Chapitre 3. Types fondamentaux

> *Il y a beaucoup, beaucoup de types de livres dans le monde, ce qui est logique, parce qu'il y a beaucoup, beaucoup de types de gens, et tout le monde veut lire quelque chose de différent.*
>
> —Lemony Snicket

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

Ce chapitre couvre les types fondamentaux de Rust pour représenter les valeurs. Ces types au niveau de la source ont des homologues concrets au niveau de la machine avec des coûts et des performances prévisibles. Bien que Rust ne promet pas qu'il représentera les choses exactement comme vous l'avez demandé, il prend soin de s'écarter de vos demandes uniquement 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 à l'avance. Vous devez épeler les types d'arguments de fonction et de valeurs de retour, les champs destruct et quelques autres constructions. Cependant, deux caractéristiques de Rust rendent cela moins difficile que vous ne le pensez:

- Compte tenu des types que vous épelez, *l'inférence de type* de Rust déterminera 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'éliminer*, le type. Par exemple, vous pouvez épeler chaque type d'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 doit être un , un vecteur d'entiers signés 16 bits ; aucun autre type ne fonctionnerait. Et il s'ensuit que chaque élément du vecteur doit être un . C'est exactement le genre de raisonnement que l'inférence de type rust applique, vous permettant d'écrire à la place: `v Vec<i16> i16`

```
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 sens. L'inférence de type redonne une grande partie de la lisibilité des langages typés dynamiquement, tout en détectant les erreurs de type au moment de la compilation.

- Les fonctions peuvent être *génériques* : une seule fonction peut fonctionner sur des valeurs de nombreux types différents.
  En Python et JavaScript, toutes les fonctions fonctionnent de cette façon naturellement : une fonction peut fonctionner sur n'importe quelle valeur qui possède les propriétés et les méthodes dont la fonction aura besoin. (C'est la caractéristique souvent appelée *typage de canard* : s'il charlatan comme un canard, c'est un canard.) Mais c'est précisément cette flexibilité qui rend si difficile pour ces langues de détecter les erreurs de type tôt; les tests sont souvent le seul moyen d'attraper de telles erreurs. Les fonctions génériques de Rust donnent au langage un degré de la même flexibilité, tout en détectant 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 écrire, par exemple, une fonction spécifique pour chaque entier par rapport à l'écriture d'une fonction générique qui gère tous les entiers. Nous discuterons en détail des fonctions génériques au chapitre 11. `sum`

Le reste de ce chapitre couvre les types de Rust de bas en haut, en commençant par des 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 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 dans Rust

Type	Description	Valeurs
`i8`,,,, ,,,, `i16 i32 i64 i128 u8 u16 u32 u64 u128`	Entiers signés et non signés, d'une largeur de bits donnée	`42`, ,,, , (littéral de l'octet) – `5i8 0x400u16 0o100i16 20_922_789_888_000u64 b'*' u8`
`isize`, `usize`	Entiers signés et non signés, de la 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, 32 bits de large	`'*'`,,,, `'\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 }`
`struct T (i32, char);`	Structure de type Tuple	`T(120, 'X')`

Type	Description	Valeurs
`struct E;`	Structure de type unité; n'a pas de champs	`E`
`enum Attend { OnTime, Late(u32) }`	Énumération, type de données algébriques	`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 modifiables : 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 : pointeur non propriétaire vers le texte UTF-8 `str`	`"そば: soba"`, `&s[0..12]`
`[f64; 4]`, `[u8; 256]`	Tableau, 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]`
`&[u8]`, `&mut [u8]`	Référence à la tranche : référence à une partie d'un tableau ou d'un vecteur, comprenant le pointeur et la longueur	`&v[10..20]`, `&mut a[..]`

Type	Description	Valeurs
