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 explorerons la puissance que cela apporte et comment travailler avec ses limites. Nous couvrirons le flux de contrôle, qui dans Rust est entièrement axé sur l'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, nous visons à couvrir autant de syntaxe que possible en quelques pages.

Un langage d'expression

Rust ressemble visuellement à la famille de langages C, mais c'est un peu une ruse. En C, il y a une nette distinction entre les *expressions,* les bits 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;
}
```

Les expressions ont des valeurs. Les déclarations ne le font pas.

La rouille 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, et sont des déclarations. Ils ne produisent pas de valeur et ne peuvent pas être utilisés au milieu d'une expression. Dans Rust, et *peut* produire des valeurs. Nous avons déjà vu une expression qui produit une valeur numérique au chapitre 2 : `if switch if match match`

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

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

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

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

```
    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 dans 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 de l'expression Rust. Nous discuterons de tous ces types d'expressions dans ce chapitre. Les opérateurs sont répertoriés 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 , l'opérateur a la priorité la plus élevée, de sorte que l'accès au champ se produit en premier.) `limit < 2 * broom.size + 1 .`

Tableau 6-1. Expressions

Type d'expression	Exemple	Traits apparentés
Littéral de tableau	`[1, 2, 3]`	
Répéter le littéral de tableau	`[0; 50]`	
Tuple	`(6, "crullers")`	
Groupement	`(2 + 2)`	
Bloquer	`{ f(); g() }`	

Contrôler les expressions de flux	`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`	

Type d'expression	Exemple	Traits apparentés
	`return 0`	
Appel de macro	`println!("ok")`	
Chemin	`std::f64::consts::PI`	
Littéral Struct	`Point {x: 0, y: 0}`	
Accès au champ Tuple	`pair.0`	`Deref`, `DerefMut`
Accès au champ Struct	`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`
Index	`arr[0]`	`Index`, `IndexMut` `Deref`, `DerefMut`
Vérification des erreurs	`create_dir("tmp")?`	
Logique/bitwise NOT	`!ok`	`Not`
Négation	`-num`	`Neg`
Déréférencement	`*ptr`	`Deref`, `DerefMut`

Type d'expression	Exemple	Traits apparentés
Emprunter	`&val`	
Type coulé	`x as u32`	
Multiplication	`n * 2`	`Mul`
Division	`n / 2`	`Div`
Reste (module)	`n % 2`	`Rem`
Addition	`n + 1`	`Add`
Soustraction	`n - 1`	`Sub`
Décalage à gauche	`n << 1`	`Shl`
Décalage à droite	`n >> 1`	`Shr`
Bitwise ET	`n & 1`	`BitAnd`
Bitwise exclusif OU	`n ^ 1`	`BitXor`
Binaire OU	`n | 1`	`BitOr`
Moins de	`n < 1`	`std::cmp::PartialOrd`
Inférieur ou égal à	`n <= 1`	`std::cmp::PartialOrd`
Plus grand que	`n > 1`	`std::cmp::PartialOrd`
Supérieur ou égal à	`n >= 1`	`std::cmp::PartialOrd`

Type d'expression	Exemple	Traits apparentés
Égal	`n == 1`	`std::cmp::PartialEq`
Pas égal	`n != 1`	`std::cmp::PartialEq`
Logique ET	`x.ok && y.ok`	
Logique OU	`x.ok || backup.ok`	
Gamme exclusive finale	`start .. stop`	
Gamme end-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 être enchaînés utilement sont associatifs à gauche. C'est-à-dire une chaîne d'opérations telle que est regroupée comme , et non . Les opérateurs qui peuvent être enchaînés de cette manière sont tous ceux auxquels vous pouvez vous attendre: `a - b - c (a - b) - c a - (b - c)`

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

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

Blocs et points-virgules

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

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

Dans certains langages, en particulier JavaScript, vous êtes autorisé à omettre des points-virgules, et le langage les remplit simplement pour vous, ce qui est une commodité mineure. C'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 une valeur à la fin est une caractéristique intéressante, qui semble 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 avez fait 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)`
```

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

Déclarations

Outre les expressions et les points-virgules, un bloc peut contenir un nombre illimité de déclarations. Les plus courantes sont les déclarations, qui déclarent des variables locales : `let`

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

Le type et l'initialiseur sont facultatifs. Le point-virgule est obligatoire. Comme tous les identificateurs 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 »: il comprend des lettres grecques, des caractères latins accentués et bien d'autres symboles - tout ce que l'annexe standard Unicode n ° 31 déclare approprié. Les emoji ne sont pas autorisés.

Une déclaration peut déclarer une variable sans l'initialiser. La variable peut ensuite 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`

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

Ici, il y a deux façons différentes d'initialiser la variable locale, mais de toute façon, elle sera initialisée exactement une fois, il n'est donc pas nécessaire de la déclarer . `name name 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 son déplacement. Rust veut vraiment que vous n'utilisiez des valeurs que tant qu'elles existent!)

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

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

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

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

