Chapitre 6. Expressions

> *Les programmeurs LISP connaissent la valeur de tout, mais le coût de rien.*
>
> —Alan Perlis, épigramme #55

Dans ce chapitre, nous couvrirons les *expressions* de Rust, les blocs de construction qui composent le corps des fonctions Rust et donc la majorité du code Rust. La plupart des choses dans Rust sont des expressions. Dans ce chapitre, nous allons explorer la puissance que cela apporte et comment travailler avec ses limites. Nous couvrirons le flux de contrôle, qui dans Rust est entièrement orienté expression, et comment les opérateurs fondamentaux de Rust fonctionnent de manière isolée et combinée.

Quelques concepts qui entrent techniquement dans cette catégorie, tels que les fermetures et les itérateurs, sont suffisamment profonds pour que nous leur consacrions un chapitre entier plus tard. Pour l'instant, notre objectif est de couvrir autant de syntaxe que possible en quelques pages.

Un langage d'expression

Rouillerressemble visuellement à la famille des langages C, mais c'est un peu une ruse. En C, il y a une nette distinction entre *les expressions,* des morceaux de code qui ressemblent à ceci :

```
5 * (fahr-32) / 9
```

et *des déclarations,* qui ressemblent plus à ceci :

```
for (; begin != end; ++begin) {
    if (*begin == target)
        break;
}
```

Expressionsont des valeurs. Les déclarations ne le font pas.

Rust est ce qu'on appelle un *langage d'expression* . Cela signifie qu'il suit une tradition plus ancienne, remontant à Lisp, où les expressions font tout le travail.

En C, `if` et `switch` sont des instructions. Ils ne produisent pas de valeur et ne peuvent pas être utilisés au milieu d'une expression. Dans Rust,

`if` et `match` *peut* produire des valeurs. Nous avons déjà vu une `match` expression qui produit une valeur numérique au chapitre 2 :

```
    pixels[r * bounds.0 + c] =
        match escapes(Complex { re: point.0, im:point.1 }, 255) {
            None => 0,
            Some(count) => 255 - count as u8
        };
```

Une `if` expression peut être utilisée pour initialiser une variable :

```
    let status =
        if cpu.temperature <= MAX_TEMP {
            HttpStatus:: Ok
        } else {
            HttpStatus::ServerError  // server melted
        };
```

Une `match` expression peut être passée en argument à une fonction ou une macro :

```
    println!("Inside the vat, you see {}.",
        match vat.contents {
            Some(brain) => brain.desc(),
            None => "nothing of interest"
        });
```

Cela explique pourquoi Rust n'a pas l'opérateur ternaire de C ( ). En C, il s'agit d'un analogue pratique au niveau de l'expression de l' instruction. Ce serait redondant en Rust : l' expression gère les deux cas. `expr1 ? expr2 : expr3 if if`

La plupart des outils de flux de contrôle en C sont des instructions. Dans Rust, ce sont toutes des expressions.

Priorité et associativité

Le tableau 6-1 résume la syntaxe des expressions Rust. Nous aborderons tous ces types d'expressionsdans ce chapitre. Les opérateurs sont classés par ordre de priorité, du plus élevé au plus bas. (Comme la plupart des langages de programmation, Rust a la *priorité des opérateurs*pour déterminer l'ordre des opérations lorsqu'une expression contient plusieurs opérateurs adjacents. Par exemple, dans `limit < 2 * broom.size + 1` , l' `.` opérateur a la priorité la plus élevée, donc l'accès au champ a lieu en premier.)

Tableau 6-1. Expressions

Type d'expression	Exemple	Traits associés
Littéral de tableau	`[1, 2, 3]`	
Répéter le littéral du tableau	`[0; 50]`	
Tuple	`(6, "crullers")`	
Regroupement	`(2 + 2)`	
Bloquer	`{ f(); g() }`	
Expressions de flux de contrôle	`if ok { f() }`	
	`if ok { 1 } else { 0 }`	
	`if let Some(x) = f() { x } else { 0 }`	
	`match x { None => 0, _ => 1 }`	
	`for v in e { f(v); }`	`std::iter::IntoIterator`
	`while ok { ok = f(); }`	
	`while let Some(x) = it.next() { f(x); }`	
	`loop { next_event(); }`	
	`break`	
	`continue`	
	`return 0`	

Type d'expression	Exemple	Traits associés
Appel de macro	`println!("ok")`	
Chemin	`std::f64::consts::PI`	
Littéral de structure	`Point {x: 0, y: 0}`	
Accès au champ Tuple	`pair.0`	`Deref`, `DerefMut`
Accès au champ de structure	`point.x`	`Deref`, `DerefMut`
Appel de méthode	`point.translate(50, 50)`	`Deref`, `DerefMut`
Appel de fonction	`stdin()`	`Fn(Arg0, ...) -> T`, `FnMut(Arg0, ...) -> T`, `FnOnce(Arg0, ...) -> T`
Indice	`arr[0]`	`Index`, , `IndexMut` `Deref` `DerefMut`
Vérification des erreurs	`create_dir("tmp")?`	
NON logique/au niveau du bit	`!ok`	`Not`
Négation	`-num`	`Neg`
Déréférence	`*ptr`	`Deref`, `DerefMut`
Emprunter	`&val`	
Fonte de type	`x as u32`	

Type d'expression	Exemple	Traits associés
Multiplication	`n * 2`	`Mul`
Division	`n / 2`	`Div`
Reste (module)	`n % 2`	`Rem`
Ajout	`n + 1`	`Add`
Soustraction	`n - 1`	`Sub`
Décalage à gauche	`n << 1`	`Shl`
Décalage à droite	`n >> 1`	`Shr`
ET au niveau du bit	`n & 1`	`BitAnd`
OU exclusif au niveau du bit	`n ^ 1`	`BitXor`
OU au niveau du bit	`n | 1`	`BitOr`
Moins que	`n < 1`	`std::cmp::PartialOrd`
Inférieur ou égal	`n <= 1`	`std::cmp::PartialOrd`
Plus grand que	`n > 1`	`std::cmp::PartialOrd`
Meilleur que ou égal	`n >= 1`	`std::cmp::PartialOrd`
Égal	`n == 1`	`std::cmp::PartialEq`
Inégal	`n != 1`	`std::cmp::PartialEq`

