Defining and Instantiating Structs

Structs are similar to tuples, discussed in "The Tuple Type" section, in that both hold multiple related values. Like tuples, the pieces of a struct can be different types. Unlike with tuples, in a struct you'll name each piece of data so it's clear what the values mean. Adding these names means that structs are more flexible than tuples: you don't have to rely on the order of the data to specify or access the values of an instance.

To define a struct, we enter the keyword `struct` and name the entire struct. A struct's name should describe the significance of the pieces of data being grouped together. Then, inside curly brackets, we define the names and types of the pieces of data, which we call *fields*. For example, Listing 5-1 shows a struct that stores information about a user account.

Filename: src/main.rs

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Listing 5-1: A `User` struct definition

To use a struct after we've defined it, we create an *instance* of that struct by specifying concrete values for each of the fields. We create an instance by stating the name of the struct and then add curly brackets containing `key: value` pairs, where the keys are the names of the fields and the values are the data we want to store in those fields. We don't have to specify the fields in the same order in which we declared them in the struct. In other words, the struct definition is like a general template for the type, and instances fill in that template with particular data to create values of the type. For example, we can declare a particular user as shown in Listing 5-2.

Filename: src/main.rs

```
fn main() {
    let user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
}
```

Listing 5-2: Creating an instance of the `User` struct

To get a specific value from a struct, we use dot notation. For example, to access this user's email address, we use `user1.email`. If the instance is mutable, we can change a value by using the dot notation and assigning into a particular field. Listing 5-3 shows how to change the value in the `email` field of a mutable `User` instance.

Filename: src/main.rs

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };

    user1.email = String::from("anotheremail@example.com");
}
```

Listing 5-3: **Changing the value in the `email` field of a `User` instance**

Note that the entire instance must be mutable; Rust doesn't allow us to mark only certain fields as mutable. As with any expression, we can construct a new instance of the struct as the last expression in the function body to implicitly return that new instance.

Listing 5-4 shows a `build_user` function that returns a `User` instance with the given email and username. The `active` field gets the value of `true`, and the `sign_in_count` gets a value of `1`.

Filename: src/main.rs

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

Listing 5-4: **A `build_user` function that takes an email and username and returns a `User` instance**

It makes sense to name the function parameters with the same name as the struct fields, but having to repeat the `email` and `username` field names and variables is a bit tedious. If the struct had more fields, repeating each name would get even more annoying. Luckily, there's a convenient shorthand!

Using the Field Init Shorthand

Because the parameter names and the struct field names are exactly the same in Listing 5-4, we can use the *field init shorthand* syntax to rewrite `build_user` so it behaves exactly the same but doesn't have the repetition of `username` and `email`, as shown in Listing 5-5.

Filename: src/main.rs

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

Listing 5-5: A `build_user` function that uses field init shorthand because the `username` and `email` parameters have the same name as struct fields

Here, we're creating a new instance of the `User` struct, which has a field named `email`. We want to set the `email` field's value to the value in the `email` parameter of the `build_user` function. Because the `email` field and the `email` parameter have the same name, we only need to write `email` rather than `email: email`.

Creating Instances from Other Instances with Struct Update Syntax

It's often useful to create a new instance of a struct that includes most of the values from another instance of the same type, but changes some. You can do this using *struct update syntax*.

First, in Listing 5-6 we show how to create a new `User` instance in `user2` regularly, without the update syntax. We set a new value for `email` but otherwise use the same values from `user1` that we created in Listing 5-2.

Filename: src/main.rs

```
fn main() {
    // --snip--

    let user2 = User {
        active: user1.active,
        username: user1.username,
        email: String::from("another@example.com"),
        sign_in_count: user1.sign_in_count,
    };
}
```

Listing 5-6: Creating a new `User` instance using all but one of the values from `user1`

Using struct update syntax, we can achieve the same effect with less code, as shown in Listing 5-7. The syntax `..` specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

Filename: src/main.rs

