Chapitre 3. Types fondamentaux

> *Il existe de nombreux types de livres dans le monde, ce qui est logique, car il existe de très nombreux types de personnes et tout le monde veut lire quelque chose de différent.*
>
> —Snicket citronné

Dans une large mesure, le langage Rust est conçu autour de ses types. Sa prise en charge d'un code haute performance découle du fait qu'il permet aux développeurs de choisir la représentation des données qui correspond le mieux à la situation, avec le bon équilibre entre simplicité et coût. Les garanties de sécurité de la mémoire et des threads de Rust reposent également sur la solidité de son système de types, et la flexibilité de Rust découle de ses types et traits génériques.

Ce chapitre couvre les types fondamentaux de Rust pour représenter les valeurs. Ces types au niveau de la source ont des équivalents concrets au niveau de la machine avec des coûts et des performances prévisibles. Bien que Rust ne promette pas qu'il représentera les choses exactement comme vous l'avez demandé, il prend soin de ne s'écarter de vos demandes que lorsqu'il s'agit d'une amélioration fiable.

Comparé à un langage typé dynamiquement comme JavaScript ou Python, Rust nécessite plus de planification de votre part. Vous devez préciser les types d'arguments de fonction et les valeurs de retour, les champs de structure et quelques autres constructions. Cependant, deux fonctionnalités de Rust rendent cela moins problématique que prévu :

- Étant donné les types que vous épelez, l' *inférence de type de Rust*trouvera la plupart du reste pour vous. En pratique, il n'y a souvent qu'un seul type qui fonctionnera pour une variable ou une expression donnée ; lorsque c'est le cas, Rust vous permet de laisser de côté, ou d' *elide* , le type. Par exemple, vous pouvez épeler chaque type dans une fonction, comme ceci :

```
fn build_vector() -> Vec<i16> {
    let mut v: Vec<i16> = Vec:: <i16>::new();
    v.push(10i16);
    v.push(20i16);
    v
}
```

Mais c'est encombré et répétitif. Étant donné le type de retour de la fonction, il est évident que `v` doit être a `Vec<i16>`, un vecteur d'entiers signés 16 bits ; aucun autre type ne fonctionnerait. Et de là, il s'ensuit que chaque élément du vecteur doit être un `i16`. C'est exactement le genre de raisonnement que l'inférence de type de Rust applique, vous permettant d'écrire à la place :

```
fn build_vector() -> Vec<i16> {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v
}
```

Ces deux définitions sont exactement équivalentes et Rust générera le même code machine dans les deux cas. L'inférence de type redonne une grande partie de la lisibilité des langages à typage dynamique, tout en capturant les erreurs de type au moment de la compilation.

- Les fonctions peuvent être *génériques*: une seule fonction peut travailler sur des valeurs de plusieurs types différents.
  En Python et JavaScript, toutes les fonctions fonctionnent naturellement de cette façon : une fonction peut opérer sur n'importe quelle valeur qui possède les propriétés et les méthodes dont la fonction aura besoin. (C'est la caractéristique souvent appelée *duck typing*: s'il cancane comme un canard, c'est un canard.) Mais c'est précisément cette flexibilité qui rend si difficile pour ces langages la détection précoce des erreurs de type ; les tests sont souvent le seul moyen de détecter de telles erreurs. Les fonctions génériques de Rust donnent au langage le même degré de flexibilité, tout en captant toutes les erreurs de type au moment de la compilation.
  Malgré leur flexibilité, les fonctions génériques sont tout aussi efficaces que leurs homologues non génériques. Il n'y a aucun avantage inhérent en termes de performances à tirer de l'écriture, par exemple, d'une `sum` fonction spécifique pour chaque entier par rapport à l'écriture d'une fonction générique qui gère tous les entiers. Nous aborderons les fonctions génériques en détail au chapitre 11 .

Le reste de ce chapitre couvre les types de Rust de bas en haut, en commençant par les types numériques simples comme les entiers et les valeurs à virgule flottante, puis en passant aux types qui contiennent plus de données : boîtes, tuples, tableaux et chaînes.

Voici un résumé des types de types que vous verrez dans Rust. Le tableau 3-1 montre les types primitifs de Rust, certains types très courants de la bibliothèque standard et quelques exemples de types définis par l'utilisateur.

Tableau 3-1. Exemples de types en Rust

Taper	La description	Valeurs
`i8`, `i16`, `i32`, `i64`, `i128` `u8`, `u16`, `u32`, `u64`,`u128`	Entiers signés et non signés, de largeur de bit donnée	`42`, `-5i8`, `0x400u16`, `0o100i16`, `20_922_789_888_000u64`, `b'*'` (`u8` octet littéral)
`isize`,`usize`	Entiers signés et non signés, de même taille qu'une adresse sur la machine (32 ou 64 bits)	`137`, `-0b0101_0010isize`, `0xffff_fc00usize`
`f32`,`f64`	Nombres à virgule flottante IEEE, simple et double précision	`1.61803`, `3.14f32`, `6.0221e23f64`
`bool`	booléen	`true`,`false`
`char`	Caractère Unicode, largeur 32 bits	`'*'`, `'\n'`, `'字'`, `'\x7f'`,`'\u{CA0}'`
`(char, u8, i32)`	Tuple : types mixtes autorisés	`('%', 0x7f, -1)`
`()`	"Unité" (tuple vide)	`()`
`struct S { x: f32, y: f32 }`	Structure de champ nommé	`S { x: 120.0, y: 209.0 }`

Taper	La description	Valeurs
`struct T (i32, char);`	Structure de type tuple	`T(120, 'X')`
`struct E;`	Structure de type unité ; n'a pas de champs	`E`
`enum Attend { OnTime, Late(u32) }`	Énumération, type de données algébrique	`Attend::Late(5)`, `Attend::OnTime`
`Box<Attend>`	Boîte : posséder un pointeur vers la valeur dans le tas	`Box::new(Late(15))`
`&i32`, `&mut i32`	Références partagées et mutables : pointeurs non propriétaires qui ne doivent pas survivre à leur référent	`&s.y`, `&mut v`
`String`	Chaîne UTF-8, dimensionnée dynamiquement	`"ラーメン: ramen".to_string()`
`&str`	Référence à `str` : pointeur non propriétaire vers le texte UTF-8	`"そば: soba"`, `&s[0..12]`
`[f64; 4]`, `[u8; 256]`	Array, longueur fixe ; éléments tous du même type	`[1.0, 0.0, 0.0, 1.0]`, `[b' '; 256]`
`Vec<f64>`	Vecteur, longueur variable ; éléments tous du même type	`vec![0.367, 2.718, 7.389]`

Taper	La description	Valeurs
`&[u8]`, `&mut [u8]`	Référence à la tranche : référence à une partie d'un tableau ou d'un vecteur, comprenant un pointeur et une longueur	`&v[10..20]`, `&mut a[..]`
`Option<&str>`	Valeur facultative : soit `None` (absent), soit `Some(v)` (présent, avec la valeur `v`)	`Some("Dr."), None`
`Result<u64, Error>`	Résultat de l'opération qui peut échouer : soit une valeur de succès `Ok(v)`, soit une erreur `Err(e)`	`Ok(4096), Err(Error::last_os_error())`
`&dyn Any`, `&mut dyn Read`	Objet trait : référence à toute valeur qui implémente un ensemble donné de méthodes	`value as &dyn Any`, `&mut file as &mut dyn Read`
`fn(&str) -> bool`	Pointeur vers la fonction	`str::is_empty`
(Les types de fermeture n'ont pas de forme écrite)	Fermeture	`|a, b| { a*a + b*b }`

La plupart de ces types sont traités dans ce chapitre, à l'exception des suivants :

- Nous donnons aux `struct` types leur propre chapitre, Chapitre 9 .
- Nous donnons aux types énumérés leur propre chapitre, le chapitre 10 .
- Nous décrivons les objets trait au chapitre 11 .
- Nous décrivons l'essentiel de `String` et `&str` ici, mais fournissons plus de détails au chapitre 17 .
- Nous couvrons les types de fonctions et de fermetures au chapitre 14 .

Types numériques à largeur fixe

La base du système de types de Rust est une collection de types numériques à largeur fixe, choisis pour correspondre aux types que presque tous les processeurs modernes implémentent directement dans le matériel.

Les types numériques à largeur fixe peuvent déborder ou perdre en précision, mais ils conviennent à la plupart des applications et peuvent être des milliers de fois plus rapides que des représentations telles que des entiers à précision arbitraire et des rationnels exacts. Si vous avez besoin de ces types de représentations numériques, elles sont prises en charge dans la `num` caisse.

Les noms des types numériques de Rust suivent un modèle régulier, épelant leur largeur en bits et la représentation qu'ils utilisent (Tableau 3-2).

Tableau 3-2. Types numériques de rouille

Taille (bits)	Entier non signé	Entier signé	Point flottant
8	`u8`	`i8`	
16	`u16`	`i16`	
32	`u32`	`i32`	`f32`
64	`u64`	`i64`	`f64`
128	`u128`	`i128`	
Mot machine	`usize`	`isize`	

Ici, un *mot machine*est une valeur de la taille d'une adresse sur la machine sur laquelle le code s'exécute, 32 ou 64 bits.

Types entiers

Rust n'est pas signétypes entiersutiliser leur plage complète pour représenter les valeurs positives et zéro (tableau 3-3).

Tableau 3-3. Rust types entiers non signés

Taper	Intervalle
`u8`	0 à 2^{8} –1 (0 à 255)
`u16`	0 à 2^{16} –1 (0 à 65 535)
`u32`	0 à 2^{32} –1 (0 à 4 294 967 295)
`u64`	0 à 2^{64} –1 (0 à 18 446 744 073 709 551 615 ou 18 quintillions)
`u128`	0 à 2^{128} –1 (0 à environ $3,4\times10^{38}$)
`usize`	0 à 2^{32} –1 ou 2^{64} –1

Rust est signéles types entiers utilisent la représentation en complément à deux, en utilisant les mêmes modèles de bits que le type non signé correspondant pour couvrir une plage de valeurs positives et négatives (Tableau 3-4).