Type d'expression	Exemple	Traits associés
ET logique	`x.ok && y.ok`	
OU logique	`x.ok || backup.ok`	
Fin de gamme exclusive	`start .. stop`	
Fin de gamme inclusive	`start ..= stop`	
Mission	`x = val`	
Affectation composée	`x *= 1`	`MulAssign`
	`x /= 1`	`DivAssign`
	`x %= 1`	`RemAssign`
	`x += 1`	`AddAssign`
	`x -= 1`	`SubAssign`
	`x <<= 1`	`ShlAssign`
	`x >>= 1`	`ShrAssign`
	`x &= 1`	`BitAndAssign`
	`x ^= 1`	`BitXorAssign`
	`x |= 1`	`BitOrAssign`
Fermeture	`|x, y| x + y`	

Tous les opérateurs qui peuvent utilement être chaînés sont associatifs à gauche. Autrement dit, une chaîne d'opérations telle que `a - b - c` est regroupée en tant que `(a - b) - c`, et non `a - (b - c)`. Les opérateurs qui peuvent être chaînés de cette manière sont tous ceux auxquels vous pourriez vous attendre :

```
* / % + - << >> & ^ | && || comme
```

Les opérateurs de comparaison, les opérateurs d'affectation et les opérateurs de plage `..` et `..=` ne peuvent pas du tout être chaînés.

Blocs et points-virgules

Blocssont le type d'expression le plus général. Un bloc produit une valeur et peut être utilisé partout où une valeur est nécessaire :

```
let display_name = match post.author() {
    Some(author) => author.name(),
    None => {
        let network_info = post.get_network_metadata()?;
        let ip = network_info.client_address();
        ip.to_string()
    }
};
```

Le code après `Some(author) =>` est l'expression simple `author.name()`. Le code après `None =>` est une expression de bloc. Cela ne fait aucune différence pour Rust. La valeur du bloc est la valeur de sa dernière expression, `ip.to_string()`.

Notez qu'il n'y a pas de point-virguleaprès l' `ip.to_string()` appel de la méthode. La plupart des lignes de code Rust se terminent par un point-virgule ou des accolades, tout comme C ou Java. Et si un bloc ressemble à du code C, avec des points-virgules à tous les endroits familiers, il fonctionnera comme un bloc C et sa valeur sera `()`. Comme nous l'avons mentionné au chapitre 2 , lorsque vous laissez le point- virgule hors de la dernière ligne d'un bloc, cela fait de la valeur du bloc la valeur de son expression finale, plutôt que l'habituel `()`.

Dans certains langages, en particulier JavaScript, vous êtes autorisé à omettre les points-virgules, et le langage les remplit simplement pour vous, une commodité mineure. Ceci est différent. Dans Rust, le point-virgule signifie en fait quelque chose :

```
let msg = {
    // let-declaration: semicolon is always required
    let dandelion_control = puffball.open();

    // expression + semicolon: method is called, return value dropped
    dandelion_control.release_all_seeds(launch_codes);

    // expression with no semicolon: method is called,
    // return value stored in `msg`
    dandelion_control.get_status()
};
```

Cette capacité des blocs à contenir des déclarations et à produire également une valeur à la fin est une fonctionnalité intéressante, qui devient rapidement naturelle. Le seul inconvénient est que cela conduit à un message d'erreur étrange lorsque vous omettez un point-virgule par accident :

```
...
if preferences.changed() {
    page.compute_size()  // oops, missing semicolon
}
...
```

Si vous faites cette erreur dans un programme C ou Java, le compilateur vous signalera simplement qu'il vous manque un point-virgule. Voici ce que dit Rust :

```
error: mismatched types
22 |          page.compute_size()  // oops, missing semicolon
   |          ^^^^^^^^^^^^^^^^^^- help: try adding a semicolon: `;`
   |          |
   |          expected (), found tuple
   |
   = note: expected unit type `()`
              found tuple `(u32, u32)`
```

Sans le point-virgule, la valeur du bloc serait ce qui `page.compute_size()` revient, mais un `if` sans `else` doit toujours revenir `()`. Heureusement, Rust a déjà vu ce genre de chose et suggère d'ajouter le point- virgule.

Déclarations

En plus des expressions et des points-virgules, un bloc peut contenir n'importe quel nombre de déclarations. Les plus courantes sont les `let` déclarations, qui déclarent des variables locales :

```
let name : type = expr ;
```

Le type et l'initialiseur sont facultatifs. Le point-virgule est obligatoire. Comme tous les identifiants dans Rust, les noms de variables doivent commencer par une lettre ou un trait de soulignement, et ne peuvent contenir des chiffres qu'après ce premier caractère. Rust a une définition large de «lettre»: elle comprend les lettres grecques, les caractères latins accentués et bien d'autres symboles, tout ce que l'annexe standard Unicode # 31 déclare appropriée. Les emoji ne sont pas autorisés.

Une `let` déclaration peut déclarer une variable sans l'initialiser. La variable peut alors être initialisée avec une affectation ultérieure. Ceci est parfois utile, car parfois une variable doit être initialisée à partir du milieu d'une sorte de construction de flux de contrôle :

```
let name;
if user.has_nickname() {
    name = user.nickname();
} else {
    name = generate_unique_name();
    user.register(&name);
}
```

Ici, il existe deux manières différentes d' `name` initialiser la variable locale, mais dans les deux cas, elle sera initialisée exactement une fois, et `name` n'a donc pas besoin d'être déclarée `mut` .