`Option<&str>`	Valeur facultative : (absent) ou (présent, avec valeur `None` `Some(v)` `v`)	`Some("Dr."), None`
`Result<u64, Error>`	Résultat de l'opération susceptible d'échouer : soit une valeur de réussite, soit une erreur `Ok(v)` `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 le présent chapitre, à l'exception des éléments suivants :

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

Types numériques à largeur fixe

La base du système de type 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 sont adéquats pour la plupart des applications et peuvent être des milliers de fois plus rapides que les représentations telles que les entiers de précision arbitraire et les rationnels exacts. Si vous avez besoin de ce type de représentations numériques, elles sont prises en charge dans la caisse. `num`

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 rust

Taille (bits)	Entier non signé	Entier signé	Virgule flottante
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 d'entiers

Les types entiers non signés de Rust utilisent leur plage complète pour représenter des valeurs positives et nulles (tableau 3-3).

Tableau 3-3. Types entiers non signés Rust

Type	Gamme
`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, soit 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

Les types entiers signés de Rust utilisent la représentation complémentaire des 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

Type	Gamme
`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 ou -2^{63} à 2^{63}–1

Rust utilise le 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 valeurs. `u8 u8`

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

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

Les littéraux entiers dans Rust peuvent prendre un suffixe indiquant leur type : est une valeur, et est un . 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 l'épingle : stockée dans une variable d'un type particulier, transmise à une fonction qui attend un type particulier, par rapport à une autre valeur d'un type particulier, ou quelque chose comme ça. En fin de compte, si plusieurs types peuvent fonctionner, Rust par défaut si cela fait partie des possibilités. Sinon, Rust signale l'ambiguïté comme une erreur. `42u8 u8 1729isize isize i32`

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

Pour rendre les nombres longs plus lisibles, vous pouvez insérer des traits de soulignement parmi les chiffres. Par exemple, vous pouvez écrire la plus grande valeur sous la forme . 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 , ou définir le suffixe de type à partir des chiffres, comme dans . Quelques exemples de littéraux entiers sont illustrés dans le tableau 3-5. `u32 4_294_967_295 0xffff_ffff 127_u8`

Tableau 3-5. Exemples de littéraux entiers

Littéral	Type	Valeur décimale
`116i8`	`i8`	116
`0xcafeu32`	`u32`	51966
`0b0010_1010`	Déduit	42
`0o106`	Déduit	70

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

Il y a quelques caractères que vous ne pouvez pas simplement placer après la seule citation, car ce 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	Littéral octet	Équivalent numérique
Devis unique, `'`	`b'\''`	`39u8`
Backslash `\`	`b'\\'`	`92u8`
Newline	`b'\n'`	`10u8`
Retour chariot	`b'\r'`	`13u8`
Onglet	`b'\t'`	`9u8`

Pour les caractères difficiles à écrire ou à lire, vous pouvez écrire leur code en hexadécimal à la place. Un littéral octet de la forme , où est tout nombre hexadécimal à deux chiffres, représente l'octet dont la valeur est

. Par exemple, vous pouvez écrire un littéral d'octet pour le caractère de contrôle ASCII « escape » comme , puisque le code ASCII pour « escape » est 27, ou 1B en hexadécimal. Étant donné que les littéraux d'octets ne sont qu'une autre notation pour les valeurs, demandez-vous si un littéral numérique simple pourrait être plus lisible : il est probablement logique de l'utiliser au lieu de simplement seulement lorsque vous voulez souligner que la valeur représente un code ASCII. `b'\xHH' HH HH b'\x1b' u8 b'\x1b' 27`

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