Tableau 3-4. Types entiers signés Rust

Taper	Intervalle
`i8`	-2^{7} à 2^{7} –1 (–128 à 127)
`i16`	-2^{15} à 2^{15} –1 (–32 768 à 32 767)
`i32`	-2^{31} à 2^{31} –1 (–2 147 483 648 à 2 147 483 647)
`i64`	-2^{63} à 2^{63} –1 (–9 223 372 036 854 775 808 à 9 223 372 036 854 775 807)
`i128`	-2^{127} à 2^{127} –1 (environ $-1,7\times10^{38}$ à $+1,7\times10^{38}$)
`isize`	Soit -2^{31} à 2^{31} –1, soit -2^{63} à 2^{63} –1

Rust utilise le `u8` type pour les valeurs d'octets. Par exemple, la lecture de données à partir d'un fichier binaire ou d'un socket génère un flux de `u8` valeurs.

Contrairement à C et C++, Rust traite les caractères comme distincts des types numériques : a `char` n'est pas un `u8`, ni un `u32` (bien qu'il fasse 32 bits). Nous décrivons le type de Rust `char` dans "Personnages".

Les types `usize` et `isize` sont analogues à `size_t` et `ptrdiff_t` en C et C++. Leur précision correspond à la taille de l'espace d'adressage sur la machine cible : elles ont une longueur de 32 bits sur les architectures 32 bits et de 64 bits sur les architectures 64 bits. Rust nécessite que les indices de tableau soient des `usize` valeurs. Les valeurs représentant les tailles de tableaux ou de vecteurs ou le nombre d'éléments dans certaines structures de données ont également généralement le `usize` type.

Littéraux entiersen Rust peuvent prendre un suffixe indiquant leur type : `42u8` est une `u8` valeur, et `1729isize` est un `isize`. Si un littéral entier n'a pas de suffixe de type, Rust reporte la détermination de son type jusqu'à ce qu'il trouve la valeur utilisée d'une manière qui la fixe : stockée dans une variable d'un type particulier, transmise à une fonction qui attend un type particulier, comparée avec une autre valeur d'un type particulier, ou quelque chose comme ça. En fin de compte, si plusieurs types peuvent fonctionner, Rust utilise par défaut `i32` si cela fait partie des possibilités. Sinon, Rust signale l'ambiguïté comme une erreur.

Les préfixes `0x`, `0o` et `0b` désignent des littéraux hexadécimaux, octaux et binaires.

Pour rendre les nombres longs plus lisibles, vous pouvez insérer des traits de soulignement entre les chiffres. Par exemple, vous pouvez écrire la plus grande `u32` valeur sous la forme `4_294_967_295`. L'emplacement exact des traits de soulignement n'est pas significatif, vous pouvez donc diviser les nombres hexadécimaux ou binaires en groupes de quatre chiffres au lieu de trois, comme dans `0xffff_ffff`, ou définir le suffixe de type à partir des chiffres, comme dans `127_u8`. Quelques exemples de littéraux entiers sont illustrés dans le Tableau 3-5.

Tableau 3-5. Exemples de littéraux entiers

Littéral	Taper	Valeur décimale
`116i8`	`i8`	116
`0xcafeu32`	`u32`	51966
`0b0010_1010`	Inféré	42
`0o106`	Inféré	70

Bien que les types numériques et le `char` type soient distincts, Rust fournit *des littéraux d'octets*, littéraux de type caractère pour les `u8` valeurs : `b'X'` représente le code ASCII du caractère `X`, sous forme de `u8` valeur. Par exemple, puisque le code ASCII pour `A` est 65, les littéraux `b'A'` et `65u8` sont exactement équivalents. Seuls les caractères ASCII peuvent apparaître dans les littéraux d'octets.

Il y a quelques caractères que vous ne pouvez pas simplement placer après le guillemet simple, car cela serait soit syntaxiquement ambigu, soit difficile à lire. Les caractères du tableau 3-6 ne peuvent être écrits qu'à l'aide d'une notation de remplacement, introduite par une barre oblique inverse.

Tableau 3-6. Caractères nécessitant une notation de remplacement

Personnage	Octet littéral	Équivalent numérique
Simple citation, `'`	`b'\''`	`39u8`
barre oblique inverse, `\`	`b'\\'`	`92u8`
Nouvelle ligne	`b'\n'`	`10u8`
Retour chariot	`b'\r'`	`13u8`
Languette	`b'\t'`	`9u8`

Pour les caractères difficiles à écrire ou à lire, vous pouvez écrire leur code en hexadécimal à la place. Un littéral d'octet de la forme `b'\xHH'`, où `HH` est un nombre hexadécimal à deux chiffres, représente l'octet dont la valeur est `HH`. Par exemple, vous pouvez écrire un littéral d'octet pour

le caractère de contrôle « échappement » ASCII sous `b'\x1b'` la forme , puisque le code ASCII pour « échappement » est 27, ou 1B en hexadécimal. Étant donné que les littéraux d'octets ne sont qu'une autre notation pour les `u8` valeurs, demandez-vous si un simple littéral numérique pourrait être plus lisible : il est probablement judicieux de l'utiliser `b'\x1b'` plutôt que simplement `27` uniquement lorsque vous souhaitez souligner que la valeur représente un code ASCII.

Vous pouvez convertir d'un type entier à un autre à l'aide de l' `as` opérateur. Nous expliquons comment fonctionnent les conversions dans "Type Casts" , mais voici quelques exemples :

```
assert_eq!(   10_i8  as u16,    10_u16); // in range
assert_eq!( 2525_u16 as i16,  2525_i16); // in range

assert_eq!(   -1_i16 as i32,    -1_i32); // sign-extended
assert_eq!(65535_u16 as i32, 65535_i32); // zero-extended

// Conversions that are out of range for the destination
// produce values that are equivalent to the original modulo 2^N,
// where N is the width of the destination in bits. This
// is sometimes called "truncation."
assert_eq!( 1000_i16 as  u8,   232_u8);
assert_eq!(65535_u32 as i16,    -1_i16);

assert_eq!(   -1_i8  as u8,    255_u8);
assert_eq!(  255_u8  as i8,     -1_i8);
```

La bibliothèque standard fournit certaines opérations sous forme de méthodes sur des nombres entiers. Par exemple:

```
assert_eq!(2_u16.pow(4), 16);            // exponentiation
assert_eq!((-4_i32).abs(), 4);           // absolute value
assert_eq!(0b101101_u8.count_ones(), 4); // population count
```

Vous pouvez les trouver dans la documentation en ligne. Notez cependant que la documentation contient des pages séparées pour le type lui-même sous " `i32` (type primitif)", et pour le module dédié à ce type (recherchez " `std::i32` ").

Dans le code réel, vous n'aurez généralement pas besoin d'écrire les suffixes de type comme nous l'avons fait ici, car le contexte déterminera le type. Quand ce n'est pas le cas, cependant, les messages d'erreur peuvent être surprenants. Par exemple, ce qui suit ne compile pas :

```
println!("{}", (-4).abs());
```

La rouille se plaint :

```
error: can't call method `abs` on ambiguous numeric type `{integer}`
```

Cela peut être un peu déconcertant : tous les types d'entiers signés ont une `abs` méthode, alors quel est le problème ? Pour des raisons techniques, Rust veut savoir exactement quel type d'entier a une valeur avant d'appeler les propres méthodes du type. La valeur par défaut de `i32` s'applique uniquement si le type est toujours ambigu après la résolution de tous les appels de méthode, il est donc trop tard pour vous aider ici. La solution consiste à préciser le type souhaité, soit avec un suffixe, soit en utilisant la fonction d'un type spécifique :

```
println!("{}", (-4_i32).abs());
println!("{}", i32::abs(-4));
```

Notez que les appels de méthode ont une priorité plus élevée que les opérateurs de préfixe unaire, soyez donc prudent lorsque vous appliquez des méthodes à des valeurs négatives. Sans les parenthèses autour `-4_i32` de la première instruction, `-4_i32.abs()` appliquerait la `abs` méthode à la valeur positive `4` , produisant positive `4` , puis nierait cela, produisant `-4` .

Arithmétique vérifiée, enveloppante, saturée et débordante

Lorsqu'une opération arithmétique entière déborde, Rust panique dans une version de débogage. Dans une version de version, l'opération *se termine*: il produit la valeur équivalente au résultat mathématiquement correct modulo la plage de la valeur. (Dans aucun des deux cas, le comportement de débordement n'est défini, comme c'est le cas en C et C++.)

Par exemple, le code suivant panique dans une version de débogage :

```
let mut i = 1;
loop {
    i *= 10; // panic: attempt to multiply with overflow
             // (but only in debug builds!)
}
```

Dans une version de version, cette multiplication revient à un nombre négatif et la boucle s'exécute indéfiniment.

