Patterns and Matching

Patterns are a special syntax in Rust for matching against the structure of types, both complex and simple. Using patterns in conjunction with `match` expressions and other constructs gives you more control over a program's control flow. A pattern consists of some combination of the following:

- Literals
- Destructured arrays, enums, structs, or tuples
- Variables
- Wildcards
- Placeholders

Some example patterns include `x`, `(a, 3)`, and `Some(Color::Red)`. In the contexts in which patterns are valid, these components describe the shape of data. Our program then matches values against the patterns to determine whether it has the correct shape of data to continue running a particular piece of code.

To use a pattern, we compare it to some value. If the pattern matches the value, we use the value parts in our code. Recall the `match` expressions in Chapter 6 that used patterns, such as the coin-sorting machine example. If the value fits the shape of the pattern, we can use the named pieces. If it doesn't, the code associated with the pattern won't run.

This chapter is a reference on all things related to patterns. We'll cover the valid places to use patterns, the difference between refutable and irrefutable patterns, and the different kinds of pattern syntax that you might see. By the end of the chapter, you'll know how to use patterns to express many concepts in a clear way.

All the Places Patterns Can Be Used

Patterns pop up in a number of places in Rust, and you've been using them a lot without realizing it! This section discusses all the places where patterns are valid.

`match` Arms

As discussed in Chapter 6, we use patterns in the arms of `match` expressions. Formally, `match` expressions are defined as the keyword `match`, a value to match on, and one or more match arms that consist of a pattern and an expression to run if the value matches that arm's pattern, like this:

```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

For example, here's the `match` expression from Listing 6-5 that matches on an `Option<i32>` value in the variable `x`:

```
match x {
    None => None,
    Some(i) => Some(i + 1),
}
```

The patterns in this `match` expression are the `None` and `Some(i)` on the left of each arrow.

One requirement for `match` expressions is that they need to be *exhaustive* in the sense that all possibilities for the value in the `match` expression must be accounted for. One way to ensure you've covered every possibility is to have a catchall pattern for the last arm: for example, a variable name matching any value can never fail and thus covers every remaining case.

IMP

The particular pattern `_` will match anything, but it never binds to a variable, so it's often used in the last match arm. The `_` pattern can be useful when you want to ignore any value not specified, for example. We'll cover the `_` pattern in more detail in the "Ignoring Values in a Pattern" section later in this chapter.

Conditional `if let` Expressions

In Chapter 6 we discussed how to use `if let` expressions mainly as a shorter way to write the equivalent of a `match` that only matches one case. Optionally, `if let` can have a corresponding `else` containing code to run if the pattern in the `if let` doesn't match.

Listing 18-1 shows that it's also possible to mix and match `if let`, `else if`, and `else if let` expressions. Doing so gives us more flexibility than a `match` expression in which we can express only one value to compare with the patterns. Also, Rust doesn't require that the conditions in a series of `if let`, `else if`, `else if let` arms relate to each other.

The code in Listing 18-1 determines what color to make your background based on a series of checks for several conditions. For this example, we've created variables with hardcoded values that a real program might receive from user input.

Filename: src/main.rs

```
fn main() {
    let favorite_color: Option<&str> = None;
    let is_tuesday = false;
    let age: Result<u8, _> = "34".parse();

    if let Some(color) = favorite_color {
        println!("Using your favorite color, {color}, as the background");
    } else if is_tuesday {
        println!("Tuesday is green day!");
    } else if let Ok(age) = age {
        if age > 30 {
            println!("Using purple as the background color");
        } else {
            println!("Using orange as the background color");
        }
    } else {
        println!("Using blue as the background color");
    }
}
```

Listing 18-1: Mixing `if let`, `else if`, `else if let`, and `else`

If the user specifies a favorite color, that color is used as the background. If no favorite color is specified and today is Tuesday, the background color is green. Otherwise, if the user specifies their age as a string and we can parse it as a number successfully, the color is either purple or orange depending on the value of the number. If none of these conditions apply, the background color is blue.

This conditional structure lets us support complex requirements. With the hardcoded values we have here, this example will print `Using purple as the background color`.

You can see that `if let` can also introduce shadowed variables in the same way that `match` arms can: the line `if let Ok(age) = age` introduces a new shadowed `age` variable that contains the value inside the `Ok` variant. This means we need to place the `if age > 30` condition within that block: we can't combine these two conditions into `if let Ok(age) = age && age > 30`. The shadowed `age` we want to compare to 30 isn't valid until the new scope starts with the curly bracket.

The downside of using `if let` expressions is that the compiler doesn't check for exhaustiveness, whereas with `match` expressions it does. If we omitted the last `else` block and therefore missed handling some cases, the compiler would not alert us to the possible logic bug.

`while let` Conditional Loops

Similar in construction to `if let`, the `while let` conditional loop allows a `while` loop to run for as long as a pattern continues to match. In Listing 18-2 we code a `while let` loop that uses a vector as a stack and prints the values in the vector in the opposite order in which they were pushed.

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{top}");
}
```