```
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 en tant que méthodes sur des 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 toutefois que la documentation contient des pages distinctes pour le type lui-même sous « (type primitif) », et pour le module dédié à ce type (recherchez « »). `i32 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. Lorsque ce n'est pas le cas, cependant, les messages d'erreur peuvent être surprenants. Par exemple, les éléments suivants ne sont pas compilés :

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

Rust se plaint:

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

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

```
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, alors soyez prudent lorsque vous appliquez des méthodes à des valeurs annulées. Sans les parenthèses dans la première instruction, appliquerait la méthode à la valeur positive , produisant positif , puis annulerait cela, produisant . `-4_i32 -4_i32.abs() abs 4 4 -4`

Arithmétique vérifiée, enveloppante, saturée et débordante

Lorsqu'une opération arithmétique d'entier déborde, Rust panique, dans une version de débogage. Dans une version release, l'opération *s'enroule* : elle 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 release, cette multiplication s'enroule 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 quelle version :

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

Ces méthodes arithmétiques d'entiers se répartissent en quatre catégories générales :

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

```
// 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);
```

- Les opérations *d'encapsulation* renvoient 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 constructions.

- Les opérations *de saturation* renvoient la valeur représentable la plus proche du résultat mathématiquement correct. En d'autres termes, le résultat est « serré » 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 méthodes de division saturée, de reste ou de décalage binaire.

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

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

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

```
// 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 , , , ou préfixe sont indiqués dans le tableau 3-7. `checked_` `wrapping_` `saturating_` `overflowing_`

Tableau 3-7. Noms des opérations

Opération	Suffixe du nom	Exemple
Addition	`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`
Exponentiation	`pow`	`3_u8.checked_pow(4) == Some(81)`
Décalage bito-gauche	`shl`	`10_u32.wrapping_shl(34) == 40`
Décalage binaire vers la droite	`shr`	`40_u64.wrapping_shr(66) == 10`

Types à virgule flottante

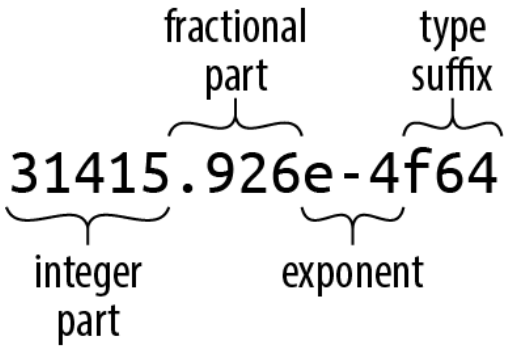
Rust fournit des types à virgule flottante IEEE à simple et double précision. Ces types comprennent des infinis positifs et négatifs, des valeurs nulles positives et négatives distinctes et *une valeur non numérique* (tableau 3-8).

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

Type	Précision	Gamme
`f32`	Ieee précision unique (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 et correspondent aux types et en C et C++ (dans les implémentations qui prennent en charge la virgule flottante IEEE) ainsi que Java (qui utilise toujours la virgule flottante IEEE). `f32 f64 float double`

Les littéraux à virgule flottante ont la forme générale schématisée à la figure 3-1.



Graphique 3-1. Littéral à virgule flottante

Chaque partie d'un nombre à virgule flottante après la partie entière est facultative, mais au moins une partie fractionnaire, exposant ou suffixe de type doit être présente pour la distinguer d'un littéral entier. La partie fractionnaire peut être constituée d'une virgule décimale unique, de même qu'une constante à virgule flottante valide. `5.`

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 type à virgule flottante pourrait convenir, il choisit par défaut. `f64`

Aux fins de l'inférence de type, Rust traite les littéraux entiers et les littéraux à virgule flottante comme des classes distinctes : il ne déduira ja-

mais un type à virgule flottante pour un littéral entier, ou vice versa. Le tableau 3-9 présente quelques exemples de littéraux à virgule flottante.

Tableau 3-9. Exemples de littéraux à virgule flottante

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

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

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

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

```
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 veillez à mettre correctement entre parenthèses les appels de méthode sur les valeurs annulées.

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

Lorsque vous recherchez dans la documentation, rappelez-vous qu'il existe des pages pour les deux types eux-mêmes, nommées " (type primitif) » et « (type primitif) », et les modules pour chaque type, et `. f32 f64 std::f32 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 faites, il suffit de mettre un type sur le littéral ou la fonction:

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

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

L'absence de conversions implicites rend parfois une expression Rust plus verbeuse 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, l'acte d'écrire des conversions numériques dans Rust nous a alertés sur des problèmes que nous aurions autrement manqués.

Nous expliquons exactement comment les conversions se comportent dans « Type Casts ».

Le type de bool

Le type booléen de Rust, , a les deux valeurs habituelles pour ces types, et . Les opérateurs de comparaison aiment et produisent des résultats: la valeur de est `. bool true false == < bool 2 < 5 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 des caractères, des entiers, des nom-

bres à virgule flottante et des pointeurs en valeurs booléennes, de sorte qu'ils peuvent être utilisés directement comme condition dans une instruction ou. Python autorise les chaînes, les listes, les dictionnaires et même les ensembles dans les contextes booléens, traitant ces valeurs comme true si elles ne sont pas vides. La rouille, cependant, est très stricte: les structures de contrôle aiment et exigent que leurs conditions soient des expressions, tout comme les opérateurs logiques de court-circuit et . Vous devez écrire , pas simplement
`. if while if while bool && || if x != 0 { ... } if x { ... }`

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

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

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

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

Caractères

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

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

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

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

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

Personnage	Caractère rouille littéral
Devis unique, `'`	`'\''`
Backslash `\`	`'\\'`
Newline	`'\n'`
Retour chariot	`'\r'`
Onglet	`'\t'`

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

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

A contient toujours un point de code Unicode dans la plage 0x0000 à 0xD7FF ou 0xE000 à 0x10FFFF. A n'est jamais une moitié de paire de substitution (c'est-à-dire un point de code dans la plage 0xD800 à 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 des contrôles dynamiques pour s'assurer que les valeurs sont toujours dans la plage autorisée. `char char char`

Rust ne convertit jamais implicitement entre et tout autre type. Vous pouvez utiliser l'opérateur de conversion pour convertir a en un type entier ;

pour les types inférieurs à 32 bits, les bits supérieurs de la valeur du caractère sont tronqués : `char as char`

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

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

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