Lorsque ce comportement par défaut n'est pas ce dont vous avez besoin, les types entiers fournissent des méthodes qui vous permettent d'épeler exactement ce que vous voulez. Par exemple, les paniques suivantes dans n'importe quel build :

```
let mut i:i32 = 1;
loop {
    // panic: multiplication overflowed (in any build)
    i = i.checked_mul(10).expect("multiplication overflowed");
}
```

Ces méthodes d'arithmétique entière se répartissent en quatre catégories générales :

- Opérations *vérifiées*renvoie un `Option` du résultat : `Some(v)` si le résultat mathématiquement correct peut être représenté comme une valeur de ce type, ou `None` s'il ne le peut pas. Par exemple:

```
// The sum of 10 and 20 can be represented as a u8.
assert_eq!(10_u8.checked_add(20), Some(30));

// Unfortunately, the sum of 100 and 200 cannot.
assert_eq!(100_u8.checked_add(200), None);

// Do the addition; panic if it overflows.
let sum = x.checked_add(y).unwrap();

// Oddly, signed division can overflow too, in one particular case.
// A signed n-bit type can represent -2ⁿ⁻¹, but not 2ⁿ⁻¹.
assert_eq!((-128_i8).checked_div(-1), None);
```

- Opérations d' *emballage*renvoie la valeur équivalente au résultat mathématiquement correct modulo la plage de la valeur :

```
// The first product can be represented as a u16;
// the second cannot, so we get 250000 modulo 2¹⁶.
assert_eq!(100_u16.wrapping_mul(200), 20000);
assert_eq!(500_u16.wrapping_mul(500), 53392);

// Operations on signed types may wrap to negative values.
assert_eq!(500_i16.wrapping_mul(500), -12144);
```

```
// In bitwise shift operations, the shift distance
// is wrapped to fall within the size of the value.
// So a shift of 17 bits in a 16-bit type is a shift
// of 1.
assert_eq!(5_i16.wrapping_shl(17), 10);
```

Comme expliqué, c'est ainsi que les opérateurs arithmétiques ordinaires se comportent dans les versions de version. L'avantage de ces méthodes est qu'elles se comportent de la même manière dans toutes les versions.

- Opérations *saturantes*renvoie la valeur représentable la plus proche du résultat mathématiquement correct. En d'autres termes, le résultat est "bridé" aux valeurs maximales et minimales que le type peut représenter :

```
assert_eq!(32760_i16.saturating_add(10), 32767);
assert_eq!((-32760_i16).saturating_sub(10), -32768);
```

Il n'y a pas de méthode de division saturante, de reste ou de décalage au niveau du bit.

- Opérations *débordantes*renvoie un tuple `(result, overflowed)`, où `result` est ce que la version d'encapsulation de la fonction renverrait, et indique si un débordement s'est produit `overflowed` : `bool`

```
assert_eq!(255_u8.overflowing_sub(2), (253, false));
assert_eq!(255_u8.overflowing_add(2), (1, true));
```

`overflowing_shl` et `overflowing_shr` s'écartent un peu du motif : ils ne renvoient true `overflowed` que si la distance de décalage était aussi grande ou supérieure à la largeur en bits du type lui-même. Le décalage réel appliqué est le décalage demandé modulo la largeur de bit du type :

```
// A shift of 17 bits is too large for `u16`, and 17 modulo 16 is 1.
assert_eq!(5_u16.overflowing_shl(17), (10, true));
```

Les noms d'opération qui suivent le préfixe , , ou sont indiqués `checked_` dans `wrapping_` le `saturating_` Tableau 3-7 `.overflowing_`

Tableau 3-7. Noms d'opération

Opération	Suffixe de nom	Exemple
Ajout	`add`	`100_i8.checked_add(27) == Some(127)`
Soustraction	`sub`	`10_u8.checked_sub(11) == None`
Multiplication	`mul`	`128_u8.saturating_mul(3) == 255`
Division	`div`	`64_u16.wrapping_div(8) == 8`
Reste	`rem`	`(-32768_i16).wrapping_rem(-1) == 0`
Négation	`neg`	`(-128_i8).checked_neg() == None`
Valeur absolue	`abs`	`(-32768_i16).wrapping_abs() == -32768`
Exponentation	`pow`	`3_u8.checked_pow(4) == Some(81)`
Décalage à gauche au niveau du bit	`shl`	`10_u32.wrapping_shl(34) == 40`
Décalage à droite au niveau du bit	`shr`	`40_u64.wrapping_shr(66) == 10`

Types à virgule flottante

Rouillerfournit des types à virgule flottante simple et double précision IEEE. Ces types incluent des infinis positifs et négatifs, des valeurs zéro positives et négatives distinctes et une valeur *non numérique* (Tableau 3-8).

Tableau 3-8. Types à virgule flottante simple et double précision IEEE

Taper	Précision	Intervalle
`f32`	Simple précision IEEE (au moins 6 chiffres décimaux)	Environ $-3{,}4 \times 10^{38}$ à $+3{,}4 \times 10^{38}$
`f64`	Double précision IEEE (au moins 15 chiffres décimaux)	Environ $-1{,}8 \times 10^{308}$ à $+1{,}8 \times 10^{308}$

Rust `f32` et `f64` correspondent aux types `float` et `double` en C et C++ (dans les implémentations prenant en charge la virgule flottante IEEE) ainsi qu'en Java (qui utilise toujours la virgule flottante IEEE).

Littéraux à virgule flottanteavoir la forme générale schématisée à la Figure 3-1 .

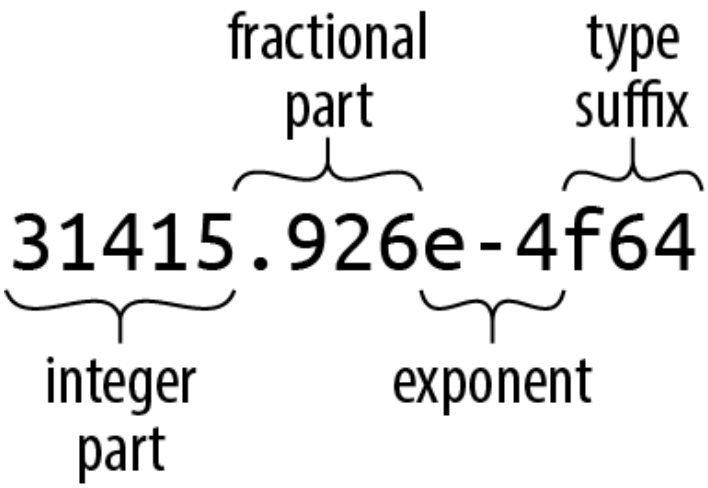

Illustration 3-1. Un littéral à virgule flottante

Chaque partie d'un nombre à virgule flottante après la partie entière est facultative, mais au moins une partie fractionnaire, un exposant ou un suffixe de type doit être présent, pour le distinguer d'un littéral entier. La partie fractionnaire peut être constituée d'un seul point décimal, c'est donc `5.` une constante à virgule flottante valide.

Si un littéral à virgule flottante n'a pas de suffixe de type, Rust vérifie le contexte pour voir comment les valeurs sont utilisées, tout comme il le fait pour les littéraux entiers. S'il trouve finalement que l'un ou l'autre des types à virgule flottante peut convenir, il choisit `f64` par défaut.

Aux fins de l'inférence de type, Rust traite les littéraux entierset les littéraux à virgule flottante en tant que classes distinctes : il n'inférera jamais un type à virgule flottante pour un littéral entier, ou vice versa. Le tableau 3-9 montre quelques exemples de littéraux à virgule flottante.

Tableau 3-9. Exemples de littéraux à virgule flottante

Littéral	Taper	Valeur mathématique
`-1.5625`	Inféré	$-(1\ ^{9}/_{16})$
`2.`	Inféré	2
`0.25`	Inféré	¼
`1e4`	Inféré	10 000
`40f32`	`f32`	40
`9.109_383_56e-31f64`	`f64`	Environ $9{,}10938356 \times 10^{-31}$

Les types `f32` et `f64` ont des constantes associées pour les valeurs spéciales requises par l'IEEE telles que `INFINITY`, `NEG_INFINITY` (infini négatif), `NAN` (la valeur non numérique) et `MIN` et `MAX` (les valeurs finies les plus grandes et les plus petites) :

```
assert!((-1. / f32:: INFINITY).is_sign_negative());
assert_eq!(-f32:: MIN, f32::MAX);
```

Les types `f32` et `f64` fournissent un complément complet de méthodes pour les calculs mathématiques ; par exemple, `2f64.sqrt()` est la racine carrée double précision de deux. Quelques exemples:

```
assert_eq!(5f32.sqrt() * 5f32.sqrt(), 5.); // exactly 5.0, per IEEE
assert_eq!((-1.01f64).floor(), -2.0);
```

Encore une fois, les appels de méthode ont une priorité plus élevée que les opérateurs de préfixe, alors assurez-vous de bien mettre entre parenthèses les appels de méthode sur les valeurs négatives.

Les modules `std::f32::consts` et `std::f64::consts` fournissent diverses constantes mathématiques couramment utilisées telles que `E`, `PI` et la racine carrée de deux.

Lors de la recherche dans la documentation, n'oubliez pas qu'il existe des pages pour les types eux-mêmes, nommés " `f32` (type primitif)" et " `f64` (type primitif)", et les modules pour chaque type, `std::f32` et `std::f64`.

Comme pour les entiers, vous n'aurez généralement pas besoin d'écrire des suffixes de type sur des littéraux à virgule flottante dans du code réel, mais lorsque vous le ferez, mettre un type sur le littéral ou sur la fonction suffira :

```
println!("{}", (2.0_f64).sqrt());
println!("{}", f64::sqrt(2.0));
```

Contrairement à C et C++, Rust n'effectue presque aucune conversion numérique implicite. Si une fonction attend un `f64` argument, c'est une erreur de passer une `i32` valeur comme argument. En fait, Rust ne convertira même pas implicitement une `i16` valeur en une `i32` valeur, même si chaque `i16` valeur est également une `i32` valeur. Mais vous pouvez toujours écrire des conversions *explicites* `as` en utilisant l' opérateur : `i as f64` , ou `x as i32` .

L'absence de conversions implicites rend parfois une expression Rust plus détaillée que ne le serait le code C ou C++ analogue. Cependant, les conversions implicites d'entiers ont un historique bien établi de bogues et de failles de sécurité, en particulier lorsque les entiers en question représentent la taille de quelque chose en mémoire et qu'un débordement imprévu se produit. D'après notre expérience, le fait d'écrire des conversions numériques dans Rust nous a alertés sur des problèmes que nous aurions autrement manqués.