Listing 18-2: Using a `while let` loop to print values for as long as `stack.pop()` returns `Some`

This example prints 3, 2, and then 1. The `pop` method takes the last element out of the vector and returns `Some(value)`. If the vector is empty, `pop` returns `None`. The `while` loop continues running the code in its block as long as `pop` returns `Some`. When `pop` returns `None`, the loop stops. We can use `while let` to pop every element off our stack.

`for` Loops

In a `for` loop, the value that directly follows the keyword `for` is a pattern. For example, in `for x in y` the `x` is the pattern. Listing 18-3 demonstrates how to use a pattern in a `for` loop to destructure, or break apart, a tuple as part of the `for` loop.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{value} is at index {index}");
}
```

Listing 18-3: Using a pattern in a `for` loop to destructure a tuple

The code in Listing 18-3 will print the following:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.52s
     Running `target/debug/patterns`
a is at index 0
b is at index 1
c is at index 2
```

We adapt an iterator using the `enumerate` method so it produces a value and the index for that value, placed into a tuple. The first value produced is the tuple `(0, 'a')`. When this value is matched to the pattern `(index, value)`, `index` will be `0` and `value` will be `'a'`, printing the first line of the output.

`let` Statements

Prior to this chapter, we had only explicitly discussed using patterns with `match` and `if let`, but in fact, we've used patterns in other places as well, including in `let` statements. For example, consider this straightforward variable assignment with `let`:

```
let x = 5;
```

Every time you've used a `let` statement like this you've been using patterns, although you might not have realized it! More formally, a `let` statement looks like this:

```
let PATTERN = EXPRESSION;
```

In statements like `let x = 5;` with a variable name in the `PATTERN` slot, the variable name is just a particularly simple form of a pattern. Rust compares the expression against the pattern and assigns any names it finds. So in the `let x = 5;` example, `x` is a pattern that means "bind what matches here to the variable `x`." Because the name `x` is the whole pattern, this pattern effectively means "bind everything to the variable `x`, whatever the value is."

To see the pattern matching aspect of `let` more clearly, consider Listing 18-4, which uses a pattern with `let` to destructure a tuple.

```
let (x, y, z) = (1, 2, 3);
```

Listing 18-4: Using a pattern to destructure a tuple and create three variables at once

Here, we match a tuple against a pattern. Rust compares the value `(1, 2, 3)` to the pattern `(x, y, z)` and sees that the value matches the pattern, so Rust binds `1` to `x`, `2` to `y`, and `3` to `z`. You can think of this tuple pattern as nesting three individual variable patterns inside it.

If the number of elements in the pattern doesn't match the number of elements in the tuple, the overall type won't match and we'll get a compiler error. For example, Listing 18-5 shows an attempt to destructure a tuple with three elements into two variables, which won't work.

```
let (x, y) = (1, 2, 3);
```

Listing 18-5: Incorrectly constructing a pattern whose variables don't match the number of elements in the tuple

Attempting to compile this code results in this type error:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0308]: mismatched types
 --> src/main.rs:2:9
  |
2 |     let (x, y) = (1, 2, 3);
  |         ^^^^^^   --------- this expression has type `({integer}, {integer},
{integer})`
  |         |
  |         expected a tuple with 3 elements, found one with 2 elements
  |
  = note: expected tuple `({integer}, {integer}, {integer})`
             found tuple `(_, _)`

For more information about this error, try `rustc --explain E0308`.
error: could not compile `patterns` (bin "patterns") due to 1 previous error
```

To fix the error, we could ignore one or more of the values in the tuple using `_` or `..`, as you'll see in the "Ignoring Values in a Pattern" section. If the problem is that we have too many variables in the pattern, the solution is to make the types match by removing variables so the number of variables equals the number of elements in the tuple.

Function Parameters

Function parameters can also be patterns. The code in Listing 18-6, which declares a function named `foo` that takes one parameter named `x` of type `i32`, should by now look familiar.