C'est une erreur d'utiliser une variable avant qu'elle ne soit initialisée. (Ceci est étroitement lié à l'erreur d'utilisation d'une valeur après qu'elle ait été déplacée. Rust veut vraiment que vous n'utilisiez les valeurs que tant qu'elles existent !)

Vous pouvez occasionnellement voir du code qui semble redéclarer une variable existante, comme ceci :

```
for line in file.lines() {
    let line = line?;
    ...
}
```

La `let` déclaration crée une nouvelle, deuxième variable, d'un type différent. Le type de la première variable `line` est `Result<String, io::Error>` . Le second `line` est un `String` . Sa définition remplace la première pour le reste du bloc. C'est ce qu'on appelle l' *ombrage*et est très courant dans les programmes Rust. Le code est équivalent à :

```
for line_result in file.lines() {
    let line = line_result?;
    ...
}
```

Dans ce livre, nous nous en tiendrons à l'utilisation d'un `_result` suffixe dans de telles situations afin que les variables aient des noms distincts.

Un bloc peut également contenir *des déclarations d'éléments*. Un élément est simplement n'importe quelle déclaration qui pourrait apparaître glo-

balement dans un programme ou un module, comme un `fn`, `struct` ou `use`.

Les chapitres suivants couvriront les éléments en détail. Pour l'instant, `fn` faitun exemple suffisant. Tout bloc peut contenir un `fn` :

```
use std:: io;
use std:: cmp::Ordering;

fn show_files() -> io::Result<()> {
    let mut v = vec![];
    ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) ->Ordering {
        a.timestamp.cmp(&b.timestamp)    // first, compare timestamps
            .reverse()                   // newest file first
            .then(a.path.cmp(&b.path))   // compare paths to break ties
    }

    v.sort_by(cmp_by_timestamp_then_name);
    ...
}
```

Lorsqu'un `fn` est déclaré à l'intérieur d'un bloc, sa portée est le bloc entier, c'est-à-dire qu'il peut être *utilisé* dans tout le bloc englobant. Mais un imbriqué `fn` ne peut pas accéder aux variables locales ou aux arguments qui se trouvent dans la portée. Par exemple, la fonction `cmp_by_timestamp_then_name` ne pouvait pas utiliser `v` directement. (Rust a également des fermetures, qui voient dans les portées englobantes. Voir le chapitre 14 .)

Un bloc peut même contenir un module entier. Cela peut sembler un peu long - avons-nous vraiment besoin de pouvoir imbriquer *chaque* élément du langage dans chaque autre élément ? - mais les programmeurs (et en particulier les programmeurs utilisant des macros) ont un moyen de trouver des utilisations pour chaque morceau d'orthogonalité fourni par le langage..

si et correspondre

La forme d'une `if` expressionest familier :

```
si condition1 {
    bloc1
} sinon si condition2 {
    bloc2
} sinon {
```

```
    bloc_n
}
```

Chacun `condition` doitêtre une expression de type `bool` ; fidèle à la forme, Rust ne convertit pas implicitement les nombres ou les pointeurs en valeurs booléennes.

Contrairement à C, les parenthèses ne sont pas nécessaires autour des conditions. En fait, `rustc` émettra un avertissement si des parenthèses inutiles sont présentes. Les accolades sont cependant obligatoires.

Les `else if` blocs, ainsi que le final `else` , sont facultatifs. Une `if` expression sans bloc se comporte exactement comme si elle avait un bloc `else` vide . `else`

`match` expressionssont quelque chose comme l' `switch` instruction C, mais plus flexible. Un exemple simple :

```
match code {
    0 => println!("OK"),
    1 => println!("Wires Tangled"),
    2 => println!("User Asleep"),
    _ => println!("Unrecognized Error {}", code)
}
```

C'est quelque chose qu'une `switch` déclaration pourrait faire. Exactement l'un des quatre bras de cette `match` expression s'exécutera, en fonction de la valeur de `code` . Le modèle générique `_` correspond à tout. C'est comme le `default:` cas dans une `switch` instruction, sauf qu'elle doit venir en dernier ; placer un `_` motif avant d'autres motifs signifie qu'il aura priorité sur eux. Ces modèles ne correspondront jamais à rien (et le compilateur vous en avertira).

Le compilateur peut optimiser ce type d' `match` utilisation d'une table de saut, tout comme une `switch` instruction en C++. Une optimisation similaire est appliquée lorsque chaque bras de a `match` produit une valeur constante. Dans ce cas, le compilateur construit un tableau de ces valeurs, et le `match` est compilé dans un accès au tableau. Hormis une vérification des limites, il n'y a pas de branchement du tout dans le code compilé.

La polyvalence de `match` découle de la variété des *modèles pris en charge*qui peut être utilisé à gauche de `=>` dans chaque bras. Ci-dessus, chaque motif est simplement un entier constant. Nous avons également montré `match` des expressions qui distinguent les deux types de `Option` valeur :

```
match params.get("name") {
    Some(name) => println!("Hello, {}!", name),
    None => println!("Greetings, stranger.")
}
```

Ce n'est qu'un aperçu de ce que les modèles peuvent faire. Un modèle peut correspondre à une plage de valeurs. Il peut décompresser les tuples. Il peut correspondre à des champs individuels de structures. Il peut rechercher des références, emprunter des parties d'une valeur, etc. Les motifs de Rust sont un mini-langage à part entière. Nous leur consacrerons plusieurs pages dans le chapitre 10 .

La forme générale d'une `match` expression est :

```
 valeur de  correspondance {
    motif  =>  expr ,
    ...
}
```

La virgule après un bras peut être supprimée si le `expr` est un bloc.