Nous expliquons exactement comment les conversionsse comporter dans "Type Casts" .

Le type booléen

Booléen de Rusttype, `bool` , a les deux valeurs habituelles pour de tels types, `true` et `false` . Opérateurs de comparaisonaime `==` et `<` produit des `bool` résultats : la valeur de `2 < 5` est `true` .

De nombreux langages sont indulgents quant à l'utilisation de valeurs d'autres types dans des contextes qui nécessitent une valeur booléenne : C et C++ convertissent implicitement les caractères, les entiers, les nombres à virgule flottante et les pointeurs en valeurs booléennes, afin qu'ils puissent être utilisés directement comme condition dans un `if` ou `while` déclaration. Python autorise les chaînes, les listes, les dictionnaires et même les ensembles dans des contextes booléens, traitant ces valeurs comme vraies si elles ne sont pas vides. Rust, cependant, est très

strict : les structures de contrôle comme `if` et `while` exigent que leurs conditions soient `bool` des expressions, tout comme les opérateurs logiques de court-circuit `&&` et `||`. Vous devez écrire `if x != 0 { ... }`, pas simplement `if x { ... }`.

`as` L' opérateur de Rust peut convertir `bool` des valeurs en types entiers :

```
assert_eq!(false as i32, 0);
assert_eq!(true  as i32, 1);
```

Cependant, `as` ne convertira pas dans l'autre sens, des types numériques vers `bool`. Au lieu de cela, vous devez écrire une comparaison explicite telle que `x != 0`.

Bien que a n'ait `bool` besoin que d'un seul bit pour le représenter, Rust utilise un octet entier pour une `bool` valeur en mémoire, vous pouvez donc créer un pointeur vers celui-ci.

Personnages

Le personnage de Rusttype `char` représente un seul caractère Unicode, sous la forme d'une valeur 32 bits.

Rust utilise le `char` type pour les caractères uniques de manière isolée, mais utilise l'encodage UTF-8 pour les chaînes et les flux de texte. Ainsi, a `String` représente son texte comme une séquence d'octets UTF-8, pas comme un tableau de caractères.

Personnageles littéraux sont des caractères entre guillemets simples, comme `'8'` ou `'!'`. Vous pouvez utiliser toute l'étendue d'Unicode : `'錆'` est un `char` littéral représentant le kanji japonais pour *sabi* (rouille).

Comme pour les littéraux d'octets, des échappements par barre oblique inverse sont requis pour quelques caractères (Tableau 3-10).

Tableau 3-10. Caractères nécessitant des échappements par barre oblique inverse

Personnage	Littéral de caractère de rouille
Simple citation, `'`	`'\''`
barre oblique inverse, `\`	`'\\'`
Nouvelle ligne	`'\n'`
Retour chariot	`'\r'`
Languette	`'\t'`

Si vous préférez, vous pouvez écrire le point de code Unicode d'un caractère en hexadécimal:

- Si le point de code du caractère est compris entre U+0000 et U+007F (c'est-à-dire s'il provient du jeu de caractères ASCII), vous pouvez écrire le caractère sous la forme `'\xHH'`, où `HH` est un nombre hexadécimal à deux chiffres. Par exemple, les caractères littéraux `'*'` et `'\x2A'` sont équivalents, car le point de code du caractère `*` est 42, ou 2A en hexadécimal.
- Vous pouvez écrire n'importe quel caractère Unicode sous la forme `'\u{HHHHHH}'`, où `HHHHHH` est un nombre hexadécimal jusqu'à six chiffres, avec des traits de soulignement autorisés pour le regroupement comme d'habitude. Par exemple, le caractère littéral `'\u{CA0}'` représente le caractère "ಠ", un caractère Kannada utilisé dans l'Unicode Look of Disapproval, "ಠ_ಠ". Le même littéral pourrait aussi être simplement écrit comme `'ಠ'`.

A `char` contient toujours un point de code Unicode dans la plage 0x0000 à 0xD7FF, ou 0xE000 à 0x10FFFF. A `char` n'est jamais une moitié de paire de substitution (c'est-à-dire un point de code compris entre 0xD800 et 0xDFFF) ou une valeur en dehors de l'espace de code Unicode (c'est-à-dire supérieure à 0x10FFFF). Rust utilise le système de type et les vérifications dynamiques pour s'assurer que `char` les valeurs sont toujours dans la plage autorisée.

Rust ne convertit jamais implicitement entre `char` et tout autre type. Vous pouvez utiliser l' `as` opérateur de conversion pour convertir a `char` en un type entier ; pour les types inférieurs à 32 bits, les bits supérieurs de la valeur du caractère sont tronqués :

```
assert_eq!('*' as i32, 42);
assert_eq!('ಠ' as u16, 0xca0);
assert_eq!('ಠ' as i8, -0x60); // U+0CA0 truncated to eight bits, signed
```

Dans l'autre sens, `u8` est le seul type `as` vers lequel l'opérateur convertira `char` : Rust a l'intention que l' `as` opérateur n'effectue que des conversions bon marché et infaillibles, mais chaque type d'entier autre que `u8` comprend des valeurs qui ne sont pas autorisées. points de code Unicode, donc ces conversions nécessiteraient run -vérifications horaires. Au lieu de cela, la fonction de bibliothèque standard `std::char::from_u32` prend n'importe quelle `u32` valeur et renvoie un `Option<char>` : si `u32` n'est pas un point de code Unicode autorisé, alors `from_u32` renvoie `None` ; sinon, elle renvoie `Some(c)` , où `c` est le `char` résultat.

La bibliothèque standard fournit des méthodes utiles sur les caractères, que vous pouvez rechercher dans la documentation en ligne sous « `char` (type primitif) » et le module « » `std::char` . Par exemple:

```
assert_eq!('*'.is_alphabetic(), false);
assert_eq!('β'.is_alphabetic(), true);
assert_eq!('8'.to_digit(10), Some(8));
assert_eq!('ಠ'.len_utf8(), 3);
assert_eq!(std:: char::from_digit(2, 10), Some('2'));
```

Naturellement, les caractères isolés ne sont pas aussi intéressants que les chaînes et les flux de texte. Nous décrirons le `String` type standard de Rust et la gestion du texteen général dans "Types de chaînes" .

Tuples

Un *tuple*est une paire, ou triple, quadruple, quintuple, etc. (donc, *n-tuple* , ou *tuple*), de valeurs de types assortis. Vous pouvez écrire un tuple comme une séquence d'éléments, séparés par des virgules et entourés de parenthèses. Par exemple, `("Brazil", 1985)` est un tuple dont le premier élément est une chaîne allouée statiquement et dont le second est un entier ; son type est `(&str, i32)` . Étant donné une valeur de tuple `t` , vous pouvez accéder à ses éléments en tant que `t.0` , `t.1` , etc.

Dans une certaine mesure, les tuples ressemblent à des tableaux: les deux types représentent une séquence ordonnée de valeurs. De nombreux langages de programmation confondent ou combinent les deux concepts,

mais dans Rust, ils sont complètement séparés. D'une part, chaque élément d'un tuple peut avoir un type différent, alors que les éléments d'un tableau doivent tous être du même type. De plus, les tuples n'autorisent que les constantes comme indices, comme `t.4`. Vous ne pouvez pas écrire `t.i` ou `t[i]` pour obtenir le `i` ème élément.

Le code Rust utilise souvent des types tuple pour renvoyer plusieurs valeurs à partir d'une fonction. Par exemple, la `split_at` méthode sur les tranches de chaîne, qui divise une chaîne en deux moitiés et les renvoie toutes les deux, est déclarée comme ceci :

```
fn split_at(&self, mid: usize) ->(&str, &str);
```

Le type de retour `(&str, &str)` est un tuple de deux tranches de chaîne. Vous pouvez utiliser la syntaxe de correspondance de modèle pour affecter chaque élément de la valeur de retour à une variable différente :

```
let text = "I see the eigenvalue in thine eye";
let (head, tail) = text.split_at(21);
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

C'est plus lisible que l'équivalent :

```
let text = "I see the eigenvalue in thine eye";
let temp = text.split_at(21);
let head = temp.0;
let tail = temp.1;
assert_eq!(head, "I see the eigenvalue ");
assert_eq!(tail, "in thine eye");
```

Vous verrez également des tuples utilisés comme une sorte de type de structure à drame minimal. Par exemple, dans le programme Mandelbrot du chapitre 2 , nous devions transmettre la largeur et la hauteur de l'image aux fonctions qui la tracent et l'écrivent sur le disque. Nous pourrions déclarer une structure avec `width` et `height` membres, mais c'est une notation assez lourde pour quelque chose d'aussi évident, donc nous avons juste utilisé un tuple :

```
/// Write the buffer `pixels`, whose dimensions are given by `bounds`, to
/// file named `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
```

```
    -> Result<(), std:: io::Error>
{ ... }
```

Le type du `bounds` paramètre est `(usize, usize)`, un tuple de deux `usize` valeurs. Certes, nous pourrions tout aussi bien écrire des paramètres `width` et séparés `height`, et le code machine serait à peu près le même dans les deux cas. C'est une question de clarté. Nous considérons la taille comme une valeur, pas deux, et l'utilisation d'un tuple nous permet d'écrire ce que nous voulons dire.

L'autre type de tuple couramment utiliséest le zéro-tuple `()`. Ceci est traditionnellement appelé le *type d'unité*car il n'a qu'une seule valeur, également écrite `()`. Rust utilise le type d'unité où il n'y a pas de valeur significative à transporter, mais le contexte nécessite néanmoins une sorte de type.

Par exemple, une fonction qui ne renvoie aucune valeur a un type de retour de `()`. La `std::mem::swap` fonction de la bibliothèque standard n'a pas de valeur de retour significative ; il échange juste les valeurs de ses deux arguments. La déclaration pour `std::mem::swap` se lit comme suit :