```
fn main() {
    // --snip--

    let user2 = User {
        email: String::from("another@example.com"),
        ..user1
    };
}
```

Listing 5-7: Using struct update syntax to set a new `email` value for a `User` instance but to use the rest of the values from `user1`

The code in Listing 5-7 also creates an instance in `user2` that has a different value for `email` but has the same values for the `username`, `active`, and `sign_in_count` fields from `user1`. The `..user1` must come last to specify that any remaining fields should get their values from the corresponding fields in `user1`, but we can choose to specify values for as many fields as we want in any order, regardless of the order of the fields in the struct's definition.

Note that the struct update syntax uses `=` like an assignment; this is because it moves the data, just as we saw in the "Variables and Data Interacting with Move" section. In this example, we can no longer use `user1` after creating `user2` because the `String` in the `username` field of `user1` was moved into `user2`. If we had given `user2` new `String` values for both `email` and `username`, and thus only used the `active` and `sign_in_count` values from `user1`, then `user1` would still be valid after creating `user2`. Both `active` and `sign_in_count` are types that implement the `Copy` trait, so the behavior we discussed in the "Stack-Only Data: Copy" section would apply. We can also still use `user1.email` in this example, because its value was not moved out of `user1`.

Using Tuple Structs Without Named Fields to Create Different Types

Rust also supports structs that look similar to tuples, called *tuple structs*. Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they just have the types of the fields. Tuple structs are useful when you want to give the whole tuple a name and make the tuple a different type from other tuples, and when naming each field as in a regular struct would be verbose or redundant.

To define a tuple struct, start with the `struct` keyword and the struct name followed by the types in the tuple. For example, here we define and use two tuple structs named `Color` and `Point`:

Filename: src/main.rs

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Note that the `black` and `origin` values are different types because they're instances of different tuple structs. Each struct you define is its own type, even though the fields within the struct might have the same types. For example, a function that takes a parameter of type `Color` cannot take a `Point` as an argument, even though both types are made up of three `i32` values. Otherwise, tuple struct instances are similar to tuples in that you can destructure them into their individual pieces, and you can use a `.` followed by the index to access an individual value. Unlike tuples, tuple structs require you to name the type of the struct when you destructure them. For example, we would write `let Point(x, y, z) = origin;` to destructure the values in the `origin` point into variables named `x`, `y`, and `z`.

Unit-Like Structs Without Any Fields

You can also define structs that don't have any fields! These are called *unit-like structs* because they behave similarly to `()`, the unit type that we mentioned in "The Tuple Type" section. Unit-like structs can be useful when you need to implement a trait on some type but don't have any data that you want to store in the type itself. We'll discuss traits in Chapter 10. Here's an example of declaring and instantiating a unit struct named `AlwaysEqual`:

Filename: src/main.rs

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual;
}
```

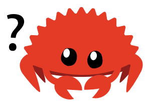
To define `AlwaysEqual`, we use the `struct` keyword, the name we want, and then a semicolon. No need for curly brackets or parentheses! Then we can get an instance of `AlwaysEqual` in the `subject` variable in a similar way: using the name we defined, without any curly brackets or parentheses. Imagine that later we'll implement behavior for this type such that every instance of `AlwaysEqual` is always equal to every instance of any other type, perhaps to have a known result for testing purposes. We wouldn't need any data to implement that behavior! You'll see in Chapter 10 how to define traits and implement them on any type, including unit-like structs.

Ownership of Struct Data

In the `User` struct definition in Listing 5-1, we used the owned `String` type rather than the `&str` string slice type. This is a deliberate choice because we want each instance of this struct to own all of its data and for that data to be valid for as long as the entire struct is valid.

It's also possible for structs to store references to data owned by something else, but to do so requires the use of *lifetimes*, a Rust feature that we'll discuss in Chapter 10. Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is. Let's say you try to store a reference in a struct without specifying lifetimes, like the following; this won't work:

Filename: src/main.rs



```rust
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        active: true,
        username: "someusername123",
        email: "someone@example.com",
        sign_in_count: 1,
    };
}
```