```
    ...
}
```

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

Un bloc peut également contenir des *déclarations d'élément*. Un élément est simplement une déclaration qui pourrait apparaître globalement dans un programme ou un module, tel qu'un , , ou . `fn struct use`

Les chapitres suivants couvriront les points en détail. Pour l'instant, c'est un exemple suffisant. Tout bloc peut contenir un : `fn 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 est déclaré à l'intérieur d'un bloc, son étendue est le bloc entier, c'est-à-dire qu'il peut être *utilisé* dans tout le bloc englobant. Mais un imbriqué ne peut pas accéder aux variables locales ou aux arguments qui se trouvent être dans la portée. Par exemple, la fonction ne pouvait pas être utilisée directement. (Rust a également des fermetures, qui se voient dans des étendues englobantes. Voir le chapitre 14.) `fn fn cmp_by_timestamp_then_name v`

Un bloc peut même contenir un module entier. Cela peut sembler un peu beaucoup – avons-nous vraiment besoin d'être en mesure d'imbriquer *chaque* morceau du langage dans tous les autres morceaux ? – 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 expression est familière : `if`

```
if condition1 {
    block1
} else if condition2 {
    block2
} else {
    block_n
}
```

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

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

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

`match` les expressions sont quelque chose comme l'instruction C, mais plus flexibles. Un exemple simple : `switch`

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

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

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

La polyvalence des provient de la variété de *motifs soutenus* qui peuvent être utilisés à gauche de dans chaque bras. Ci-dessus, chaque motif est simplement un entier constant. Nous avons également montré des expressions qui distinguent les deux types de valeur
: `match => match Option`

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

Ce n'est qu'un indice de ce que les modèles peuvent faire. Un modèle peut correspondre à une plage de valeurs. Il peut déballer les tuples. Il peut correspondre à des champs individuels de structs. Il peut chasser les références, emprunter des parties d'une valeur, et plus encore. Les motifs de Rust sont un mini-langage qui leur est propre. Nous leur consacrerons plusieurs pages au chapitre 10.

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

```
match value {
    pattern => expr,
    ...
}
```

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

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

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

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

```
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 expression doivent avoir le même type
: `match`

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

si laisser

Il existe une autre forme, l'expression: `if if let`

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

Le donné correspond à , auquel cas s'exécute, ou ne correspond pas, et s'exécute. Parfois, c'est un bon moyen d'obtenir des données d'un ou
: `expr pattern block1 block2 Option 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* de l'utiliser, car peut faire tout ce qu'il peut faire. Une expression est un raccourci pour un avec un seul motif: `if let match if let if let match`

```
match expr {
    pattern => { block1 }
    _ => { block2 }
}
```

Boucles

Il existe quatre expressions en boucle :

```
while condition {
    block
}

while let pattern = expr {
    block
}

loop {
    block
}


for pattern in iterable {
    block
}
```

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

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

La boucle est analogue à . Au début de chaque itération de boucle, la valeur de soit correspond à la donnée , auquel cas le bloc s'exécute, ou ne s'exécute pas, auquel cas la boucle se ferme. `while let if let expr pattern`

Permet d'écrire des boucles infinies. Il exécute le répété pour toujours (ou jusqu'à ce qu'un ou soit atteint ou que le fil panique). `loop block break return`

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

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

est écrit comme ceci dans Rust:

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

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

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

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

```
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 facile consiste à boucler 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 est , et le type de est . `&strings &Vec<String> rs &String`

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

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

Contrôler le flux en boucles

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

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

```
// 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 expressions d'un must produisent des valeurs avec le même type, qui devient le type de l'eux-mêmes. `break loop loop`

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

```
// 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;
    }
    ...
}
```