```
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 type standard de Rust et la gestion du texte en général dans « Types de chaînes ». `String`

Tuples

Un *tuple* est une paire, ou triple, quadruple, quintuple, etc. (d'où *n-tuple*, ou *tuple*), de valeurs de types assortis. Vous pouvez écrire un tuple sous la forme d'une séquence d'éléments, séparés par des virgules et entourés de parenthèses. Par exemple, est un tuple dont le premier élément est une chaîne allouée statiquement et dont le second est un entier ; son type est . Étant donné une valeur tuple , vous pouvez accéder à ses éléments en

tant que , , et ainsi de suite. `("Brazil", 1985) (&str, i32) t t.0 t.1`

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 être tous du même type. De plus, les tuples n'autorisent que les constantes en tant qu'indices, comme . Vous ne pouvez pas écrire ou obtenir le ème élément. `t.4 t.i t[i] i`

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

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

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

```
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 passer 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 et membres, mais c'est une notation assez lourde

pour quelque chose d'aussi évident, alors nous avons juste utilisé un tuple: `width height`

```
/// Write the buffer `pixels`, whose dimensions are given by `bounds`,
/// file named `filename`.
fn write_image(filename: &str, pixels: &[u8], bounds: (usize, usize))
    -> Result<(), std::io::Error>
{ ... }
```

Le type du paramètre est , un tuple de deux valeurs. Certes, nous pourrions tout aussi bien écrire des paramètres séparés, 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. `bounds (usize, usize) usize width height`

L'autre type de tuple couramment utilisé est le zéro-tuple . C'est ce qu'on appelle traditionnellement 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 fonction de la bibliothèque standard n'a pas de valeur de retour significative ; il ne fait qu'échanger les valeurs de ses deux arguments. La déclaration pour se lit comme suit:
`() std::mem::swap std::mem::swap`

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

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

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

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

Si vous le souhaitez, vous pouvez inclure une virgule après le dernier élément d'un tuple : les types et sont équivalents, tout comme les expressions et . Rust permet systématiquement une virgule de fin supplémentaire partout où des virgules sont utilisées : arguments de fonction, tableaux, définitions struct et enum, etc. Cela peut sembler étrange pour les lecteurs humains, mais cela peut rendre les diffs plus faciles à lire lorsque des entrées sont ajoutées et supprimées à la fin d'une liste. `(&str, i32,)` `(&str, i32)` `("Brazil", 1985,)` `("Brazil", 1985)`

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

Types de pointeurs

Rust a plusieurs types qui représentent des adresses mémoire.

C'est une grande différence entre Rust et la plupart des langues avec garbage collection. En Java, si contient un champ , alors est une référence à un autre objet créé séparément. Les objets ne contiennent jamais physiquement d'autres objets en Java. `class Rectangle` `Vector2D upperLeft;` `upperLeft` `Vector2D`

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

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 des types de pointeur. La bonne nouvelle est que les types de pointeurs utilisés dans Rust sécurisé sont contraints d'éliminer les comportements non définis, 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 dangereux.

Références

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

Il est plus facile de commencer en pensant aux références comme le type de pointeur de base de Rust. Au moment de l'exécution, une référence à an est un mot de machine unique contenant l'adresse du , qui peut se trouver sur la pile ou dans le tas. L'expression produit une référence à ; dans la terminologie Rust, nous disons qu'il *emprunte une référence à* `x`. Étant donné une référence , l'expression fait référence à la valeur pointe vers. Ceux-ci 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 son champ d'application. `i32 i32 &x x r *r r & *`

Contrairement aux pointeurs C, cependant, les références Rust ne sont jamais nulles : il n'y a tout simplement aucun moyen de produire une référence nulle dans Rust sûr. 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 libres et l'invalidation du pointeur sont exclues au moment de la compilation.

Les références à la rouille se déclinent en deux saveurs :

`&T`

: Une référence immuable et partagée. Vous pouvez avoir de nombreuses références partagées à une valeur donnée à la fois, mais elles sont en lecture seule : il est interdit de modifier la valeur vers laquelle elles pointent, comme dans C. `const T*`

`&mut T`