```
fn foo(x: i32) {
    // code goes here
}
```

Listing 18-6: A function signature uses patterns in the parameters

The `x` part is a pattern! As we did with `let`, we could match a tuple in a function's arguments to the pattern. Listing 18-7 splits the values in a tuple as we pass it to a function.

Filename: src/main.rs

```
fn print_coordinates(&(x, y): &(i32, i32)) {
    println!("Current location: ({x}, {y})");
}

fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}
```

Listing 18-7: A function with parameters that destructure a tuple

This code prints `Current location: (3, 5)`. The values `&(3, 5)` match the pattern `&(x, y)`, so `x` is the value `3` and `y` is the value `5`.

We can also use patterns in closure parameter lists in the same way as in function parameter lists, because closures are similar to functions, as discussed in Chapter 13.

At this point, you've seen several ways of using patterns, but patterns don't work the same in every place we can use them. In some places, the patterns must be irrefutable; in other circumstances, they can be refutable. We'll discuss these two concepts next.

Refutability: Whether a Pattern Might Fail to Match

Patterns come in two forms: refutable and irrefutable. Patterns that will match for any possible value passed are *irrefutable*. An example would be `x` in the statement `let x = 5;` because `x` matches anything and therefore cannot fail to match. Patterns that can fail to match for some possible value are *refutable*. An example would be `Some(x)` in the expression `if let Some(x) = a_value` because if the value in the `a_value` variable is `None` rather than `Some`, the `Some(x)` pattern will not match.

IMP

Function parameters, `let` statements, and `for` loops can only accept irrefutable patterns, because the program cannot do anything meaningful when values don't match. The `if let` and `while let` expressions accept refutable and irrefutable patterns, but the compiler warns against irrefutable patterns because by definition they're intended to handle possible failure: the functionality of a conditional is in its ability to perform differently depending on success or failure.

In general, you shouldn't have to worry about the distinction between refutable and irrefutable patterns; however, you do need to be familiar with the concept of refutability so you can respond when you see it in an error message. In those cases, you'll need to change either the pattern or the construct you're using the pattern with, depending on the intended behavior of the code.

Let's look at an example of what happens when we try to use a refutable pattern where Rust requires an irrefutable pattern and vice versa. Listing 18-8 shows a `let` statement, but for the pattern we've specified `Some(x)`, a refutable pattern. As you might expect, this code will not compile.

```
let Some(x) = some_option_value;
```

Listing 18-8: Attempting to use a refutable pattern with `let`

If `some_option_value` was a `None` value, it would fail to match the pattern `Some(x)`, meaning the pattern is refutable. However, the `let` statement can only accept an irrefutable pattern because there is nothing valid the code can do with a `None` value. At compile time, Rust will complain that we've tried to use a refutable pattern where an irrefutable pattern is required:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error[E0005]: refutable pattern in local binding
 --> src/main.rs:3:9
  |
3 |     let Some(x) = some_option_value;
  |         ^^^^^^^ pattern `None` not covered
  |
  = note: `let` bindings require an "irrefutable pattern", like a `struct` or an
`enum` with only one variant
  = note: for more information, visit https://doc.rust-lang.org/book/ch18-02-
refutability.html
  = note: the matched value is of type `Option<i32>`
help: you might want to use `let else` to handle the variant that isn't matched
  |
3 |     let Some(x) = some_option_value else { todo!() };
  |                                     ++++++++++++++++

For more information about this error, try `rustc --explain E0005`.
error: could not compile `patterns` (bin "patterns") due to 1 previous error
```

Because we didn't cover (and couldn't cover!) every valid value with the pattern `Some(x)`, Rust rightfully produces a compiler error.

If we have a refutable pattern where an irrefutable pattern is needed, we can fix it by changing the code that uses the pattern: instead of using `let`, we can use `if let`. Then if the pattern doesn't match, the code will just skip the code in the curly brackets, giving it a way to continue validly. Listing 18-9 shows how to fix the code in Listing 18-8.

```
if let Some(x) = some_option_value {
    println!("{x}");
}
```

Listing 18-9: Using `if let` and a block with refutable patterns instead of `let`

We've given the code an out! This code is perfectly valid now. However, if we give `if let` an irrefutable pattern (a pattern that will always match), such as `x`, as shown in Listing 18-10, the compiler will give a warning.

```
if let x = 5 {
    println!("{x}");
};
```

Listing 18-10: Attempting to use an irrefutable pattern with `if let`

Rust complains that it doesn't make sense to use `if let` with an irrefutable pattern:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
warning: irrefutable `if let` pattern
 --> src/main.rs:2:8
  |
2 |     if let x = 5 {
  |        ^^^^^^^^^
  |
  = note: this pattern will always match, so the `if let` is useless
  = help: consider replacing the `if let` with a `let`
  = note: `#[warn(irrefutable_let_patterns)]` on by default

warning: `patterns` (bin "patterns") generated 1 warning
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.39s
     Running `target/debug/patterns`
5
```

IMP

For this reason, match arms must use refutable patterns, except for the last arm, which should match any remaining values with an irrefutable pattern. Rust allows us to use an irrefutable pattern in a `match` with only one arm, but this syntax isn't particularly useful and could be replaced with a simpler `let` statement.

Now that you know where to use patterns and the difference between refutable and irrefutable patterns, let's cover all the syntax we can use to create patterns.

Pattern Syntax

In this section, we gather all the syntax valid in patterns and discuss why and when you might want to use each one.

Matching Literals

As you saw in Chapter 6, you can match patterns against literals directly. The following code gives some examples:

```
let x = 1;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This code prints `one` because the value in `x` is 1. This syntax is useful when you want your code to take an action if it gets a particular concrete value.