Rust vérifie les données `value` par rapport à chaque motif à tour de rôle, en commençant par le premier. Lorsqu'un modèle correspond, le correspondant `expr` est évalué et l' `match` expression est complète ; aucun autre motif n'est vérifié. Au moins un des motifs doit correspondre. Rust interdit les `match` expressions qui ne couvrent pas toutes les valeurs possibles :

```
let score = match card.rank {
    Jack => 10,
    Queen => 10,
    Ace => 11
};  // error: nonexhaustive patterns
```

Tous les blocs d'une `if` expression doivent produire des valeurs du même type :

```
let suggested_pet =
    if with_wings { Pet:: Buzzard } else { Pet::Hyena };  // ok

let favorite_number =
    if user.is_hobbit() { "eleventy-one" } else { 9 };  // error

let best_sports_team =
    if is_hockey_season() { "Predators" };  // error
```

(Le dernier exemple est une erreur car en juillet, le résultat serait `()` .)

De même, tous les bras d'une `match` expression doivent avoir le même type :

```
let suggested_pet =
    match favorites.element {
        Fire => Pet:: RedPanda,
        Air => Pet:: Buffalo,
        Water => Pet::Orca,
        _ => None  // error: incompatible types
    };
```

si laissé

Làest une `if` forme de plus, l' `if let` expression :

```
if let pattern = expr {
    block1
} else {
    block2
}
```

Le donné `expr` correspond soit `pattern` à , auquel cas `block1` s'exécute, soit ne correspond pas, et `block2` s'exécute. Parfois, c'est un bon moyen d'extraire des données d'un `Option` ou `Result` :

```
if let Some(cookie) = request.session_cookie {
    return restore_session(cookie);
}

if let Err(err) = show_cheesy_anti_robot_task() {
    log_robot_attempt(err);
    politely_accuse_user_of_being_a_robot();
} else {
    session.mark_as_human();
}
```

Il n'est jamais strictement *nécessaire* d'utiliser `if let` , car `match` peut tout `if let` faire. Une `if let` expression est un raccourci pour un `match` avec un seul motif:

```
match expr {
    motif => { bloc1 }
    _ => { bloc2 }
}
```

Boucles

Il y a quatre bouclesexpressions:

```
tant que condition {
    bloc
}

while let pattern = expr {
    bloc
}

boucle {
    bloc
}


pour motif dans iterable {
    bloc
}
```

Les boucles sont des expressions dans Rust, mais la valeur d'une boucle `while` ou `for` est toujours `()`, donc leur valeur n'est pas très utile. Une `loop` expression peut produire une valeur si vous en spécifiez une.

Une `while` boucle se comporte exactement comme l'équivalent C, sauf que, encore une fois, le `condition` doit être du type exact `bool`.

La `while let` boucleest analogue à `if let`. Au début de chaque itération de boucle, la valeur de `expr` correspond à la valeur donnée `pattern`, auquel cas le bloc s'exécute, ou non, auquel cas la boucle se termine.

Utiliser `loop` pour écrire des boucles infinies. Il exécute le `block` à plusieurs reprises pour toujours (ou jusqu'à ce qu'un `break` or `return` soit atteint ou que le thread panique).

Une `for` boucle évalue l' `iterable` expression, puis évalue `block` une fois pour chaque valeur dans l'itérateur résultant. De nombreux types peuvent être itérés, y compris toutes les collections standard telles que `Vec` et `HashMap`. `for` La boucle C standard :

```
for (int i = 0; i < 20; i++) {
    printf("%d\n", i);
}
```

s'écrit ainsi en Rust :

```
for i in 0..20 {
    println!("{}", i);
```

```
}
```

Comme en C, le dernier nombre imprimé est `19`.

L'`..` opérateur produit une *gamme*, une structure simple avec deux champs : `start` et `end`. `0..20` est le même que `std::ops::Range { start: 0, end: 20 }`. Les plages peuvent être utilisées avec des `for` boucles car `Range` il s'agit d'un type itérable : il implémente le `std::iter::IntoIterator` trait, dont nous parlerons au chapitre 15. Les collections standard sont toutes itérables, tout comme les tableaux et les tranches.

Conformément à la sémantique de déplacement de Rust, une `for` boucle sur une valeur consomme la valeur :

```
let strings:Vec<String> = error_messages();
for s in strings {                  // each String is moved into s here...
    println!("{}", s);
}                                   // ...and dropped here
println!("{} error(s)", strings.len()); // error: use of moved value
```

Cela peut être gênant. Le remède simple consiste à boucler sur une référence à la collection à la place. La variable de boucle sera alors une référence à chaque élément de la collection :

```
for rs in &strings {
    println!("String {:?} is at address {:p}.", *rs, rs);
}
```

Ici, le type de `&strings` est `&Vec<String>`, et le type de `rs` est `&String`.

L'itération sur une `mut` référence fournit une `mut` référence à chaque élément :

```
for rs in &mut strings {  // the type of rs is &mut String
    rs.push('\n');  // add a newline to each string
}
```

Le chapitre 15 couvre les `for` boucles plus en détail et montre de nombreuses autres façons d'utiliser les itérateurs.

Flux de contrôle dans les boucles

Une `break` expressionsort d'une boucle englobante. (Dans Rust, `break` ne fonctionne que dans les boucles. Ce n'est pas nécessaire dans les `match` expressions, qui sont différentes des `switch` déclarations à cet égard.)

Dans le corps d'un `loop` , vous pouvez donner `break` une expression, dont la valeur devient celle de la boucle :

```
// Each call to `next_line` returns either `Some(line)`, where
// `line` is a line of input, or `None`, if we've reached the end of
// the input. Return the first line that starts with "answer: ".
// Otherwise, return "answer: nothing".
let answer = loop {
    if let Some(line) = next_line() {
        if line.starts_with("answer: ") {
            break line;
        }
    } else {
        break "answer: nothing";
    }
};
```

Naturellement, toutes les `break` expressions de a `loop` doivent produire des valeurs de même type, qui devient le type de lui- `loop` même.