```
fn swap<T>(x: &mut T, y:&mut T);
```

Les `<T>` moyensc'est *générique* `swap _`: vous pouvez l'utiliser sur des références à des valeurs de n'importe quel type `T`. Mais la signature omet `swap` complètement le type de retour de , qui est un raccourci pour renvoyer le type d'unité :

```
fn swap<T>(x: &mut T, y: &mut T) ->();
```

De même, l' `write_image` exemple que nous avons mentionné précédemment a un type de retour de `Result<(), std::io::Error>`, ce qui signifie que la fonction renvoie une `std::io::Error` valeur si quelque chose ne va pas, mais ne renvoie aucune valeur en cas de succès.

Si vous le souhaitez, vous pouvez inclure une virgule après le dernier élément d'un tuple : les types `(&str, i32,)` et `(&str, i32)` sont équivalents, tout comme les expressions `("Brazil", 1985,)` et `("Brazil", 1985)`. Rust autorise systématiquement une virgule de fin supplémentaire partout où des virgules sont utilisées : arguments de fonction, tableaux, définitions de structure et d'énumération, etc. Cela peut sembler

étrange aux lecteurs humains, mais cela peut faciliter la lecture des différences lorsque des entrées sont ajoutées et supprimées à la fin d'une liste.

Par souci de cohérence, il existe même des tuples qui contiennent une seule valeur. Le littéral `("lonely hearts",)` est un tuple contenant une seule chaîne ; son type est `(&str,)`. Ici, la virgule après la valeur est nécessaire pour distinguer le tuple singleton d'une simple expression entre parenthèses.

Types de pointeur

Rust a plusieurs types qui représentent des adresses mémoire.

c'est une grande différenceentre Rust et la plupart des langages avec ramasse-miettes. En Java, si `class Rectangle` contient un champ `Vector2D upperLeft;`, alors `upperLeft` est une référence à un autre `Vector2D` objet créé séparément. Les objets ne contiennent jamais physiquement d'autres objets en Java.

La rouille est différente. Le langage est conçu pour aider à maintenir les allocations au minimum. Les valeurs s'imbriquent par défaut. La valeur `((0, 0), (1440, 900))` est stockée sous la forme de quatre nombres entiers adjacents. Si vous le stockez dans une variable locale, vous avez une variable locale large de quatre entiers. Rien n'est alloué dans le tas.

C'est excellent pour l'efficacité de la mémoire, mais par conséquent, lorsqu'un programme Rust a besoin de valeurs pour pointer vers d'autres valeurs, il doit utiliser explicitement les types de pointeur. La bonne nouvelle est que les types de pointeurs utilisés dans Rust sécurisé sont contraints d'éliminer les comportements indéfinis, de sorte que les pointeurs sont beaucoup plus faciles à utiliser correctement dans Rust qu'en C++.

Nous aborderons ici trois types de pointeurs : les références, les boîtes et les pointeurs non sécurisés.

Références

Une valeurde type `&String` (prononcé "ref String") est une référence à une `String` valeur, a `&i32` est une référence à un `i32`, et ainsi de suite.

Il est plus facile de commencer en considérant les références comme le type de pointeur de base de Rust. Au moment de l'exécution, une réfé-

rence à un `i32` est un seul mot machine contenant l'adresse du `i32`, qui peut être sur la pile ou dans le tas. L'expression `&x` produit une référence à `x`; dans la terminologie Rust, nous disons qu'il *emprunte une référence à `x`*. Étant donné une référence `r`, l'expression `*r` fait référence à la valeur `r` vers laquelle pointe. Ils ressemblent beaucoup aux opérateurs `&` et en C et C++. `*` Et comme un pointeur C, une référence ne libère pas automatiquement de ressources lorsqu'elle sort de la portée.

Contrairement aux pointeurs C, cependant, les références Rust ne sont jamais nulles : il n'y a tout simplement aucun moyen de produire une valeur nulle.référence en toute sécurité Rust. Et contrairement à C, Rust suit la propriété et la durée de vie des valeurs, de sorte que les erreurs telles que les pointeurs pendants, les doubles libérations et l'invalidation des pointeurs sont exclues au moment de la compilation.

Les références de rouille se déclinent en deux saveurs :

`&T`

: Un immuable, référence partagée. Vous pouvez avoir plusieurs références partagées à une valeur donnée à la fois, mais elles sont en lecture seule : la modification de la valeur vers laquelle elles pointent est interdite, comme `const T*` en C.

`&mut T`

: Une mutable, référence exclusive. Vous pouvez lire et modifier la valeur vers laquelle elle pointe, comme avec a `T*` en C. Mais tant que la référence existe, vous ne pouvez avoir aucune autre référence d'aucune sorte à cette valeur. En fait, la seule façon d'accéder à la valeur est de passer par la référence mutable.

Rust utilise cette dichotomie entre les références partagées et mutables pour appliquer une règle « auteur unique *ou* lecteurs multiples » : soit vous pouvez lire et écrire la valeur, soit elle peut être partagée par n'importe quel nombre de lecteurs, mais jamais les deux en même temps. Cette séparation, renforcée par des vérifications au moment de la compilation, est au cœur des garanties de sécurité de Rust. Le chapitre 5 explique les règles de Rust pour une utilisation sûre des références.

Des boites

Le plus simplefaçon d'allouer une valeur dans le tas est d'utiliser `Box::new` :

```
let t = (12, "eggs");
let b = Box::new(t);  // allocate a tuple in the heap
```

Le type de `t` est `(i32, &str)`, donc le type de `b` est `Box<(i32, &str)>`. L'appel à `Box::new` alloue suffisamment de mémoire pour contenir le tuple sur le tas. Lorsqu'il `b` sort de la portée, la mémoire est immédiatement libérée, sauf si `b` elle a été *déplacée*, en la retournant, par exemple. Les mouvements sont essentiels à la façon dont Rust gère les valeurs allouées par tas ; nous expliquons tout cela en détail au chapitre 4.

Pointeurs bruts

Rouillera également les types de pointeurs bruts `*mut T` et `*const T`. Les pointeurs bruts sont vraiment comme les pointeurs en C++. L'utilisation d'un pointeur brut n'est pas sûre, car Rust ne fait aucun effort pour suivre ce vers quoi il pointe. Par exemple, les pointeurs bruts peuvent être nuls, ou ils peuvent pointer vers de la mémoire qui a été libérée ou qui contient maintenant une valeur d'un type différent. Toutes les erreurs de pointage classiques du C++ sont proposées pour votre plus grand plaisir.

Cependant, vous ne pouvez que déréférencerpointeurs bruts dans un `unsafe` bloc. Un `unsafe` bloc est le mécanisme d'acceptation de Rust pour les fonctionnalités de langage avancées dont la sécurité dépend de vous. Si votre code n'a pas de `unsafe` blocs (ou si ceux qu'il contient sont écrits correctement), alors les garanties de sécurité sur lesquelles nous insistons tout au long de ce livre sont toujours valables. Pour plus de détails, reportez- vous au chapitre 22.

Tableaux, vecteurs et tranches

Rust a trois types pour représenter une séquence de valeurs en mémoire:

- Le type `[T; N]` représente un tableau de `N` valeurs, chacune de type `T`. La taille d'un tableau est une constante déterminée au moment de la compilation et fait partie du type ; vous ne pouvez pas ajouter de nouveaux éléments ou réduire un tableau.
- Le type `Vec<T>`, appelé *vecteurde `T`s*, est une séquence de valeurs de type évolutive allouée dynamiquement `T`. Les éléments d'un vecteur vivent sur le tas, vous pouvez donc redimensionner les vecteurs à vo-

lonté : mettez-y de nouveaux éléments, ajoutez-y d'autres vecteurs, supprimez des éléments, etc.

- Les types `&[T]` et `&mut [T]`, appelés *partagetranche de `T`s* et *mutabletranche de `T`s*, sont des références à une série d'éléments qui font partie d'une autre valeur, comme un tableau ou un vecteur. Vous pouvez considérer une tranche comme un pointeur vers son premier élément, avec un décompte du nombre d'éléments auxquels vous pouvez accéder à partir de ce point. Une tranche modifiable `&mut [T]` vous permet de lire et de modifier des éléments, mais ne peut pas être partagée ; une tranche partagée `&[T]` permet de partager l'accès entre plusieurs lecteurs, mais ne permet pas de modifier les éléments.

Étant donné une valeur `v` de l'un de ces trois types, l'expression `v.len()` donne le nombre d'éléments dans `v` et `v[i]` fait référence au `i` ème élément de `v`. Le premier élément est `v[0]`, et le dernier élément est `v[v.len() - 1]`. Contrôles de rouille qui `i` se situent toujours dans cette plage ; si ce n'est pas le cas, l'expression panique. La longueur de `v` peut être zéro, auquel cas toute tentative d'indexation paniquera. `i` doit être une `usize` valeur ; vous ne pouvez pas utiliser d'autre type entier comme index.

Tableaux

Il existe plusieurs façons d'écrire un tableauvaleurs. Le plus simple est d'écrire une suite de valeurs entre crochets :

```
let lazy_caterer:[u32; 6] = [1, 2, 4, 7, 11, 16];
let taxonomy = ["Animalia", "Arthropoda", "Insecta"];

assert_eq!(lazy_caterer[3], 7);
assert_eq!(taxonomy.len(), 3);
```

Pour le cas courant d'un long tableau rempli d'une certaine valeur, vous pouvez écrire , où est la valeur que chaque élément doit avoir et est la longueur. Par exemple, est un tableau de 10 000 éléments, tous définis sur : `[V; N] V N [true; 10000] bool true`