Matching Named Variables

Named variables are irrefutable patterns that match any value, and we've used them many times in the book. However, there is a complication when you use named variables in `match` expressions. Because `match` starts a new scope, variables declared as part of a pattern inside the `match` expression will shadow those with the same name outside the `match` construct, as is the case with all variables. In Listing 18-11, we declare a variable named `x` with the value `Some(5)` and a variable `y` with the value `10`. We then create a `match` expression on the value `x`. Look at the patterns in the match arms and `println!` at the end, and try to figure out what the code will print before running this code or reading further.

Filename: src/main.rs

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(y) => println!("Matched, y = {y}"),
    _ => println!("Default case, x = {x:?}"),
}

println!("at the end: x = {x:?}, y = {y}");
```

Listing 18-11: A `match` expression with an arm that introduces a shadowed variable `y`

Let's walk through what happens when the `match` expression runs. The pattern in the first match arm doesn't match the defined value of `x`, so the code continues.

The pattern in the second match arm introduces a new variable named `y` that will match any value inside a `Some` value. Because we're in a new scope inside the `match` expression, this is a new `y` variable, not the `y` we declared at the beginning with the value 10. This new `y` binding will match any value inside a `Some`, which is what we have in `x`. Therefore, this new `y` binds to the inner value of the `Some` in `x`. That value is `5`, so the expression for that arm executes and prints `Matched, y = 5`.

If `x` had been a `None` value instead of `Some(5)`, the patterns in the first two arms wouldn't have matched, so the value would have matched to the underscore. We didn't introduce the `x` variable in the pattern of the underscore arm, so the `x` in the expression is still the outer `x` that hasn't been shadowed. In this hypothetical case, the `match` would print `Default case, x = None`.

When the `match` expression is done, its scope ends, and so does the scope of the inner `y`. The last `println!` produces `at the end: x = Some(5), y = 10`.

To create a `match` expression that compares the values of the outer `x` and `y`, rather than introducing a shadowed variable, we would need to use a match guard conditional instead. We'll talk about match guards later in the "Extra Conditionals with Match Guards" section.

Multiple Patterns

In `match` expressions, you can match multiple patterns using the `|` syntax, which is the pattern *or* operator. For example, in the following code we match the value of `x` against the match arms, the first of which has an *or* option, meaning if the value of `x` matches either of the values in that arm, that arm's code will run:

```
let x = 1;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

This code prints `one or two`.

Matching Ranges of Values with `..=`

The `..=` syntax allows us to match to an inclusive range of values. In the following code, when a pattern matches any of the values within the given range, that arm will execute:

```
let x = 5;

match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}
```

If `x` is 1, 2, 3, 4, or 5, the first arm will match. This syntax is more convenient for multiple match values than using the `|` operator to express the same idea; if we were to use `|` we would have to specify `1 | 2 | 3 | 4 | 5`. Specifying a range is much shorter, especially if we want to match, say, any number between 1 and 1,000!

The compiler checks that the range isn't empty at compile time, and because the only types for which Rust can tell if a range is empty or not are `char` and numeric values, ranges are only allowed with numeric or `char` values. IMP

Here is an example using ranges of `char` values:

```
let x = 'c';

match x {
    'a'..='j' => println!("early ASCII letter"),
    'k'..='z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Rust can tell that `'c'` is within the first pattern's range and prints `early ASCII letter`.

Destructuring to Break Apart Values

We can also use patterns to destructure structs, enums, and tuples to use different parts of these values. Let's walk through each value.

Destructuring Structs

Listing 18-12 shows a `Point` struct with two fields, `x` and `y`, that we can break apart using a pattern with a `let` statement.

Filename: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x: a, y: b } = p;
    assert_eq!(0, a);
    assert_eq!(7, b);
}
```

Listing 18-12: Destructuring a struct's fields into separate variables

This code creates the variables `a` and `b` that match the values of the `x` and `y` fields of the `p` struct. This example shows that the names of the variables in the pattern don't have to match the field names of the struct. However, it's common to match the variable names to the field names to make it easier to remember which variables came from which fields. Because of this common usage, and because writing `let Point { x: x, y: y } = p;` contains a lot of duplication, Rust has a shorthand for patterns that match struct fields: you only need to list the name of the struct field, and the variables created from the pattern will have the same names. Listing 18-13 behaves in the same way as the code in Listing 18-12, but the variables created in the `let` pattern are `x` and `y` instead of `a` and `b`.

