Rust

Chapter 18:

Patterns and Matching

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.

Patterns

Patterns are composed of:

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

To use a pattern, we compare it to some value. If the pattern matches the value, we use the value parts in our code.

Patterns in `match`

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

`match` expressions must be exhaustive. Common practice is to have a final "match all" pattern, like `_`.

Conditional `if let` Expressions

`if let` expressions are a shorter way to write a `match` with only one case.

```rust
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, {}, as the background", color);
    } 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");
    }
}
```

`while let` Conditional Loops

`while let` conditional loops allows a while loop to run for as long as a pattern continues to match.

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

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

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

The **`while`** loop continues running the code in its block as long as **`pop`** returns **`Some`**. When **`pop`** returns **`None`**, the loop stops.

`for` Loops

In a `for` loop, the pattern is the value that directly follows the keyword `for`, so in `for x in y` the `x` is the pattern.

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

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

The first call to `enumerate` produces the tuple `(0, 'a')`. When this value is matched to the pattern `(index, value)`, `index` will be `0` and `value` will be `'a'`.

`let` Statements

So far we've only explicitly discussed patterns with control flow expressions like `match`. But `let` statements use patterns too!

```
let x = 5; // let PATTERN = EXPRESSION;
```

`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.”

Function Parameters

Function parameters can also be patterns!

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

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

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

Refutability

Whether a Pattern Might Fail to Match

Refutability

Patterns come in two forms: refutable and irrefutable.

Patterns that will match for any possible value passed are *irrefutable*.

Example: **`let x = 5;`**, because **`x`** matches anything and cannot fail to match.

Patterns that can fail to match for some possible value are *refutable*.

Example: **`if let Some(x) = a_value`**, because if **`a_value`** is **`None`**, the pattern won't match.

Irrefutable patterns

Irrefutable patterns must be passed to:

- Function parameters
- `let` statements
- `for` loops

Refutable and Irrefutable patterns

Either can be passed to:

- **`if let`** expressions
- **`while let`** expressions

Error messages

The terminology appears in compiler errors.

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

Produces:

```
error[E0005]: refutable pattern in local binding: `None` not covered
 -->
  |
3 | let Some(x) = some_option_value;
  |     ^^^^^^^ pattern `None` not covered
```

Error messages

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

Produces:

```
warning: irrefutable if-let pattern
 --> <anon>:2:5
  |
2 | /     if let x = 5 {
3 | |     println!("{}", x);
4 | | };
  | |_^
  |
  = note: #[warn(irrefutable_let_patterns)] on by default
```

Pattern Syntax

Matching literals

```
let x = 1;

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

Matching Named Variables

```rust
fn main() {
    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 = {:?}, y = {:?}", x, y);
}

// Matched, y = 5
// at the end: x = Some(5), y = 10
```

Multiple Patterns

```
let x = 1;

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

// one or two
```

Matching Ranges of Values with ..=

```rust
let x = 5;

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

// one through five
```

Destructuring Structs

```rust
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);
}
```

This code creates the variables x and y that match the x and y fields of the p variable.

Destructuring Enums

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

```rust
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 {} and in the y direction {}",
                x,
                y
            );
        }
        Message::Write(text) => println!("Text message: {}", text),
        Message::ChangeColor(r, g, b) => {
            println!(
                "Change the color to red {}, green {}, and blue {}",
                r,
                g,
                b
            )
        }
    }
}
```

Destructuring Nested Structs and Enums

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

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

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

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

Destructuring Structs and Tuples

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 });
```

Ignoring Values in a Pattern

Ignoring an Entire Value with _

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

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

This code will completely ignore the value passed as the first argument.

Ignoring Parts of a Value with a Nested _

We can also use _ inside another pattern to ignore just part of a value.

```
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);
```

Ignoring Remaining Parts of a Value with `..`

With values that have many parts, we can use the `..` syntax to use only a few parts and ignore the rest.

```
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),
}
```

The syntax `..` will expand to as many values as it needs to be.

However, using `..` must be unambiguous.

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, along with the pattern matching, for that arm to be chosen.

```
let num = Some(4);

match num {
    Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

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

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

@ Bindings

The *at* operator (`@`) lets us create a variable that holds a value at the same time we're testing that value to see whether it matches a pattern.

```rust
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) // 5
    },
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    },
    Message::Hello { id } => {
        println!("Found some other id: {}", id)
    },
}
```

Summary

Rust's patterns are very useful in that they help distinguish 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.