```
let mut sieve = [true; 10000];
for i in 2..100 {
    if sieve[i] {
        let mut j = i * i;
        while j < 10000 {
            sieve[j] = false;
            j += i;
```

```
            }
        }
    }

    assert!(sieve[211]);
    assert!(!sieve[9876]);
```

Vous verrez cette syntaxe utilisée pour les tampons de taille fixe : `[0u8; 1024]` peut être un tampon d'un kilo-octet, rempli de zéros. Rust n'a pas de notation pour un tableau non initialisé. (En général, Rust garantit que le code ne peut jamais accéder à une sorte de valeur non initialisée.)

La longueur d'un tableau fait partie de son type et est fixée au moment de la compilation. Si `n` est une variable, vous ne pouvez pas écrire `[true; n]` pour obtenir un tableau d' `n` éléments. Lorsque vous avez besoin d'un tableau dont la longueur varie au moment de l'exécution (et vous le faites généralement), utilisez plutôt un vecteur.

Les méthodes utiles que vous aimeriez voir sur les tableaux (itération sur les éléments, recherche, tri, remplissage, filtrage, etc.) sont toutes fournies en tant que méthodes sur les tranches, et non sur les tableaux. Mais Rust convertit implicitement une référence à un tableau en une tranche lors de la recherche de méthodes, vous pouvez donc appeler directement n'importe quelle méthode de tranche sur un tableau :

```
let mut chaos = [3, 5, 4, 1, 2];
chaos.sort();
assert_eq!(chaos, [1, 2, 3, 4, 5]);
```

Ici, la `sort` méthode est en fait définie sur des tranches, mais puisqu'elle prend son opérande par référence, Rust produit implicitement une `&mut [i32]` tranche faisant référence à l'ensemble du tableau et la transmet à `sort` pour opérer dessus. En fait, la `len` méthode que nous avons mentionnée précédemment est également une méthode de tranche. Nous couvrons les tranches plus en détail dans "Slices" .

Vecteurs

Un vecteur `Vec<T>` est un tableau redimensionnable d'éléments de type `T` , alloués sur le tas.

Il existe plusieurs façons de créer des vecteurs. Le plus simple est d'utiliser la `vec!` macro, ce qui nous donne une syntaxe pour les vecteurs qui

ressemble beaucoup à un littéral de tableau :

```
let mut primes = vec![2, 3, 5, 7];
assert_eq!(primes.iter().product::<i32>(), 210);
```

Mais bien sûr, il s'agit d'un vecteur, pas d'un tableau, nous pouvons donc y ajouter des éléments dynamiquement :

```
primes.push(11);
primes.push(13);
assert_eq!(primes.iter().product::<i32>(), 30030);
```

Vous pouvez également créer un vecteur en répétant une valeur donnée un certain nombre de fois, toujours en utilisant une syntaxe qui imite les littéraux de tableau :

```
fn new_pixel_buffer(rows: usize, cols: usize) ->Vec<u8> {
    vec![0; rows * cols]
}
```

La `vec!` macro équivaut à appeler `Vec::new` pour créer un nouveau vecteur vide, puis à pousser les éléments dessus, ce qui est un autre idiome :

```
let mut pal = Vec::new();
pal.push("step");
pal.push("on");
pal.push("no");
pal.push("pets");
assert_eq!(pal, vec!["step", "on", "no", "pets"]);
```

Une autre possibilité est de construire un vecteur à partir des valeurs produites par un itérateur :

```
let v:Vec<i32> = (0..5).collect();
assert_eq!(v, [0, 1, 2, 3, 4]);
```

Vous aurez souvent besoin de fournir le type lors de l'utilisation `collect` (comme nous l'avons fait ici), car il peut créer de nombreux types de collections, pas seulement des vecteurs. En spécifiant le type de `v`, nous avons précisé le type de collection que nous voulons.

Comme pour les tableaux, vous pouvez utiliser les méthodes slice sur les vecteurs :

```
// A palindrome!
let mut palindrome = vec!["a man", "a plan", "a canal", "panama"];
palindrome.reverse();
// Reasonable yet disappointing:
assert_eq!(palindrome, vec!["panama", "a canal", "a plan", "a man"]);
```

Ici, la `reverse` méthodeest en fait défini sur les tranches, mais l'appel emprunte implicitement une `&mut [&str]` tranche au vecteur et l'invoque `reverse` sur celle-ci.

`Vec` est un type essentiel à Rust - il est utilisé presque partout où l'on a besoin d'une liste de taille dynamique - il existe donc de nombreuses autres méthodes qui construisent de nouveaux vecteurs ou étendent ceux qui existent déjà. Nous les aborderons au chapitre 16 .

A `Vec<T>` se compose de trois valeurs: un pointeur vers le tampon alloué par tas pour les éléments, qui est créé et détenu par le `Vec<T>` ; le nombre d'éléments que la mémoire tampon a la capacité de stocker ; et le nombre qu'il contient actuellement (en d'autres termes, sa longueur). Lorsque le tampon a atteint sa capacité, ajouter un autre élément au vecteur implique d'allouer un tampon plus grand, d'y copier le contenu actuel, de mettre à jour le pointeur du vecteur et sa capacité à décrire le nouveau tampon, et enfin de libérer l'ancien.

Si vous connaissez le nombre d'éléments dont un vecteur aura besoin à l'avance, au lieu de `Vec::new` vous pouvez appeler `Vec::with_capacity` pour créerun vecteur avec un tampon suffisamment grand pour les contenir tous, dès le début ; ensuite, vous pouvez ajouter les éléments au vecteur un par un sans provoquer de réallocation. La `vec!` macro utilise une astuce comme celle-ci, car elle sait combien d'éléments le vecteur final aura. Notez que cela n'établit que la taille initiale du vecteur ; si vous dépassez votre estimation, le vecteur agrandit simplement son stockage comme d'habitude.

De nombreuses fonctions de bibliothèque recherchent la possibilité d'utiliser à la `Vec::with_capacity` place de `Vec::new` . Par exemple, dans l' `collect` exemple, l'itérateur `0..5` sait à l'avance qu'il donnera cinq valeurs, et la `collect` fonction en profite pour pré-allouer le vecteur qu'il renvoie avec la capacité correcte. Nous verrons comment cela fonctionne au chapitre 15 .

`len` Tout comme la méthode d'un vecteurrenvoie le nombre d'éléments qu'il contient maintenant, sa `capacity` méthode renvoie le nombre d'éléments qu'il pourrait contenir sans réallocation :

```
let mut v = Vec::with_capacity(2);
assert_eq!(v.len(), 0);
assert_eq!(v.capacity(), 2);

v.push(1);
v.push(2);
assert_eq!(v.len(), 2);
assert_eq!(v.capacity(), 2);

v.push(3);
assert_eq!(v.len(), 3);
// Typically prints "capacity is now 4":
println!("capacity is now {}", v.capacity());
```

La capacité imprimée à la fin n'est pas garantie d'être exactement 4, mais elle sera d'au moins 3, puisque le vecteur contient trois valeurs.

Vous pouvez insérer et supprimer des éléments où vous le souhaitez dans un vecteur, bien que ces opérations déplacent tous les éléments après la position affectée vers l'avant ou vers l'arrière, elles peuvent donc être lentes si le vecteur est long :

```
let mut v = vec![10, 20, 30, 40, 50];

// Make the element at index 3 be 35.
v.insert(3, 35);
assert_eq!(v, [10, 20, 30, 35, 40, 50]);

// Remove the element at index 1.
v.remove(1);
assert_eq!(v, [10, 30, 35, 40, 50]);
```

Vous pouvez utiliser la `pop` méthode pour supprimer le dernier élément et le renvoyer. Plus précisément, extraire une valeur de a `Vec<T>` renvoie un `Option<T>` : `None` si le vecteur était déjà vide, ou `Some(v)` si son dernier élément avait été `v` :

```
let mut v = vec!["Snow Puff", "Glass Gem"];
assert_eq!(v.pop(), Some("Glass Gem"));
assert_eq!(v.pop(), Some("Snow Puff"));
assert_eq!(v.pop(), None);
```

Vous pouvez utiliser une `for` boucle pour parcourir un vecteur :

```
// Get our command-line arguments as a vector of Strings.
let languages: Vec<String> = std:: env::args().skip(1).collect();
for l in languages {
    println!("{}: {}", l,
             if l.len() % 2 == 0 {
                 "functional"
             } else {
                 "imperative"
             });
}
```

L'exécution de ce programme avec une liste de langages de programmation est éclairante :

```
$cargo run Lisp Schéma C C++ Fortran
   Compiling proglangs v0.1.0 (/home/jimb/rust/proglangs)
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
     Running `target/debug/proglangs Lisp Scheme C C++ Fortran`
Lisp: functional
Scheme: functional
C: imperative
C++: imperative
Fortran: imperative
$
```

Enfin, une définition satisfaisante du terme *langage fonctionnel*.

Malgré son rôle fondamental, `Vec` est un type ordinaire défini dans Rust, non intégré au langage. Nous couvrirons les techniques nécessaires pour implémenter de tels typesau chapitre 22 .

Tranches

Une tranche, écrit `[T]` sans spécifier la longueur, est une région d'un tableauou vecteur. Étant donné qu'une tranche peut avoir n'importe quelle longueur, les tranches ne peuvent pas être stockées directement dans des variables ou transmises en tant qu'arguments de fonction. Les tranches sont toujours passées par référence.

Une référence à une tranche est un *pointeur gras*: une valeur de deux mots comprenant un pointeur vers le premier élément de la tranche et le nombre d'éléments dans la tranche.

Supposons que vous exécutiez le code suivant :