Filename: src/main.rs

```
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let p = Point { x: 0, y: 7 };

    let Point { x, y } = p;
    assert_eq!(0, x);
    assert_eq!(7, y);
}
```

Listing 18-13: Destructuring struct fields using struct field shorthand

This code creates the variables `x` and `y` that match the `x` and `y` fields of the `p` variable. The outcome is that the variables `x` and `y` contain the values from the `p` struct.

We can also destructure with literal values as part of the struct pattern rather than creating variables for all the fields. Doing so allows us to test some of the fields for particular values while creating variables to destructure the other fields.

In Listing 18-14, we have a `match` expression that separates `Point` values into three cases: points that lie directly on the `x` axis (which is true when `y = 0`), on the `y` axis (`x = 0`), or neither.

Filename: src/main.rs

```
fn main() {
    let p = Point { x: 0, y: 7 };

    match p {
        Point { x, y: 0 } => println!("On the x axis at {x}"),
        Point { x: 0, y } => println!("On the y axis at {y}"),
        Point { x, y } => {
            println!("On neither axis: ({x}, {y})");
        }
    }
}
```

Listing 18-14: Destructuring and matching literal values in one pattern

The first arm will match any point that lies on the `x` axis by specifying that the `y` field matches if its value matches the literal `0`. The pattern still creates an `x` variable that we can use in the code for this arm.

Similarly, the second arm matches any point on the `y` axis by specifying that the `x` field matches if its value is `0` and creates a variable `y` for the value of the `y` field. The third arm

doesn't specify any literals, so it matches any other `Point` and creates variables for both the `x` and `y` fields.

In this example, the value `p` matches the second arm by virtue of `x` containing a 0, so this code will print `On the y axis at 7`.

Remember that a `match` expression stops checking arms once it has found the first matching pattern, so even though `Point { x: 0, y: 0}` is on the `x` axis and the `y` axis, this code would only print `On the x axis at 0`.

Destructuring Enums

We've destructured enums in this book (for example, Listing 6-5 in Chapter 6), but haven't yet explicitly discussed that the pattern to destructure an enum corresponds to the way the data stored within the enum is defined. As an example, in Listing 18-15 we use the `Message` enum from Listing 6-2 and write a `match` with patterns that will destructure each inner value.

Filename: src/main.rs

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

fn main() {
    let msg = Message::ChangeColor(0, 160, 255);

    match msg {
        Message::Quit => {
            println!("The Quit variant has no data to destructure.");
        }
        Message::Move { x, y } => {
            println!("Move in the x direction {x} and in the y direction {y}");
        }
        Message::Write(text) => {
            println!("Text message: {text}");
        }
        Message::ChangeColor(r, g, b) => {
            println!("Change the color to red {r}, green {g}, and blue {b}")
        }
    }
}
```

Listing 18-15: Destructuring enum variants that hold different kinds of values

This code will print `Change the color to red 0, green 160, and blue 255`. Try changing the value of `msg` to see the code from the other arms run.

For enum variants without any data, like `Message::Quit`, we can't destructure the value any further. We can only match on the literal `Message::Quit` value, and no variables are in that pattern.

For struct-like enum variants, such as `Message::Move`, we can use a pattern similar to the pattern we specify to match structs. After the variant name, we place curly brackets and then list the fields with variables so we break apart the pieces to use in the code for this arm. Here we use the shorthand form as we did in Listing 18-13.

For tuple-like enum variants, like `Message::Write` that holds a tuple with one element and `Message::ChangeColor` that holds a tuple with three elements, the pattern is similar to the pattern we specify to match tuples. The number of variables in the pattern must match the number of elements in the variant we're matching.

Destructuring Nested Structs and Enums

So far, our examples have all been matching structs or enums one level deep, but matching can work on nested items too! For example, we can refactor the code in Listing 18-15 to support RGB and HSV colors in the `ChangeColor` message, as shown in Listing 18-16.