: Une référence mutable et exclusive. Vous pouvez lire et modifier la valeur vers laquelle il pointe, comme avec a en C. Mais tant que la référence existe, vous n'avez peut-être pas d'autres références d'aucune sorte à cette valeur. En fait, la seule façon d'accéder à la valeur est via la référence modifiable. `T*`

Rust utilise cette dichotomie entre les références partagées et mutables pour appliquer une règle « un seul écrivain *ou* plusieurs lecteurs »: soit vous pouvez lire et écrire la valeur, soit elle peut être partagée par n'im-

porte quel nombre de lecteurs, mais jamais les deux en même temps. Cette séparation, appliquée par des contrôles 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 de référence sûre.

Boîtes

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

Pointeurs bruts

Rust a également les types de pointeurs bruts et . 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 qu'il pointe. Par exemple, les pointeurs bruts peuvent être null ou pointer vers une mémoire qui a été libérée ou qui contient maintenant une valeur d'un type différent. Toutes les erreurs de pointeur classiques de C ++ sont proposées pour votre plaisir. `*mut T *const T`

Toutefois, vous ne pouvez déréférencer que les pointeurs bruts d'un bloc. Un bloc est le mécanisme d'opt-in de Rust pour les fonctionnalités linguistiques avancées dont la sécurité dépend de vous. Si votre code n'a pas de blocs (ou si ceux qu'il fait sont écrits correctement), alors les garanties de sécurité que nous soulignons tout au long de ce livre sont toujours valables. Pour plus de détails, voir le chapitre 22. `unsafe unsafe unsafe`

Tableaux, vecteurs et tranches

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

- Le type représente un tableau de valeurs, chacune de type . 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. `[T; N]` `N` `T`
- Le type , appelé *vecteur de `T`s*, est une séquence de valeurs de type cultivable allouée dynamiquement. Les éléments d'un vecteur vivent sur le tas, vous pouvez donc redimensionner les vecteurs à volonté : poussez-y de nouveaux éléments, ajoutez-y d'autres vecteurs, supprimez des éléments, etc. `Vec<T>` `T`
- Les types et , *appelés tranche partagée de `T`s* et *tranche mutable 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, ainsi qu'un décompte du nombre d'éléments auxquels vous pouvez accéder à partir de ce point. Une tranche modifiable vous permet de lire et de modifier des éléments, mais ne peut pas être partagée ; une tranche partagée vous permet de partager l'accès entre plusieurs lecteurs, mais ne vous permet pas de modifier des éléments. `&[T]` `&mut [T]` `&mut [T]` `&[T]`

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

Tableaux

Il existe plusieurs façons d'écrire des valeurs de tableau. Le plus simple est d'écrire une série 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 tableau long rempli d'une 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: 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 aucune sorte de valeur non initialisée.) `[0u8; 1024]`

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

Les méthodes utiles que vous souhaitez voir sur les tableaux (itération sur les éléments, recherche, tri, remplissage, filtrage, etc.) sont toutes fournies sous forme de méthodes sur des tranches, et non de tableaux. Mais Rust convertit implicitement une référence à un tableau en tranche lors de la recherche de méthodes, de sorte que vous pouvez 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 méthode est en fait définie sur des tranches, mais comme elle prend son opérande par référence, Rust produit implicitement une tranche se référant à l'ensemble du tableau et la transmet pour opérer. En fait, la

méthode que nous avons mentionnée précédemment est également une méthode de tranche. Nous couvrons les tranches plus en détail dans « Tranches ». `sort &mut [i32] sort len`

Vecteurs

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

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

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

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

Comme pour les tableaux, vous pouvez utiliser des méthodes de tranche sur des 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 méthode est en fait définie sur des tranches, mais l'appel emprunte implicitement une tranche au vecteur et l'appelle. `reverse &mut [&str] reverse`

`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 existants. Nous les couvrirons au chapitre 16.

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

Si vous connaissez le nombre d'éléments dont un vecteur aura besoin à l'avance, vous pouvez appeler pour créer un vecteur avec un tampon suffisamment grand pour les contenir tous, dès le début; ensuite, vous pouvez ajouter les éléments au vecteur un à la fois sans provoquer de réaffectation. La 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. `Vec::new Vec::with_capacity vec!`

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

Tout comme la méthode d'un vecteur renvoie le nombre d'éléments qu'elle contient maintenant, sa méthode renvoie le nombre d'éléments qu'elle pourrait contenir sans réaffectation : `len capacity`