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

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

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

```
// 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 utilisées avec . `continue`

expressions de retour

Une expression quitte la fonction active et renvoie une valeur à l'appelant. `return`

`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 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. `return`

Mais cela ne signifie pas que c'est inutile, ou simplement une concession aux utilisateurs qui ne sont pas expérimentés avec les langages d'expression. Comme une expression, peut abandonner le travail en cours. Par exemple, dans le chapitre 2, nous avons utilisé l'opérateur pour vérifier les erreurs après l'appel d'une fonction qui peut échouer
`:return break return ?`

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

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

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

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

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

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

Pourquoi Rust Has loop

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

- Rust vérifie que chaque chemin d'accès à une fonction renvoie une valeur du type de retour attendu. Pour ce 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 sans initialisation. Cela implique de vérifier chaque chemin à travers une fonc-

tion pour s'assurer qu'il n'y a aucun moyen d'atteindre un endroit où une variable est utilisée sans avoir déjà passé par le code qui l'initialise.
- Rust met en garde contre le code inaccessible. Le code est inaccessible si aucun chemin *d'accès* à travers la fonction ne l'atteint.

C'est ce qu'on appelle des analyses *sensibles au flux*. Ils ne sont pas nouveaux; Java a 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 parle parfois le compilateur, et l'intelligence, qui peut aider à éliminer les faux avertissements et les cas où le compilateur rejette un programme parfaitement sûr. 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 d'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'instruction, de sorte que le fait que la boucle ne produise pas de an n'est pas pertinent. `return while i32`

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

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

Donc, dans Rust, ces expressions n'ont pas un type normal. Les expressions qui ne se terminent pas normalement se voient attribuer le type

spécial et sont exemptées des règles sur les types à faire correspondre. Vous pouvez voir dans la signature de fonction de `:! ! std::process::exit()`

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

Les moyens qui ne reviennent jamais. C'est une *fonction divergente*. `! exit()`

Vous pouvez écrire vos propres fonctions divergentes en utilisant la même syntaxe, ce qui 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 syntaxe pour 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, est une variable du type inventé , qui a une méthode inventée. (Nous montrerons comment définir vos propres méthodes lorsque nous commencerons à parler des types définis par l'utilisateur au chapitre 9.) `player Player .location()`

Rust fait généralement une distinction nette entre les références et les valeurs auxquelles elles se réfèrent. Si vous passez a à une fonction qui attend un , il s'agit d'une erreur de type. Vous remarquerez que l'opérateur assouplit un peu ces règles. Dans l'appel de méthode , il peut s'agir

d'un , d'une référence de type , ou d'un pointeur intelligent de type ou . La méthode peut prendre le joueur soit par valeur, soit par référence. La même syntaxe fonctionne dans tous les cas, car l'opérateur de Rust déréférence automatiquement ou emprunte une référence à celui-ci au besoin. `&i32 i32 . player.location() player Player &Player Box<Player> Rc<Player> .location() .location() . player`

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

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

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

Naturellement, les appels de méthode peuvent être enchaînés :

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

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

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

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

```
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 Rust sous le nom de *turbofish*. `::<...>`

Alternativement, il est souvent possible de laisser tomber 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 un bon style d'omettre les types chaque fois qu'ils peuvent être déduits.

Champs et éléments

Les champs d'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 un type de référence ou de pointeur intelligent, elle est automatiquement déréférencée, tout 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 crochets est automatiquement déréférencée.

Des expressions comme celles-ci sont *appelées lvalues*, car elles peuvent apparaître sur le côté gauche d'une affectation :

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

Bien entendu, cela n'est autorisé que si , et sont déclarés en tant que variables. `game coords pieces mut`

L'extraction d'une tranche à partir d'un tableau ou d'un vecteur est simple :

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

Ici peut être un tableau, une tranche ou un vecteur; le résultat, quoi qu'il en soit, est une tranche de longueur empruntée. est considéré comme em-

prunté pour la durée de vie de . `game_moves end - midpoint game_moves second_half`

L'opérateur permet d'omettre l'un ou l'autre opérande ; il produit jusqu'à quatre types d'objets différents en fonction des 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 *à moitié ouvertes*) : la valeur finale n'est pas incluse dans la plage représentée. Par exemple, la plage comprend les nombres , et . `0 .. 3 0 1 2`

L'opérateur produit des plages *finales inclusives* (ou *fermées*), qui incluent la valeur finale : `..=`

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

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

Seules les plages qui incluent une valeur de départ sont itérables, car une boucle doit avoir un endroit pour commencer. Mais dans le découpage de tableau, les six formes sont utiles. Si le début ou la fin de la plage est omis, il est par défaut le début ou la fin des données découpées.