Une `continue` expression saute à l'itération de boucle suivante :

```
// Read some data, one line at a time.
for line in input_lines {
    let trimmed = trim_comments_and_whitespace(line);
    if trimmed.is_empty() {
        // Jump back to the top of the loop and
        // move on to the next line of input.
        continue;
    }
    ...
}
```

En `for` boucle, `continue` avanceà la valeur suivante de la collection. S'il n'y a plus de valeurs, la boucle se termine. De la même manière, dans une `while` boucle, `continue` revérifie la condition de la boucle. Si c'est maintenant faux, la boucle se termine.

Une boucle peut être *étiquetée* avec une durée de vie. Dans l'exemple suivant, `'search:` est une étiquette pour la `for` boucle externe. Ainsi, `break 'search` quitte cette boucle, pas la boucle interne :

```
'search:
for room in apartment {
```

```
        for spot in room.hiding_spots() {
            if spot.contains(keys) {
                println!("Your keys are {} in the {}.", spot, room);
                break 'search;
            }
        }
    }
```

A `break` peut avoir à la fois une étiquette et une expression de valeur:

```
    // Find the square root of the first perfect square
    // in the series.
    let sqrt = 'outer:loop {
        let n = next_number();
        for i in 1.. {
            let square = i * i;
            if square == n {
                // Found a square root.
                break 'outer i;
            }
            if square > n {
                // `n` isn't a perfect square, try the next
                break;
            }
        }
    };
```

Les étiquettes peuvent également être utiliséesavec `continue`.

expression de retour

Une `return` expressionquitte la fonction en cours, renvoyant une valeur à l'appelant.

`return` sans valeur est un raccourci pour `return ()` :

```
    fn f() {     // return type omitted: defaults to ()
        return;  // return value omitted: defaults to ()
    }
```

Les fonctions n'ont pas besoin d'avoir une `return` expression explicite. Le corps d'une fonction fonctionne comme une expression de bloc : si la dernière expression n'est pas suivie d'un point-virgule, sa valeur est la valeur de retour de la fonction. En fait, c'est le moyen préféré de fournir la valeur de retour d'une fonction dans Rust.

Mais cela ne signifie pas que `return` c'est inutile, ou simplement une concession aux utilisateurs qui ne sont pas expérimentés avec les lan-

gages d'expression. Comme une `break` expression, `return` peut abandonner un travail en cours. Par exemple, au chapitre 2 , nous avons utilisé l' `?` opérateur pour vérifier les erreurs après avoir appelé une fonction qui peut échouer :

```
let output = File::create(filename)?;
```

Nous avons expliqué qu'il s'agit d'un raccourci pour une `match` expression :

```
let output = match File::create(filename) {
    Ok(f) => f,
    Err(err) => return Err(err)
};
```

Ce code commence par appeler `File::create(filename)` . Si cela revient `Ok(f)` , alors l' `match` expression entière est évaluée à `f` , donc `f` est stockée dans `output` , et nous continuons avec la ligne de code suivante après le `match` .

Sinon, nous allons faire correspondre `Err(err)` et frapper l' `return` expression. Lorsque cela se produit, peu importe que nous soyons en train d'évaluer une `match` expression pour déterminer la valeur de la variable `output` . Nous abandonnons tout cela et quittons la fonction englobante, renvoyant l'erreur que nous avons obtenue de `File::create()` .

Nous couvrirons l' `?` opérateur plus complètement dans "Propagation des erreurs" .

Pourquoi Rust a une boucle

Plusieurs éléments du compilateur Rust analysent le flux de contrôle dans votre programme :

- Rust vérifie que chaque chemin à travers une fonction renvoie une valeur du type de retour attendu. Pour le faire correctement, il doit savoir s'il est possible d'atteindre la fin de la fonction.
- Rust vérifie que les variables locales ne sont jamais utilisées non initialisées. Cela implique de vérifier chaque chemin à travers une fonction pour s'assurer qu'il n'y a aucun moyen d'atteindre un endroit où une variable est utilisée sans avoir déjà traversé le code qui l'initialise.
- Rust met en garde contre un code inaccessible. Le code est inaccessible si *aucun* chemin à travers la fonction ne l'atteint.

Ceux-ci sont dits *sensibles au flux*analyses. Ils n'ont rien de nouveau; Java a eu une analyse «d'affectation définie», similaire à celle de Rust, pendant des années.

Lors de l'application de ce type de règle, un langage doit trouver un équilibre entre la simplicité, qui permet aux programmeurs de comprendre plus facilement de quoi le compilateur parle parfois, et l'intelligence, qui peut aider à éliminer les faux avertissements et les cas où le compilateur rejette un programme sécuritaire. Rust a opté pour la simplicité. Ses analyses sensibles au flux n'examinent pas du tout les conditions de boucle, mais supposent simplement que n'importe quelle condition dans un programme peut être vraie ou fausse.

Cela amène Rust à rejeter certains programmes sûrs :

```
fn wait_for_process(process: &mut Process) ->i32 {
    while true {
        if process.wait() {
            return process.exit_code();
        }
    }
}  // error: mismatched types: expected i32, found ()
```

L'erreur ici est fausse. Cette fonction ne sort que via l' `return` instruction, donc le fait que la `while` boucle ne produise pas un `i32` n'est pas pertinent.

L' `loop` expression est proposée comme une solution "dire ce que vous voulez dire" à ce problème.

Le système de type de Rust est également affecté par le flux de contrôle. Nous avons dit précédemment que toutes les branches d'une `if` expression doivent avoir le même type. Mais il serait idiot d'appliquer cette règle aux blocs qui se terminent par une expression `break` ou , un infini ou un appel à ou . Ce que toutes ces expressions ont en commun, c'est qu'elles ne se terminent jamais de la manière habituelle, en produisant une valeur. A ou sort brusquement du bloc courant, un infini ne finit jamais du tout, et ainsi de suite. `return loop panic!`
`() std::process::exit() break return loop`

Ainsi, dans Rust, ces expressions n'ont pas de type normal. Les expressions qui ne se terminent pas normalement se voient attribuer le type spécial `!` et sont exemptées des règles concernant les types devant correspondre. Vous pouvez voir `!` dans la signature de fonction de `std::process::exit()` :

```
fn exit(code: i32) ->!
```

Le `!` moyen qui `exit()` ne revient jamais. C'est une *fonction divergente*.