The compiler will complain that it needs lifetime specifiers:

```
$ cargo run
   Compiling structs v0.1.0 (file:///projects/structs)
error[E0106]: missing lifetime specifier
 --> src/main.rs:3:15
  |
3 |     username: &str,
  |               ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 ~     username: &'a str,
  |

error[E0106]: missing lifetime specifier
 --> src/main.rs:4:12
  |
4 |     email: &str,
  |            ^ expected named lifetime parameter
  |
help: consider introducing a named lifetime parameter
  |
1 ~ struct User<'a> {
2 |     active: bool,
3 |     username: &str,
4 ~     email: &'a str,
  |

For more information about this error, try `rustc --explain E0106`.
error: could not compile `structs` (bin "structs") due to 2 previous
errors
```

In Chapter 10, we'll discuss how to fix these errors so you can store references in structs, but for now, we'll fix errors like these using owned types like `String` instead of references like `&str`.

An Example Program Using Structs

To understand when we might want to use structs, let's write a program that calculates the area of a rectangle. We'll start by using single variables, and then refactor the program until we're using structs instead.

Let's make a new binary project with Cargo called *rectangles* that will take the width and height of a rectangle specified in pixels and calculate the area of the rectangle. Listing 5-8 shows a short program with one way of doing exactly that in our project's *src/main.rs*.

Filename: src/main.rs

```rust
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!(
        "The area of the rectangle is {} square pixels.",
        area(width1, height1)
    );
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

Listing 5-8: **Calculating the area of a rectangle specified by separate width and height variables**

Now, run this program using `cargo run`:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.42s
     Running `target/debug/rectangles`
The area of the rectangle is 1500 square pixels.
```

This code succeeds in figuring out the area of the rectangle by calling the `area` function with each dimension, but we can do more to make this code clear and readable.

The issue with this code is evident in the signature of `area`:

```rust
fn area(width: u32, height: u32) -> u32 {
```

The `area` function is supposed to calculate the area of one rectangle, but the function we wrote has two parameters, and it's not clear anywhere in our program that the parameters are related. It would be more readable and more manageable to group width and height together. We've already discussed one way we might do that in "The Tuple Type" section of Chapter 3: by using tuples.

Refactoring with Tuples

Listing 5-9 shows another version of our program that uses tuples.

Filename: src/main.rs

```
fn main() {
    let rect1 = (30, 50);

    println!(
        "The area of the rectangle is {} square pixels.",
        area(rect1)
    );
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

Listing 5-9: Specifying the width and height of the rectangle with a tuple

In one way, this program is better. Tuples let us add a bit of structure, and we're now passing just one argument. But in another way, this version is less clear: tuples don't name their elements, so we have to index into the parts of the tuple, making our calculation less obvious.

Mixing up the width and height wouldn't matter for the area calculation, but if we want to draw the rectangle on the screen, it would matter! We would have to keep in mind that `width` is the tuple index `0` and `height` is the tuple index `1`. This would be even harder for someone else to figure out and keep in mind if they were to use our code. Because we haven't conveyed the meaning of our data in our code, it's now easier to introduce errors.

Refactoring with Structs: Adding More Meaning

We use structs to add meaning by labeling the data. We can transform the tuple we're using into a struct with a name for the whole as well as names for the parts, as shown in Listing 5-10.

Filename: src/main.rs

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        area(&rect1)
    );
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.width * rectangle.height
}
```

Listing 5-10: Defining a `Rectangle` struct

Here, we've defined a struct and named it `Rectangle`. Inside the curly brackets, we defined the fields as `width` and `height`, both of which have type `u32`. Then, in `main`, we created a particular instance of `Rectangle` that has a width of `30` and a height of `50`.

Our `area` function is now defined with one parameter, which we've named `rectangle`, whose type is an immutable borrow of a struct `Rectangle` instance. As mentioned in Chapter 4, we want to borrow the struct rather than take ownership of it. This way, `main` retains its ownership and can continue using `rect1`, which is the reason we use the `&` in the function signature and where we call the function.

The `area` function accesses the `width` and `height` fields of the `Rectangle` instance (note that accessing fields of a borrowed struct instance does not move the field values, which is why you often see borrows of structs). Our function signature for `area` now says exactly what we mean: calculate the area of `Rectangle`, using its `width` and `height` fields. This conveys that the width and height are related to each other, and it gives descriptive names to the values rather than using the tuple index values of `0` and `1`. This is a win for clarity.