Ainsi, une implémentation de quicksort, l'algorithme de tri classique de diviser pour 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 , sont traitées au chapitre 5. `& &mut`

L'opérateur unaire est utilisé pour accéder à la valeur indiquée 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, il arrive qu'un itérateur produise des références, mais que le programme ait 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 est , donc est un . `elem &u64 *elem u64`

Opérateurs arithmétiques, binaires, de comparaison et logiques

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

Rust a les opérateurs arithmétiques habituels, , , , , et . Comme mentionné dans le chapitre 3, le dépassement d'entier est détecté et provoque une panique dans les versions de débogage. La bibliothèque standard fournit des méthodes telles que l'arithmétique non cochée. `+ - * / % a.wrapping_add(b)`

La division des entiers arrondit vers zéro, et la division d'un entier par zéro déclenche une panique même dans les versions de version. Les entiers ont une méthode qui renvoie un (si est nul) et ne panique jamais. `a.checked_div(b) Option None b`

Unary annule un certain 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, calcule le reste signé, ou module, de l'arrondi de division vers zéro. Le résultat a le même signe que l'opérande gauche. Notez que cela peut être utilisé sur des nombres à virgule flottante ainsi que sur des entiers : `a % b %`

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

Rust hérite également des opérateurs entiers binaires de C, , , , et . Cependant, Rust utilise au lieu de pour bitwise NOT: `& | ^ << >> ! ~`

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

Cela signifie qu'il ne peut pas être utilisé sur un entier pour signifier « n est nul ». Pour cela, écrivez `n == 0 .!n n`

Le décalage de bits s'étend toujours sur les types entiers signés et l'extension zéro sur les types entiers non signés. Étant donné que Rust a des entiers non signés, il n'a pas besoin d'un opérateur shift non signé, comme l'opérateur de Java. >>>

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

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

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

Mission

L'opérateur peut être utilisé pour affecter des variables et leurs champs ou éléments. Mais l'affectation n'est pas aussi courante dans Rust que dans d'autres langages, car les variables sont immuables par défaut. `= mut`

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

L'affectation composée est prise en charge :

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

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

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

Rust n'a pas d'opérateurs d'incrément et de décrément de C et . `++ --`

Caractères moulés

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

```
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 quel type numérique intégré à 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 signe, un entier non signé est étendu à zéro, etc. Bref, il n'y a pas de surprises.
  La conversion d'un type à virgule flottante en un type entier arrondit vers zéro : la valeur de est . Si la valeur est trop grande pour tenir dans le type entier, la distribution produit la valeur la plus proche que le type entier peut représenter : la valeur de est . `-1.99 as i32 -1 1e6 as u8 255`
- Les valeurs de type ou , ou de type C, peuvent être converties en n'importe quel type entier. (Nous couvrirons les énumérations au chapitre 10.) `bool char enum`
  La conversion dans l'autre sens n'est pas autorisée, car , et les types ont tous des restrictions sur leurs valeurs qui devraient être appliquées avec des contrôles d'exécution. Par exemple, la conversion d'un type à type est interdite car certaines valeurs, comme , correspondent à des points de code de substitution Unicode et ne feraient donc pas de valeurs valides. Il existe une méthode standard, , qui effectue la vérification d'exécution et renvoie un ; 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 est préférable de les laisser aux
bibliothèques. `bool char enum u16 char u16 0xd800 char std::char::from_u32() Option<char>`
À titre d'exception, a peut être converti en type , car tous les entiers compris entre 0 et 255 sont des points de code Unicode valides pour être conservés. `u8 char char`
- Certains moulages impliquant des types de pointeurs dangereux 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 effectue même sans cast. Un exemple trivial est la conversion d'une référence en une non-référence. `mut mut`

Plusieurs conversions automatiques plus importantes peuvent se produire, cependant:

- Les valeurs de type sont converties automatiquement en type sans moulage. `&String &str`
- Valeurs de type auto-conversion en . `&Vec<i32> &[i32]`
- Valeurs de type auto-conversion en . `&Box<Chessboard> &Chessboard`

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

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

Fermetures

La rouille a *des fermetures*, des valeurs de fonction légères. Une fermeture se compose généralement d'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 santé 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 dis-le 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 compile les instructions de la 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. Le premier travail d'un programme est de s'exécuter, mais ce n'est pas son seul travail. 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 construits par d'autres équipes. Et même juste pour s'exécuter, 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 structs et les enums, qui font de même pour vos données.

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