Vous pouvez écrire vos propres fonctions divergentes en utilisant la même syntaxe, et c'est parfaitement naturel dans certains cas :

```
fn serve_forever(socket: ServerSocket, handler: ServerHandler) ->! {
    socket.listen();
    loop {
        let s = socket.accept();
        handler.handle(s);
    }
}
```

Bien sûr, Rust considère alors qu'il s'agit d'une erreur si la fonction peut revenir normalement.

Avec ces blocs de construction de flux de contrôle à grande échelle en place, nous pouvons passer aux expressions plus fines généralement utilisées dans ce flux, comme les appels de fonction et les opérateurs arithmétiques.

Appels de fonction et de méthode

La syntaxepour appeler des fonctions et des méthodes est la même dans Rust que dans de nombreux autres langages :

```
let x = gcd(1302, 462);  // function call

let room = player.location();  // method call
```

Dans le deuxième exemple ici, `player` est une variable de type composé `Player`, qui a une `.location()` méthode composée. (Nous montrerons comment définir vos propres méthodes lorsque nous commencerons à parler des types définis par l'utilisateur au chapitre 9 .)

Rust fait généralement une distinction nette entre les références et les valeurs auxquelles elles se réfèrent. Si vous passez a `&i32` à une fonction qui attend un `i32`, c'est une erreur de type. Vous remarquerez que l'`.` opérateur assouplit un peu ces règles. Dans l'appel de méthode `player.location()`, `player` il peut s'agir de a `Player`, d'une référence de type `&Player` ou d'un pointeur intelligent de type `Box<Player>` or `Rc<Player>`. La `.location()` méthode peut prendre le joueur soit par valeur, soit par référence. La même `.location()` syntaxe fonctionne dans tous les cas, car l'`.` opérateur de Rust déréférence automatiquement `player` ou lui emprunte une référence selon les besoins.

Une troisième syntaxe est utilisée pour appeler les fonctions associées au type, comme `Vec::new()` :

```
let mut numbers = Vec::new();  // type-associated function call
```

Celles-ci sont similaires aux méthodes statiquesdans les langages orientés objet : les méthodes ordinaires sont appelées sur des valeurs (comme `my_vec.len()`) et les fonctions associées au type sont appelées sur des types (comme `Vec::new()`).

Naturellement, les appels de méthode peuvent être chaînés :

```
// From the Actix-based web server in Chapter 2:
server
    .bind("127.0.0.1:3000").expect("error binding server to address")
    .run().expect("error running server");
```

Une bizarrerie de la syntaxe Rust est que dans un appel de fonction ou un appel de méthode, la syntaxe habituelle pour les types génériques, `Vec<T>`, ne fonctionne pas :

```
return Vec<i32>::with_capacity(1000);  // error: something about chained compa

let ramp = (0 .. n).collect<Vec<i32>>();  // same error
```

Le problème est que dans les expressions, `<` est l'opérateur inférieur à. Le compilateur Rust suggère utilement d'écrire à la `::<T>` place de `<T>` dans ce cas, et cela résout le problème :

```
return Vec:: <i32>::with_capacity(1000);  // ok, using ::<

let ramp = (0 .. n).collect::<Vec<i32>>();  // ok, using ::<
```

Le symbole `::<...>` est affectueusement connu dans la communauté de Rust sous le nom de *turbofish*.

Alternativement, il est souvent possible de supprimer les paramètres de type et de laisser Rust les déduire :

```
return Vec::with_capacity(10);  // ok, if the fn return type is Vec<i32>

let ramp:Vec<i32> = (0 .. n).collect();  // ok, variable's type is given
```

Il est considéré comme bon style d'omettre les types chaque fois qu'ils peuvent être déduits.

Champs et éléments

Les champsd'une structure sont accessibles à l'aide d'une syntaxe familière. Les tuples sont les mêmes sauf que leurs champs ont des nombres plutôt que des noms :

```
game.black_pawns    // struct field
coords.1            // tuple element
```

Si la valeur à gauche du point est une référence ou un type de pointeur intelligent, elle est automatiquement déréférencée, comme pour les appels de méthode.

Les crochets accèdent aux éléments d'un tableau, d'une tranche ou d'un vecteur :

```
pieces[i]           // array element
```

La valeur à gauche des parenthèses est automatiquement déréférencée.

Des expressions comme ces trois sont appelées *lvalues*, car ils peuvent apparaître sur le côté gauche d'un devoir :

```
game.black_pawns = 0x00ff0000_00000000_u64;
coords.1 = 0;
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

Bien sûr, cela n'est autorisé que si `game` , `coords` , et `pieces` sont déclarés comme `mut` variables.

Extraire une tranche d'un tableau ou d'un vecteur est simple :

```
let second_half = &game_moves[midpoint .. end];
```

Ici, `game_moves` il peut s'agir d'un tableau, d'une tranche ou d'un vecteur ; le résultat, quoi qu'il en soit, est une tranche empruntée de longueur `end - midpoint` . `game_moves` est considéré comme emprunté pour la durée de vie de `second_half` .

L' `..` opérateurpermet d'omettre l'un ou l'autre des opérandes ; il produit jusqu'à quatre types d'objets différents selon les opérandes présents :