```
let v: Vec<f64> = vec![0.0,  0.707,  1.0,  0.707];
let a:[f64; 4] =      [0.0, -0.707, -1.0, -0.707];

let sv: &[f64] = &v;
let sa:&[f64] = &a;
```

Dans les deux dernières lignes, Rust convertit automatiquement la `&Vec<f64>` référence et la `&[f64; 4]` référence en références de tranche qui pointent directement vers les données.

À la fin, la mémoire ressemble à la figure 3-2 .

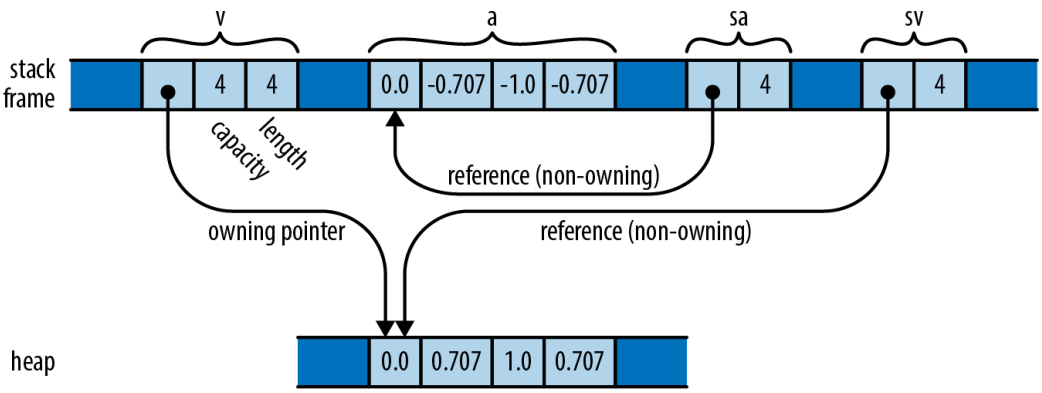

Illustration 3-2. Un vecteur `v` et un tableau `a` en mémoire, avec des tranches `sa` et `sv` se référant à chacun

Alors qu'une référence ordinaire est un pointeur non propriétaire vers une valeur unique, une référence à une tranche est un pointeur non propriétaire vers une plage de valeurs consécutives en mémoire. Cela fait des références de tranche un bon choix lorsque vous souhaitez écrire une fonction qui opère sur un tableau ou un vecteur. Par exemple, voici une fonction qui imprime une tranche de nombres, un par ligne :

```
fn print(n:&[f64]) {
    for elt in n {
        println!("{}", elt);
    }
}

print(&a);  // works on arrays
print(&v);  // works on vectors
```

Étant donné que cette fonction prend une référence de tranche comme argument, vous pouvez l'appliquer à un vecteur ou à un tableau, comme indiqué. En fait, de nombreuses méthodes que vous pourriez considérer

comme appartenant à des vecteurs ou à des tableaux sont des méthodes définies sur des tranches : par exemple, les méthodes `sort` et `reverse`, qui trient ou inversent une séquence d'éléments en place, sont en fait des méthodes sur le type de tranche `[T]`.

Vous pouvez obtenir une référence à une tranche d'un tableau ou d'un vecteur, ou à une tranche d'une tranche existante, en l'indexant avec une plage :

```
print(&v[0..2]);    // print the first two elements of v
print(&a[2..]);     // print elements of a starting with a[2]
print(&sv[1..3]);   // print v[1] and v[2]
```

Comme pour les accès aux tableaux ordinaires, Rust vérifie que les indices sont valides. Essayer d'emprunter une tranche qui s'étend au-delà de la fin des données entraîne une panique.

Étant donné que les tranches apparaissent presque toujours derrière les références, nous nous référons souvent à des types comme `&[T]` ou `&str` en tant que "tranches", en utilisant le nom le plus court pour le concept le plus courant..

Types de chaînes

Programmeursfamilier avec C++ se souviendra qu'il existe deux types de chaînes dans le langage. Les littéraux de chaîne ont le type de pointeur `const char *`. La bibliothèque standard propose également une classe, `std::string`, pour créer dynamiquement des chaînes au moment de l'exécution.

Rust a un design similaire. Dans cette section, nous montrerons toutes les façons d'écrire des littéraux de chaîne, puis nous présenterons les deux types de chaîne de Rust. Nous fournissons plus de détails sur les chaînes et la gestion du texte au chapitre 17.

Littéraux de chaîne

Chaîne de caractèresles littéraux sont entourés de guillemets doubles. Ils utilisent les mêmes séquences d'échappement antislash que les `char` littéraux :

```
let speech = "\"Ouch!\" said the well.\n";
```

Dans les littéraux de chaîne, contrairement aux `char` littéraux, les guillemets simples n'ont pas besoin d'une barre oblique inverse, contrairement aux guillemets doubles.

Une chaîne peut s'étendre sur plusieurs lignes :

```
println!("In the room the women come and go,
    Singing of Mount Abora");
```

Le caractère de saut de ligne dans ce littéral de chaîne est inclus dans la chaîne et donc dans la sortie. Il en va de même pour les espaces au début de la deuxième ligne.

Si une ligne d'une chaîne se termine par une barre oblique inverse, le caractère de saut de ligne et l'espace de tête sur la ligne suivante sont supprimés :

```
println!("It was a bright, cold day in April, and \
    there were four of us—\
    more or less.");
```

Cela imprime une seule ligne de texte. La chaîne contient un seul espace entre "et" et "là" car il y a un espace avant la barre oblique inverse dans le programme, et aucun espace entre le tiret cadratin et "plus".

Dans quelques cas, la nécessité de doubler chaque antislash dans une chaîne est une nuisance. (Les exemples classiques sont les expressions régulières et les chemins Windows.) Pour ces cas, Rust propose *des chaînes brutes*. Une chaîne brute est étiquetée avec la lettre minuscule `r`. Toutes les barres obliques inverses et les espaces blancs à l'intérieur d'une chaîne brute sont inclus textuellement dans la chaîne. Aucune séquence d'échappement n'est reconnue :

```
let default_win_install_path = r"C:\Program Files\Gorillas";

let pattern = Regex::new(r"\d+(\.\d+)*");
```

Vous ne pouvez pas inclure un guillemet double dans une chaîne brute simplement en mettant une barre oblique inverse devant - rappelez-vous, nous avons dit *qu'aucune* séquence d'échappement n'est reconnue. Cependant, il existe également un remède à cela. Le début et la fin d'une chaîne brute peuvent être marqués par des signes dièse :

```
println!(r###"
    This raw string started with 'r###"'.
    Therefore it does not end until we reach a quote mark ('"')
    followed immediately by three pound signs ('###'):
"###);
```

Vous pouvez ajouter aussi peu ou autant de signes dièse que nécessaire pour indiquer clairement où se termine la chaîne brute.

Chaînes d'octets

Un stringlittéral avec le `b` préfixe est une *chaîne d'octets* . Une telle chaîne est une tranche de `u8` valeurs, c'est-à-dire des octets, plutôt qu'un texte Unicode :

```
let method = b"GET";
assert_eq!(method, &[b'G', b'E', b'T']);
```

Le type de `method` est `&[u8; 3]` : c'est une référence à un tableau de trois octets. Il n'a aucune des méthodes de chaîne dont nous parlerons dans une minute. La chose qui ressemble le plus à une chaîne est la syntaxe que nous avons utilisée pour l'écrire.

Les chaînes d'octets peuvent utiliser toutes les autres syntaxes de chaîne que nous avons présentées : elles peuvent s'étendre sur plusieurs lignes, utiliser des séquences d'échappement et utiliser des barres obliques inverses pour joindre des lignes. Les chaînes d'octets brutes commencent par `br"` .

Les chaînes d'octets ne peuvent pas contenir de caractères Unicode arbitraires. Ils doivent se contenter d'ASCII et de `\xHH` séquences d'échappement.

Chaînes en mémoire

RouillerLes chaînes sont des séquences de caractères Unicode, mais elles ne sont pas stockées en mémoire sous forme de tableaux de `char` s. Au lieu de cela, ils sont stockés en utilisant UTF-8, un codage à largeur variable. Chaque caractère ASCII d'une chaîne est stocké dans un octet. Les autres caractères occupent plusieurs octets.

La figure 3-3 montre les valeurs `String` et `&str` créées par le code suivant :