```
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, de sorte qu'ils peuvent être lents 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 méthode pour supprimer le dernier élément et le renvoyer. Plus précisément, faire apparaître une valeur à partir de a renvoie un : si le vecteur était déjà vide, ou si son dernier élément avait été : `pop Vec<T> Option<T> None Some(v) 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 boucle pour itérer sur un vecteur : `for`

```
// 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 Scheme 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, est un type ordinaire défini dans Rust, non intégré dans le langage. Nous couvrirons les techniques nécessaires à la

mise en œuvre de tels types au chapitre 22. `Vec`

Tranches

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

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 référence et la référence en références de tranche qui pointent directement vers les données. `&Vec<f64>` `&[f64; 4]`

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

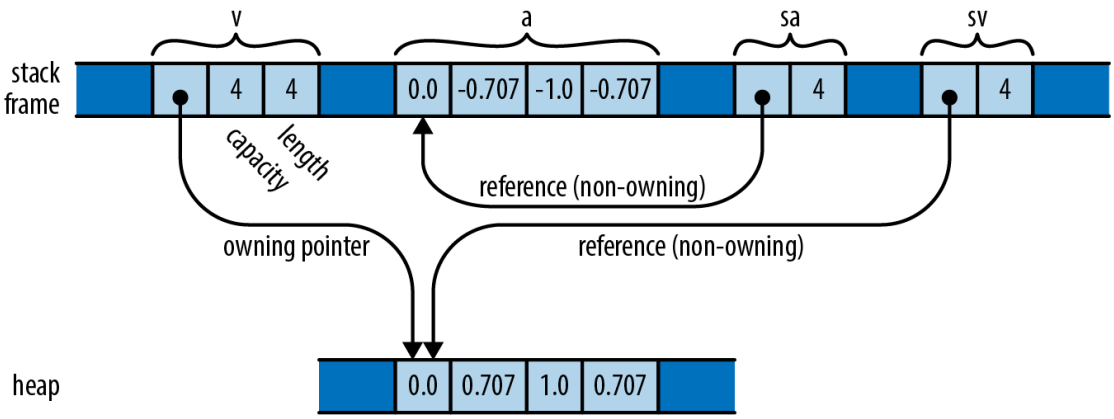


Figure 3-2. Un vecteur et un tableau en mémoire, avec des tranches et se référant à chacun `v` `a` `sa` `sv`

Alors qu'une référence ordinaire est un pointeur non propriétaire vers une seule valeur, 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 fonctionne 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 illustré. 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 et, qui trient ou inversent une séquence d'éléments en place, sont en fait des méthodes sur le type de tranche . `sort reverse [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 index 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 simplement à des types comme ou comme des « tranches », en utilisant le nom plus court pour le concept le plus commun. `&[T] &str`

Types de chaînes

Les programmeurs familiers avec C++ se souviendront qu'il existe deux types de chaînes dans le langage. Les littéraux de chaîne ont le type de pointeur . La bibliothèque standard propose également une classe, , pour

créer dynamiquement des chaînes au moment de l'exécution. `const char * std::string`

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

Littéraux de chaîne

Les littéraux de chaîne sont placés entre guillemets doubles. Ils utilisent les mêmes séquences d'échappement de barre oblique inverse que les littéraux : `char`

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

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

Une chaîne peut s'étendre sur plusieurs lignes :

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

