalente à la compatibilité peut perdre des informations essentielles, il ne doit donc pas être appliqué négligemment. Par exemple, il serait incorrect dans la plupart des contextes de stocker en tant que
."difficult" "di\u{fb03}cult" "ffi" "2⁵" "25"

Unicode Normalization Form C et Normalization Form D (NFC et NFD) utilisent les formes composées au maximum et décomposées au maximum de chaque caractère, mais n'essayez pas d'unifier des séquences équivalentes de compatibilité. Les formulaires de normalisation NFKC et NFKD sont comme NFC et NFD, mais normalisent toutes les séquences équivalentes de compatibilité à un simple représentant de leur classe.

Le « Character Model For the World Wide Web » du World Wide Web Consortium recommande d'utiliser NFC pour tous les contenus. L'annexe Unicode Identifier and Pattern Syntax suggère d'utiliser NFKC pour les identificateurs dans les langages de programmation et propose des principes pour adapter le formulaire si nécessaire.

La caisse de normalisation unicode

La caisse de Rust fournit un trait qui ajoute des méthodes pour mettre le texte dans l'une des quatre formes normalisées. Pour l'utiliser, ajoutez la ligne suivante à la section de votre fichier *Cargo.toml* : `unicode-normalization &str [dependencies]`

```
unicode-normalization = "0.1.17"
```

Avec cette déclaration en place, a quatre nouvelles méthodes qui renvoient des itérateurs sur une forme normalisée particulière de la chaîne : `&str`

```
use unicode_normalization::UnicodeNormalization;

// No matter what representation the left-hand string uses
// (you shouldn't be able to tell just by looking),
// these assertions will hold.
assert_eq!("Phở".nfd().collect::<String>(), "Pho\u{31b}\u{309}");
assert_eq!("Phở".nfc().collect::<String>(), "Ph\u{1edf}");
```

```
// The left-hand side here uses the "ffi" ligature character.
assert_eq!("① Di\u{fb03}culty".nfkc().collect::<String>(), "1 Difficul
```

Prendre une chaîne normalisée et la normaliser à nouveau sous la même forme est garanti pour renvoyer un texte identique.

Bien que toute sous-chaîne d'une chaîne normalisée soit elle-même normalisée, la concaténation de deux chaînes normalisées n'est pas nécessairement normalisée : par exemple, la deuxième chaîne peut commencer par combiner des caractères qui doivent être placés avant de combiner des caractères à la fin de la première chaîne.

Tant qu'un texte n'utilise pas de points de code non attribués lorsqu'il est normalisé, Unicode promet que sa forme normalisée ne changera pas dans les futures versions de la norme. Cela signifie que les formulaires normalisés peuvent généralement être utilisés en toute sécurité dans le stockage persistant, même si la norme Unicode évolue.