```
enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}

enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(Color),
}

fn main() {
    let msg = Message::ChangeColor(Color::Hsv(0, 160, 255));

    match msg {
        Message::ChangeColor(Color::Rgb(r, g, b)) => {
            println!("Change color to red {r}, green {g}, and blue {b}");
        }
        Message::ChangeColor(Color::Hsv(h, s, v)) => {
            println!("Change color to hue {h}, saturation {s}, value {v}")
        }
        _ => (),
    }
}
```

Listing 18-16: Matching on nested enums

The pattern of the first arm in the `match` expression matches a `Message::ChangeColor` enum variant that contains a `Color::Rgb` variant; then the pattern binds to the three inner `i32` values. The pattern of the second arm also matches a `Message::ChangeColor` enum variant, but the inner enum matches `Color::Hsv` instead. We can specify these complex conditions in one `match` expression, even though two enums are involved.

Destructuring Structs and Tuples

We can mix, match, and nest destructuring patterns in even more complex ways. The following example shows a complicated destructure where we nest structs and tuples inside a tuple and destructure all the primitive values out:

```
let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

This code lets us break complex types into their component parts so we can use the values we're interested in separately.

Destructuring with patterns is a convenient way to use pieces of values, such as the value from each field in a struct, separately from each other.

Ignoring Values in a Pattern

You've seen that it's sometimes useful to ignore values in a pattern, such as in the last arm of a `match`, to get a catchall that doesn't actually do anything but does account for all remaining possible values. There are a few ways to ignore entire values or parts of values in a pattern: using the `_` pattern (which you've seen), using the `_` pattern within another pattern, using a name that starts with an underscore, or using `..` to ignore remaining parts of a value. Let's explore how and why to use each of these patterns.

Ignoring an Entire Value with _

We've used the underscore as a wildcard pattern that will match any value but not bind to the value. This is especially useful as the last arm in a `match` expression, but we can also use it in any pattern, including function parameters, as shown in Listing 18-17.

Filename: src/main.rs

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {y}");
}

fn main() {
    foo(3, 4);
}
```

Listing 18-17: Using `_` in a function signature

This code will completely ignore the value `3` passed as the first argument, and will print `This code only uses the y parameter: 4`.

In most cases when you no longer need a particular function parameter, you would change the signature so it doesn't include the unused parameter. Ignoring a function parameter can be especially useful in cases when, for example, you're implementing a trait when you need a certain type signature but the function body in your implementation doesn't need one of the parameters. You then avoid getting a compiler warning about unused function parameters, as you would if you used a name instead.

Ignoring Parts of a Value with a Nested _

We can also use `_` inside another pattern to ignore just part of a value, for example, when we want to test for only part of a value but have no use for the other parts in the corresponding code we want to run. Listing 18-18 shows code responsible for managing a setting's value. The

business requirements are that the user should not be allowed to overwrite an existing customization of a setting but can unset the setting and give it a value if it is currently unset.

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {setting_value:?}");
```

Listing 18-18: Using an underscore within patterns that match `Some` variants when we don't need to use the value inside the `Some`

This code will print `Can't overwrite an existing customized value` and then `setting is Some(5)`. In the first match arm, we don't need to match on or use the values inside either `Some` variant, but we do need to test for the case when `setting_value` and `new_setting_value` are the `Some` variant. In that case, we print the reason for not changing `setting_value`, and it doesn't get changed.

In all other cases (if either `setting_value` or `new_setting_value` are `None`) expressed by the `_` pattern in the second arm, we want to allow `new_setting_value` to become `setting_value`.

We can also use underscores in multiple places within one pattern to ignore particular values. Listing 18-19 shows an example of ignoring the second and fourth values in a tuple of five items.

```
let numbers = (2, 4, 8, 16, 32);

match numbers {
    (first, _, third, _, fifth) => {
        println!("Some numbers: {first}, {third}, {fifth}")
    }
}
```

Listing 18-19: Ignoring multiple parts of a tuple

This code will print `Some numbers: 2, 8, 32`, and the values 4 and 16 will be ignored.

Ignoring an Unused Variable by Starting Its Name with _

If you create a variable but don't use it anywhere, Rust will usually issue a warning because an unused variable could be a bug. However, sometimes it's useful to be able to create a variable you won't use yet, such as when you're prototyping or just starting a project. In this situation, you can tell Rust not to warn you about the unused variable by starting the name of the variable with an underscore. In Listing 18-20, we create two unused variables, but when we compile this code, we should only get a warning about one of them.

Filename: src/main.rs

```
fn main() {
    let _x = 5;
    let y = 10;
}
```

Listing 18-20: Starting a variable name with an underscore to avoid getting unused variable warnings

Here we get a warning about not using the variable `y`, but we don't get a warning about not using `_x`.