```
..      // RangeFull
a ..    // RangeFrom { start: a }
```

```
.. b    // RangeTo { end: b }
a .. b  // Range { start: a, end: b }
```

Les deux dernières formes sont *exclusives à la fin*(ou *semi-ouvert*) : la valeur finale n'est pas comprise dans la plage représentée. Par exemple, la plage `0 .. 3` comprend les nombres `0` , `1` et `2` .

L' `..=` opérateurproduit des gammes *inclusives* (ou *fermées)*, qui incluent la valeur finale :

```
..= b    // RangeToInclusive { end: b }
a ..= b  // RangeInclusive::new(a, b)
```

Par exemple, la plage `0 ..= 3` comprend les nombres `0` , `1` , `2` et `3` .

Seules les plages qui incluent une valeur de départ sont itérables, car une boucle doit avoir un point de départ. Mais dans le découpage en tableaux, les six formes sont utiles. Si le début ou la fin de la plage est omis, il s'agit par défaut du début ou de la fin des données découpées.

Ainsi, une implémentation de quicksort, l'algorithme de tri classique diviser pour mieux régner, pourrait ressembler, en partie, à ceci :

```
fn quicksort<T: Ord>(slice:&mut [T]) {
    if slice.len() <= 1 {
        return;  // Nothing to sort.
    }

    // Partition the slice into two parts, front and back.
    let pivot_index = partition(slice);

    // Recursively sort the front half of `slice`.
    quicksort(&mut slice[.. pivot_index]);

    // And the back half.
    quicksort(&mut slice[pivot_index + 1 ..]);
}
```

Opérateurs de référence

L'adresse des opérateurs, `&` et `&mut` , sont traités au chapitre 5 .

`*` L' opérateur unairepermet d'accéder à la valeur pointée par une référence. Comme nous l'avons vu, Rust suit automatiquement les références lorsque vous utilisez l' `.` opérateur pour accéder à un champ ou à une méthode, de sorte que l' `*` opérateur n'est nécessaire que lorsque nous voulons lire ou écrire la valeur entière vers laquelle pointe la référence.

Par exemple, parfois un itérateur produit des références, mais le programme a besoin des valeurs sous-jacentes :

```
let padovan:Vec<u64> = compute_padovan_sequence(n);
for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```

Dans cet exemple, le type de `elem` est `&u64`, tout `*elem` comme a `u64`.

Opérateurs arithmétiques, binaires, de comparaison et logiques

Le binaire de Rustles opérateurs sont comme ceux de beaucoup d'autres langages. Pour gagner du temps, nous supposons la familiarité avec l'une de ces langues et nous nous concentrons sur les quelques points où Rust s'écarte de la tradition.

Rust a l'arithmétique habituelleopérateurs, `+`, `-`, `*`, `/` et `%`. Comme mentionné au chapitre 3, un débordement d'entier est détecté et provoque une panique dans les versions de débogage. La bibliothèque standard fournit des méthodes telles `a.wrapping_add(b)` que l'arithmétique non vérifiée.

Entierla division arrondit vers zéro et diviser un entier par zéro déclenche une panique même dans les versions de version. Les entiers ont une méthode `a.checked_div(b)` qui renvoie un `Option(None` si `b` est égal à zéro) et ne panique jamais.

Unaire `-` nie un nombre. Il est pris en charge pour tous les types numériques à l'exception des entiers non signés. Il n'y a pas d'opérateur unaire `+`.

```
println!("{}", -100);     // -100
println!("{}", -100u32);  // error: can't apply unary `-` to type `u32`
println!("{}", +100);     // error: expected expression, found `+`
```

Comme en C, `a % b` calcule le reste signé, ou module, de la division arrondie vers zéro. Le résultat a le même signe que l'opérande de gauche. Notez que cela `%` peut être utilisé sur les nombres à virgule flottante ainsi que sur les entiers :

```
let x = 1234.567 % 10.0;  // approximately 4.567
```

Rust hérite également des C au niveau du bitopérateurs entiers, `&`, `|`, `^`, `<<` et `>>` .Cependant, Rust utilise à la `!` place de `~` pour NOT au niveau du bit :

```
let hi:u8 = 0xe0;
let lo = !hi;  // 0x1f
```

Cela signifie que `!n` cela ne peut pas être utilisé sur un nombre entier `n` pour signifier "n est égal à zéro". Pour cela, écrivez `n == 0` .

Le décalage de bits est toujours une extension de signe sur les types d'entiers signés et une extension de zéro sur les types d'entiers non signés. Puisque Rust a des entiers non signés, il n'a pas besoin d'un opérateur de décalage non signé, comme l' `>>>` opérateur de Java.

Les opérations au niveau du bit ont une priorité plus élevée que les comparaisons, contrairement au C, donc si vous écrivez `x & BIT != 0` , cela signifie `(x & BIT) != 0` , comme vous l'aviez probablement prévu. C'est bien plus utile que l'interprétation de C, `x & (BIT != 0)` , qui teste le mauvais bit !

La comparaison de Rustles opérateurs sont `==` , `!=` , `<` , `<=` , `>` et `>=` . Les deux valeurs comparées doivent avoir le même type.

Rust possède également les deux logiques de court-circuitopérateurs `&&` et `||` . Les deux opérandes doivent avoir le type exact `bool` .

Mission

L' `=` opérateurpeut être utilisé pour affecter des `mut` variables et leurs champs ou éléments. Mais affectationn'est pas aussi courant dans Rust que dans d'autres langages, car les variables sont immuables par défaut.

Comme décrit au chapitre 4 , si la valeur a un non- `Copy` type, l'affectation la *déplace* vers la destination. La propriété de la valeur est transférée de la source à la destination. La valeur précédente de la destination, le cas échéant, est supprimée.

L'affectation composée est prise en charge :

```
total += item.price;
```

Cela équivaut à `total = total + item.price;` . D'autres opérateurs sont également pris en charge : `-=` , `*=` , etc. La liste complète est donnée dans le Tableau 6-1 , plus haut dans ce chapitre.