```
let noodles = "noodles".to_string();
let oodles = &noodles[1..];
let poodles = "ಠ_ಠ";
```

A `String` a un tampon redimensionnable contenant du texte UTF-8. Le tampon est alloué sur le tas, il peut donc redimensionner son tampon selon les besoins ou à la demande. Dans l'exemple, `noodles` est a `String` qui possède un tampon de huit octets, dont sept sont en cours d'utilisation. Vous pouvez considérer a `String` comme un `Vec<u8>` qui est garanti pour contenir un UTF-8 bien formé ; en fait, c'est ainsi qu'il `String` est mis en œuvre.

A `&str` (prononcé « stir » ou « string slice ») est une référenceà une série de texte UTF-8 appartenant à quelqu'un d'autre : il "emprunte" le texte. Dans l'exemple, `oodles` est une `&str` référence aux six derniers octets du texte appartenant à `noodles` , il représente donc le texte "oodles". Comme les autres références de tranche, a `&str` est un pointeur gras, contenant à la fois l'adresse des données réelles et leur longueur. Vous pouvez considérer a `&str` comme n'étant rien de plus qu'un `&[u8]` qui est garanti pour contenir un UTF-8 bien formé.

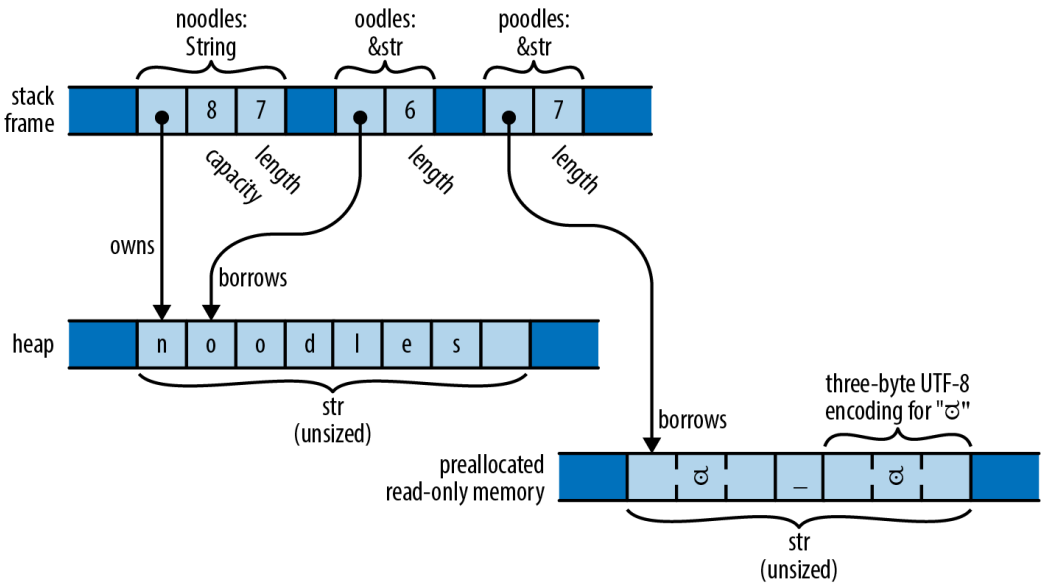

Illustration 3-3. `String` , `&str` , et `str`

Un littéral de chaîne est un `&str` qui fait référence à du texte préalloué, généralement stocké dans une mémoire en lecture seule avec le code machine du programme. Dans l'exemple précédent, `poodles` est un littéral de chaîne, pointant vers sept octets qui sont créés lorsque le programme commence l'exécution et qui durent jusqu'à ce qu'il se termine.

Méthode A `String` ou `&str` _ `.len()` renvoie sa longueur. La longueur est mesurée en octets, pas en caractères :

```
assert_eq!("ಠ_ಠ".len(), 7);
assert_eq!("ಠ_ಠ".chars().count(), 3);
```

Il est impossible de modifier un `&str` :

```
let mut s = "hello";
s[0] = 'c';     // error: `&str` cannot be modified, and other reasons
s.push('\n');  // error: no method named `push` found for reference `&str`
```

Pour créer de nouvelles chaînes au moment de l'exécution, utilisez `String`.

Le type `&mut str` existe, mais il n'est pas très utile, car presque toutes les opérations sur UTF-8 peuvent modifier sa longueur globale en octets, et une tranche ne peut pas réallouer son référent. En fait, les seules opérations disponibles sur `&mut str` sont `make_ascii_uppercase` et `make_ascii_lowercase`, qui modifient le texte en place et n'affectent que les caractères à un octet, par définition.

Chaîne de caractères

`&str` ressemble beaucoup à `&[T]` : un gros pointeur vers des données. `String` est analogue à `Vec<T>`, comme décrit dans le Tableau 3-11 .

Tableau 3-11. `Vec<T>` et `String` comparaison

	Vec<T>	Chaîne de caractères
Libère automatiquement les tampons	Oui	Oui
Cultivable	Oui	Oui
`::new()` et `::with_capacity()` fonctions associées au type	Oui	Oui
`.reserve()` et `.capacity()` méthodes	Oui	Oui
`.push()` et `.pop()` méthodes	Oui	Oui
Syntaxe de plage `v[start..stop]`	Oui, retours `&[T]`	Oui, retours `&str`
Conversion automatique	`&Vec<T>` à `&[T]`	`&String` à `&str`
Hérite des méthodes	De `&[T]`	De `&str`

Comme un `Vec`, chacun `String` a son propre tampon alloué par tas qui n'est partagé avec aucun autre `String`. Lorsqu'une `String` variable sort de la portée, le tampon est automatiquement libéré, sauf si `String` elle a été déplacée.

Il existe plusieurs façons de créer des `String`s :

- La `.to_string()` méthodeconvertitun `&str` à un `String`. Cela copie la chaîne :

  ```
  let error_message = "too many pets".to_string();
  ```

  La `.to_owned()` méthodefait la même chose, et vous pouvez le voir utilisé de la même manière. Cela fonctionne également pour d'autres types, comme nous le verrons au chapitre 13 .

- La `format!()` macrofonctionne comme `println!()`, saufqu'il renvoie un nouveau `String` au lieu d'écrire du texte sur stdout, et qu'il n'ajoute pas automatiquement une nouvelle ligne à la fin :

```
assert_eq!(format!("{}°{:02}'{:02}"N", 24, 5, 23),
           "24°05'23"N".to_string());
```

- Les tableaux, les tranches et les vecteurs de chaînes ont deux méthodes, `.concat()` et `.join(sep)`, qui forment un nouveau `String` à partir de plusieurs chaînes :

```
let bits = vec!["veni", "vidi", "vici"];
assert_eq!(bits.concat(), "venividivici");
assert_eq!(bits.join(", "), "veni, vidi, vici");
```

Le choix se pose parfois du type à utiliser : `&str` ou `String`. Le chapitre 5 aborde cette question en détail. Pour l'instant, il suffira de souligner que a `&str` peut faire référence à n'importe quelle tranche de n'importe quelle chaîne, qu'il s'agisse d'un littéral de chaîne(stocké dans l'exécutable) ou a `String` (alloué et libéré à l'exécution). Cela signifie qu'il `&str` est plus approprié pour les arguments de fonction lorsque l'appelant doit être autorisé à transmettre l'un ou l'autre type de chaîne.

Utilisation de chaînes

Les chaînes prennent en charge les opérateurs `==` et . `!=` Deux chaînes sont égales si elles contiennent les mêmes caractères dans le même ordre (qu'elles pointent ou non vers le même emplacement en mémoire) :

```
assert!("ONE".to_lowercase() == "one");
```

Les chaînes prennent également en charge la comparaisonopérateurs `<`, `<=`, `>`, et `>=`, ainsi que de nombreuses méthodes et fonctions utiles que vous pouvez trouver dans la documentation en ligne sous « `str` (type primitif) » ou le `std::str` module « » (ou passez simplement au chapitre 17). Voici quelques exemples:

```
assert!("peanut".contains("nut"));
assert_eq!("ಠ_ಠ".replace("ಠ", "■"), "■_■");
assert_eq!("    clean\n".trim(), "clean");

for word in "veni, vidi, vici".split(", ") {
```

```
        assert!(word.starts_with("v"));
    }
```

Gardez à l'esprit que, compte tenu de la nature d'Unicode, une simple `char` comparaison par `char` comparaison ne donne *pas* toujours les réponses attendues. Par exemple, les chaînes Rust `"th\u{e9}"` et `"the\u{301}"` sont toutes deux des représentations Unicode valides pour thé , le mot français pour thé. Unicode indique qu'ils doivent être affichés et traités de la même manière, mais Rust les traite comme deux chaînes complètement distinctes. De même, les opérateurs de commande de Rust `<` utilisent un ordre lexicographique simple basé sur des valeurs de points de code de caractères. Cet ordre ne ressemble que parfois à l'ordre utilisé pour le texte dans la langue et la culture de l'utilisateur. Nous abordons ces questions plus en détail au chapitre 17 .

Autres types de type chaîne

Rust garantit que les chaînes sont valides UTF-8. Parfois, un programme doit vraiment être capable de gérer des chaînes qui ne sont *pas* valides en Unicode. Cela se produit généralement lorsqu'un programme Rust doit interagir avec un autre système qui n'applique pas de telles règles. Par exemple, dans la plupart des systèmes d'exploitation, il est facile de créer un fichier avec un nom de fichier qui n'est pas Unicode valide. Que doit-il se passer lorsqu'un programme Rust rencontre ce type de nom de fichier ?

La solution de Rust consiste à proposer quelques types de type chaîne pour ces situations :

- Tenez-vous en à `String` et `&str` pour le texte Unicode.
- Lorsque vous travaillez avec des noms de fichiers, utilisez `std::path::PathBuf` et à la `&Path` place.
- Lorsque vous travaillez avec des données binaires qui ne sont pas du tout encodées en UTF-8, utilisez `Vec<u8>` et `&[u8]` .
- Lorsque vous travaillez avec des noms de variables d'environnement et des arguments de ligne de commande sous la forme native présentée par le système d'exploitation, utilisez `OsString` et `&OsStr` .
- Lors de l'interopérabilité avec des bibliothèques C qui utilisent des chaînes terminées par un caractère nul, utilisez `std::ffi::CString` et `&CStr` .

Tapez les alias

Le `type` mot clépeut être utilisé comme `typedef` en C++ pour déclarer un nouveau nom pour un type existant :

```
type Bytes = Vec<u8>;
```

Le type `Bytes` que nous déclarons ici est un raccourci pour ce type particulier de `Vec` :

```
fn decode(data:&Bytes) {
    ...
}
```

Au-delà des bases

Les types sont une partie centrale de Rust. Nous continuerons à parler des types et à en introduire de nouveaux tout au long du livre. En particulier, les types définis par l'utilisateur de Rust donnent au langage une grande partie de sa saveur, car c'est là que les méthodes sont définies. Il existe trois types de types définis par l'utilisateur, et nous les aborderons dans trois chapitres successifs : les structures au chapitre 9 , les énumérations au chapitre 10 et les traits au chapitre 11 .

Les fonctions et les fermetures ont leurs propres types, traités au chapitre 14 . Et les types qui composent la bibliothèque standard sont couverts tout au long du livre. Par exemple, le chapitre 16 présente les types de collection standard.

Tout cela devra cependant attendre. Avant de poursuivre, il est temps d'aborder les concepts qui sont au cœur de la sécurité de Rustrègles.