Note that there is a subtle difference between using only `_` and using a name that starts with an underscore. The syntax `_x` still binds the value to the variable, whereas `_` doesn't bind at all. To show a case where this distinction matters, Listing 18-21 will provide us with an error.

```
    let s = Some(String::from("Hello!"));

    if let Some(_s) = s {
        println!("found a string");
    }

    println!("{s:?}");
```

Listing 18-21: An unused variable starting with an underscore still binds the value, which might take ownership of the value

We'll receive an error because the `s` value will still be moved into `_s`, which prevents us from using `s` again. However, using the underscore by itself doesn't ever bind to the value. Listing 18-22 will compile without any errors because `s` doesn't get moved into `_`.

```
let s = Some(String::from("Hello!"));

if let Some(_) = s {
    println!("found a string");
}

println!("{s:?}");
```

Listing 18-22: Using an underscore does not bind the value

This code works just fine because we never bind `s` to anything; it isn't moved.

Ignoring Remaining Parts of a Value with `..`

With values that have many parts, we can use the `..` syntax to use specific parts and ignore the rest, avoiding the need to list underscores for each ignored value. The `..` pattern ignores any parts of a value that we haven't explicitly matched in the rest of the pattern. In Listing 18-23, we have a `Point` struct that holds a coordinate in three-dimensional space. In the `match` expression, we want to operate only on the `x` coordinate and ignore the values in the `y` and `z` fields.

```
struct Point {
    x: i32,
    y: i32,
    z: i32,
}

let origin = Point { x: 0, y: 0, z: 0 };

match origin {
    Point { x, .. } => println!("x is {x}"),
}
```

Listing 18-23: Ignoring all fields of a `Point` except for `x` by using `..`

We list the `x` value and then just include the `..` pattern. This is quicker than having to list `y: _` and `z: _`, particularly when we're working with structs that have lots of fields in situations where only one or two fields are relevant.

The syntax `..` will expand to as many values as it needs to be. Listing 18-24 shows how to use `..` with a tuple.

Filename: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (first, .., last) => {
            println!("Some numbers: {first}, {last}");
        }
    }
}
```

Listing 18-24: Matching only the first and last values in a tuple and ignoring all other values

In this code, the first and last value are matched with `first` and `last`. The `..` will match and ignore everything in the middle.

However, using `..` must be unambiguous. If it is unclear which values are intended for matching and which should be ignored, Rust will give us an error. Listing 18-25 shows an example of using `..` ambiguously, so it will not compile.

Filename: src/main.rs

```
fn main() {
    let numbers = (2, 4, 8, 16, 32);

    match numbers {
        (.., second, ..) => {
            println!("Some numbers: {second}")
        },
    }
}
```

Listing 18-25: An attempt to use `..` in an ambiguous way

When we compile this example, we get this error:

```
$ cargo run
   Compiling patterns v0.1.0 (file:///projects/patterns)
error: `..` can only be used once per tuple pattern
 --> src/main.rs:5:22
  |
5 |         (.., second, ..) => {
  |          --          ^^ can only be used once per tuple pattern
  |          |
  |          previously used here

error: could not compile `patterns` (bin "patterns") due to 1 previous error
```

It's impossible for Rust to determine how many values in the tuple to ignore before matching a value with `second` and then how many further values to ignore thereafter. This code could mean that we want to ignore `2`, bind `second` to `4`, and then ignore `8`, `16`, and `32`; or that we want to ignore `2` and `4`, bind `second` to `8`, and then ignore `16` and `32`; and so forth. The variable name `second` doesn't mean anything special to Rust, so we get a compiler error because using `..` in two places like this is ambiguous.

Extra Conditionals with Match Guards

A *match guard* is an additional `if` condition, specified after the pattern in a `match` arm, that must also match for that arm to be chosen. Match guards are useful for expressing more complex ideas than a pattern alone allows.

The condition can use variables created in the pattern. Listing 18-26 shows a `match` where the first arm has the pattern `Some(x)` and also has a match guard of `if x % 2 == 0` (which will be true if the number is even).

```
let num = Some(4);

match num {
    Some(x) if x % 2 == 0 => println!("The number {x} is even"),
    Some(x) => println!("The number {x} is odd"),
    None => (),
}
```

Listing 18-26: Adding a match guard to a pattern

This example will print `The number 4 is even`. When `num` is compared to the pattern in the first arm, it matches, because `Some(4)` matches `Some(x)`. Then the match guard checks whether the remainder of dividing `x` by 2 is equal to 0, and because it is, the first arm is selected.

If `num` had been `Some(5)` instead, the match guard in the first arm would have been false because the remainder of 5 divided by 2 is 1, which is not equal to 0. Rust would then go to the second arm, which would match because the second arm doesn't have a match guard and therefore matches any `Some` variant.

There is no way to express the `if x % 2 == 0` condition within a pattern, so the match guard gives us the ability to express this logic. The downside of this additional expressiveness is that the compiler doesn't try to check for exhaustiveness when match guard expressions are involved.

In Listing 18-11, we mentioned that we could use match guards to solve our pattern-shadowing problem. Recall that we created a new variable inside the pattern in the `match` expression instead of using the variable outside the `match`. That new variable meant we couldn't test against the value of the outer variable. Listing 18-27 shows how we can use a match guard to fix this problem.

Filename: src/main.rs