Contrairement à C, Rust ne prend pas en charge l'affectation de chaînage : vous ne pouvez pas écrire `a = b = 3` pour affecter la valeur `3` à la fois à `a` et `b`. L'affectation est suffisamment rare dans Rust pour que vous ne manquiez pas ce raccourci.

Rust n'a pas les opérateurs d'incrémentation et de décrémentation de C `++` et `--`.

Moulages de type

La conversion d'une valeur d'un type à un autre nécessite généralement un cast expliciteà Rust. Les casts utilisent le mot- `as` clé :

```
let x = 17;              // x is type i32
let index = x as usize;  // convert to usize
```

Plusieurs types de moulages sont autorisés :

- Les nombres peuvent être convertis de n'importe lequel des types numériques intégrés en n'importe quel autre.
  La conversion d'un entier en un autre type d'entier est toujours bien définie. La conversion en un type plus étroit entraîne une troncature. Un entier signé converti en un type plus large est étendu par un signe, un entier non signé est étendu par zéro, et ainsi de suite. Bref, pas de surprise.
  La conversion d'un type à virgule flottante en un type entier arrondit vers zéro : la valeur de `-1.99 as i32` est `-1`. Si la valeur est trop grande pour tenir dans le type entier, le cast produit la valeur la plus proche que le type entier peut représenter : la valeur de `1e6 as u8` est `255`.
- Les valeurs de type `bool` ou `char`, ou d'un type de `enum` type C, peuvent être converties en n'importe quel type entier. (Nous couvrirons les énumérations au chapitre 10 .)
  La conversion dans l'autre sens n'est pas autorisée, car les types `bool`, `char` et `enum` ont tous des restrictions sur leurs valeurs qui devraient être appliquées avec des vérifications à l'exécution. Par exemple, la conversion de a `u16` en type `char` est interdite car certaines `u16` valeurs , telles que , correspondent à des points de code de substitution Unicode et ne constitueraient donc pas des valeurs `0xd800` valides . `char` Il existe une méthode standard, `std::char::from_u32()`, qui effectue la vérification à l'exécution et renvoie un `Option<char>` ; mais plus précisément, le besoin de ce type de conversion est devenu rare. Nous convertissons généralement des chaînes ou des flux entiers à la fois, et les algorithmes sur le texte

Unicode sont souvent non triviaux et il vaut mieux les laisser aux bibliothèques.
Exceptionnellement, a `u8` peut être converti en type `char`, puisque tous les entiers de 0 à 255 sont des points de code Unicode valides pour `char` tenir.
- Certains transtypages impliquant des types de pointeurs non sécurisés sont également autorisés. Voir « Pointeurs bruts ».

Nous avons dit qu'une conversion nécessite *généralement un casting*. Quelques conversions impliquant des types de référence sont si simples que le langage les exécute même sans transtypage. Un exemple trivial est la conversion d'une `mut` référence en une non- `mut` référence.

Cependant, plusieurs conversions automatiques plus importantes peuvent se produire :

- Les valeurs de type `&String` sont automatiquement converties en type `&str` sans transtypage.
- Les valeurs de type `&Vec<i32>` se convertissent automatiquement en `&[i32]`.
- Les valeurs de type `&Box<Chessboard>` se convertissent automatiquement en `&Chessboard`.

Celles-ci sont appelées *coercitions de deref*, car ils s'appliquent aux types qui implémentent le `Deref` trait intégré. Le but de `Deref` la coercition est de faire en sorte que les types de pointeurs intelligents, comme `Box`, se comportent autant que possible comme la valeur sous-jacente. Utiliser a `Box<Chessboard>` est la plupart du temps comme utiliser un plain `Chessboard`, grâce à `Deref`.

Les types définis par l'utilisateur peuvent `Deref` également implémenter le trait. Lorsque vous devez écrire votre propre type de pointeur intelligent, consultez « Deref et DerefMut ».

Fermetures

La rouille a des *fermetures*, des valeurs de type fonction légères. Une fermeture consiste généralement en une liste d'arguments, donnée entre des barres verticales, suivie d'une expression :

```
let is_even = |x| x % 2 == 0;
```

Rust déduit les types d'arguments et le type de retour. Vous pouvez également les écrire explicitement, comme vous le feriez pour une fonction. Si vous spécifiez un type de retour, le corps de la fermeture doit être un bloc, par souci de cohérence syntaxique :

```
let is_even = |x: u64| ->bool x % 2 == 0;  // error

let is_even = |x: u64| ->bool { x % 2 == 0 };  // ok
```

L'appel d'une fermeture utilise la même syntaxe que l'appel d'une fonction :

```
assert_eq!(is_even(14), true);
```

Les fermetures sont l'une des caractéristiques les plus délicieuses de Rust, et il y a beaucoup plus à dire à leur sujet. Nous le dirons au chapitre 14 .

En avant

Les expressions sont ce que nous considérons comme du "code en cours d'exécution". Ils font partie d'un programme Rust qui se compile en instructions machine. Pourtant, ils ne représentent qu'une petite fraction de l'ensemble de la langue.

Il en va de même dans la plupart des langages de programmation. La première tâche d'un programme est de s'exécuter, mais ce n'est pas sa seule tâche. Les programmes doivent communiquer. Ils doivent être testables. Ils doivent rester organisés et flexibles pour pouvoir continuer à évoluer. Ils doivent interagir avec le code et les services créés par d'autres équipes. Et même juste pour fonctionner, les programmes dans un langage typé statiquement comme Rust ont besoin de plus d'outils pour organiser les données que de simples tuples et tableaux.

À venir, nous passerons plusieurs chapitres à parler des fonctionnalités dans ce domaine : les modules et les caisses, qui donnent la structure de votre programme, puis les structures et les énumérations, qui font la même chose pour vos données..

Tout d'abord, nous consacrerons quelques pages au sujet important de ce qu'il faut faire lorsque les choses tournent mal.