Le caractère de nouvelle 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 nouvelle ligne et l'espace blanc de début de 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 em et « more ».

Dans quelques cas, la nécessité de doubler chaque barre oblique inverse d'une chaîne est une nuisance. (Les exemples classiques sont les expres-

sions régulières et les chemins d'accès Windows.) Pour ces cas, Rust propose des *cordes brutes*. Une chaîne brute est marquée avec la lettre minuscule . 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 : `r`

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

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

Vous ne pouvez pas inclure un caractère à guillemets doubles dans une chaîne brute simplement en plaçant une barre oblique inverse devant elle – rappelez-vous, nous avons dit qu'aucune séquence d'échappement *n'est* reconnue. Cependant, il existe également un remède pour 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 littéral de chaîne avec le préfixe est une *chaîne d'octets*. Une telle chaîne est une tranche de valeurs, c'est-à-dire des octets, plutôt que du texte Unicode : `b u8`

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

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

Les chaînes d'octets peuvent utiliser toutes les autres syntaxes de chaîne que nous avons montrées : 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 de séquences ASCII et d'échappement. `\xHH`

Chaînes en mémoire

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

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

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

A a un tampon redimensionnable contenant du texte UTF-8. La mémoire tampon est allouée sur le tas, de sorte qu'il peut redimensionner sa mémoire tampon selon les besoins ou la demande. Dans l'exemple, est un qui possède un tampon de huit octets, dont sept sont utilisés. Vous pouvez considérer un comme un qui est garanti de contenir UTF-8 bien formé; en fait, c'est ainsi que l'on met en œuvre. `String noodles String String Vec<u8> String`

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

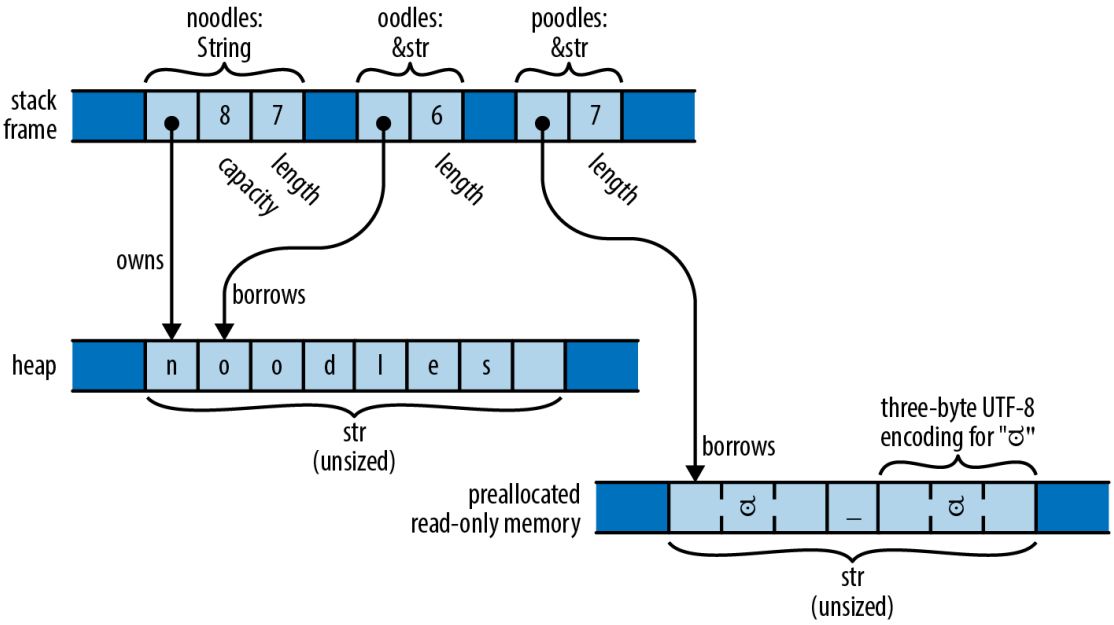


Figure 3-3. , , et `String &str str`

Un littéral de chaîne est un qui fait référence au texte préalloué, généralement stocké dans la mémoire en lecture seule avec le code machine du programme. Dans l'exemple précédent, 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 ferme. `&str poodles`

La méthode de A ou renvoie sa longueur. La longueur est mesurée en octets, et non en caractères : `String &str .len()`