```
fn main() {
    let x = Some(5);
    let y = 10;

    match x {
        Some(50) => println!("Got 50"),
        Some(n) if n == y => println!("Matched, n = {n}"),
        _ => println!("Default case, x = {x:?}"),
    }

    println!("at the end: x = {x:?}, y = {y}");
}
```

Listing 18-27: Using a match guard to test for equality with an outer variable

This code will now print `Default case, x = Some(5)`. The pattern in the second match arm doesn't introduce a new variable `y` that would shadow the outer `y`, meaning we can use the outer `y` in the match guard. Instead of specifying the pattern as `Some(y)`, which would have shadowed the outer `y`, we specify `Some(n)`. This creates a new variable `n` that doesn't shadow anything because there is no `n` variable outside the `match`.

The match guard `if n == y` is not a pattern and therefore doesn't introduce new variables. This `y` *is* the outer `y` rather than a new shadowed `y`, and we can look for a value that has the same value as the outer `y` by comparing `n` to `y`.

You can also use the *or* operator `|` in a match guard to specify multiple patterns; the match guard condition will apply to all the patterns. Listing 18-28 shows the precedence when combining a pattern that uses `|` with a match guard. The important part of this example is that the `if y` match guard applies to `4`, `5`, *and* `6`, even though it might look like `if y` only applies to `6`.

```
    let x = 4;
    let y = false;

    match x {
        4 | 5 | 6 if y => println!("yes"),
        _ => println!("no"),
    }
```

Listing 18-28: Combining multiple patterns with a match guard

The match condition states that the arm only matches if the value of `x` is equal to `4`, `5`, or `6` *and* if `y` is `true`. When this code runs, the pattern of the first arm matches because `x` is `4`, but the match guard `if y` is false, so the first arm is not chosen. The code moves on to the second arm, which does match, and this program prints `no`. The reason is that the `if` condition applies to the whole pattern `4 | 5 | 6`, not only to the last value `6`. In other words, the precedence of a match guard in relation to a pattern behaves like this:

```
(4 | 5 | 6) if y => ...
```

rather than this:

```
4 | 5 | (6 if y) => ...
```

After running the code, the precedence behavior is evident: if the match guard were applied only to the final value in the list of values specified using the `|` operator, the arm would have matched and the program would have printed `yes`.

@ Bindings

The *at* operator `@` lets us create a variable that holds a value at the same time as we're testing that value for a pattern match. In Listing 18-29, we want to test that a `Message::Hello id` field is within the range `3..=7`. We also want to bind the value to the variable `id_variable` so we can use it in the code associated with the arm. We could name this variable `id`, the same as the field, but for this example we'll use a different name.

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello {
        id: id_variable @ 3..=7,
    } => println!("Found an id in range: {id_variable}"),
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    }
    Message::Hello { id } => println!("Found some other id: {id}"),
}
```

Listing 18-29: Using `@` to bind to a value in a pattern while also testing it

This example will print `Found an id in range: 5`. By specifying `id_variable @` before the range `3..=7`, we're capturing whatever value matched the range while also testing that the value matched the range pattern.

In the second arm, where we only have a range specified in the pattern, the code associated with the arm doesn't have a variable that contains the actual value of the `id` field. The `id` field's value could have been 10, 11, or 12, but the code that goes with that pattern doesn't know which it is. The pattern code isn't able to use the value from the `id` field, because we haven't saved the `id` value in a variable.

In the last arm, where we've specified a variable without a range, we do have the value available to use in the arm's code in a variable named `id`. The reason is that we've used the struct field shorthand syntax. But we haven't applied any test to the value in the `id` field in this arm, as we did with the first two arms: any value would match this pattern.

Using `@` lets us test a value and save it in a variable within one pattern.

Summary

Rust's patterns are very useful in distinguishing between different kinds of data. When used in `match` expressions, Rust ensures your patterns cover every possible value, or your program won't compile. Patterns in `let` statements and function parameters make those constructs more useful, enabling the destructuring of values into smaller parts at the same time as assigning to variables. We can create simple or complex patterns to suit our needs.

Next, for the penultimate chapter of the book, we'll look at some advanced aspects of a variety of Rust's features.