```
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 `&s
```

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

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

Corde

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

Tableau 3-11. et comparaison `Vec〈T〉 String`

	Vec<T>	**Corde**
Libère automatiquement les tampons	Oui	Oui
Cultivable	Oui	Oui
`::new()` et fonctions associées au type `::with_capacity()`	Oui	Oui
`.reserve()` et méthodes `.capacity()`	Oui	Oui
`.push()` et méthodes `.pop()`	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 , chacun a son propre tampon alloué au tas qui n'est partagé avec aucun autre . Lorsqu'une variable sort de la portée, la mémoire tampon est automatiquement libérée, sauf si elle a été déplacée. `Vec String String String String`

Il existe plusieurs façons de créer s : `String`

- La méthode convertit a en . Cela copie la chaîne : `.to_string() &str String`

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

  La méthode fait la même chose, et vous pouvez la voir utilisée de la même manière. Cela fonctionne également pour d'autres types, comme

nous en discuterons au chapitre 13. `.to_owned()`

- La macro fonctionne comme , sauf qu'elle renvoie un nouveau texte au lieu d'écrire du texte dans stdout, et elle n'ajoute pas automatiquement une nouvelle ligne à la fin : `format!() println!() String`

```
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, et , qui forment une nouvelle à partir de nombreuses chaînes : `.concat() .join(sep) String`

```
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 : ou . Le chapitre 5 aborde cette question en détail. Pour l'instant, il suffira de souligner que a 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 d'un (alloué et libéré au moment de l'exécution). Cela signifie que cela est plus approprié pour les arguments de fonction lorsque l'appelant doit être autorisé à passer l'un ou l'autre type de chaîne. `&str String &str String &str`

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 les opérateurs de comparaison , , et , ainsi que de nombreuses méthodes et fonctions utiles que vous pouvez trouver dans la documentation en ligne sous « (type primitif) » ou le module « » (ou simplement basculer vers le chapitre 17). En voici quelques exemples : `< <= > >= str std::str`

```
assert!("peanut".contains("nut"));
assert_eq!("☉_☉".replace("☉", "■"), "■_■");
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 comparaison ne donne *pas* toujours les réponses attendues. Par exemple, les chaînes Rust et sont toutes deux des représentations Unicode valides pour thé, le mot Français pour thé. Unicode dit qu'ils devraient tous deux être affichés et traités de la même manière, mais Rust les traite comme deux chaînes complètement distinctes. De même, les opérateurs d'ordre de Rust utilisent un ordre lexicographique simple basé sur des valeurs de point de code de caractère. 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. `char char "th\u{e9}" "the\u{301}" <`

Autres types de chaînes

Rust garantit que les chaînes sont valides UTF-8. Parfois, un programme a vraiment besoin de pouvoir traiter des chaînes qui *ne sont pas* des Unicode valides. 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 valide Unicode. Que devrait-il se passer lorsqu'un programme Rust rencontre ce type de nom de fichier?

La solution de Rust est d'offrir quelques types de chaînes pour ces situations:

- S'en tenir à et pour le texte Unicode. `String &str`
- Lorsque vous travaillez avec des noms de fichiers, utilisez et à la place. `std::path::PathBuf &Path`
- Lorsque vous travaillez avec des données binaires qui ne sont pas du tout codées en UTF-8, utilisez et . `Vec<u8> &[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 et . `OsString &OsStr`
- Lors de l'interopérabilité avec des bibliothèques C qui utilisent des chaînes terminées par une valeur NULL, utilisez et . `std::ffi::CString &CStr`

Tapez des alias

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

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

Le type que nous déclarons ici est un raccourci pour ce type particulier de : `Bytes 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 couvrirons dans trois chapitres successifs : les structs du chapitre 9, les enums du chapitre 10 et les traits du chapitre 11.

Les fonctions et les fermetures ont leurs propres types, couverts 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 attendre, cependant. Avant de passer à autre chose, il est temps de s'attaquer aux concepts qui sont au cœur des règles de sécurité de Rust.