Adding Useful Functionality with Derived Traits

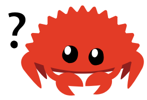
It'd be useful to be able to print an instance of `Rectangle` while we're debugging our program and see the values for all its fields. Listing 5-11 tries using the `println!` macro as we have used in previous chapters. This won't work, however.

Filename: src/main.rs

```
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {rect1}");
}
```



Listing 5-11: Attempting to print a `Rectangle` instance

When we compile this code, we get an error with this core message:

```
error[E0277]: `Rectangle` doesn't implement `std::fmt::Display`
```

The `println!` macro can do many kinds of formatting, and by default, the curly brackets tell `println!` to use formatting known as `Display`: output intended for direct end user consumption. The primitive types we've seen so far implement `Display` by default because there's only one way you'd want to show a `1` or any other primitive type to a user. But with structs, the way `println!` should format the output is less clear because there are more display possibilities: Do you want commas or not? Do you want to print the curly brackets? Should all the fields be shown? Due to this ambiguity, Rust doesn't try to guess what we want, and structs don't have a provided implementation of `Display` to use with `println!` and the `{}` placeholder.

If we continue reading the errors, we'll find this helpful note:

```
   = help: the trait `std::fmt::Display` is not implemented for `Rectangle`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for
pretty-print) instead
```

Let's try it! The `println!` macro call will now look like `println!("rect1 is {rect1:?}");`. Putting the specifier `:?` inside the curly brackets tells `println!` we want to use an output format called `Debug`. The `Debug` trait enables us to print our struct in a way that is useful for developers so we can see its value while we're debugging our code.

Compile the code with this change. Drat! We still get an error:

```
error[E0277]: `Rectangle` doesn't implement `Debug`
```

But again, the compiler gives us a helpful note:

```
  = help: the trait `Debug` is not implemented for `Rectangle`
  = note: add `#[derive(Debug)]` to `Rectangle` or manually `impl Debug for
Rectangle`
```

Rust *does* include functionality to print out debugging information, but we have to explicitly opt in to make that functionality available for our struct. To do that, we add the outer attribute `#[derive(Debug)]` just before the struct definition, as shown in Listing 5-12.

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!("rect1 is {rect1:?}");
}
```

Listing 5-12: Adding the attribute to derive the `Debug` trait and printing the `Rectangle` instance using debug formatting

Now when we run the program, we won't get any errors, and we'll see the following output:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle { width: 30, height: 50 }
```

Nice! It's not the prettiest output, but it shows the values of all the fields for this instance, which would definitely help during debugging. When we have larger structs, it's useful to have output that's a bit easier to read; in those cases, we can use `{:#?}` instead of `{:?}` in the `println!` string. In this example, using the `{:#?}` style will output the following:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/rectangles`
rect1 is Rectangle {
    width: 30,
    height: 50,
}
```

Another way to print out a value using the `Debug` format is to use the `dbg!` macro, which takes ownership of an expression (as opposed to `println!`, which takes a reference), prints the file and line number of where that `dbg!` macro call occurs in your code along with the resultant value of that expression, and returns ownership of the value.

> Note: Calling the `dbg!` macro prints to the standard error console stream (`stderr`), as opposed to `println!`, which prints to the standard output console stream (`stdout`). We'll talk more about `stderr` and `stdout` in the "Writing Error Messages to Standard Error Instead of Standard Output" section in Chapter 12.

Here's an example where we're interested in the value that gets assigned to the `width` field, as well as the value of the whole struct in `rect1`:

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

fn main() {
    let scale = 2;
    let rect1 = Rectangle {
        width: dbg!(30 * scale),
        height: 50,
    };

    dbg!(&rect1);
}
```

We can put `dbg!` around the expression `30 * scale` and, because `dbg!` returns ownership of the expression's value, the `width` field will get the same value as if we didn't have the `dbg!` call there. We don't want `dbg!` to take ownership of `rect1`, so we use a reference to `rect1` in the next call. Here's what the output of this example looks like:

```
$ cargo run
   Compiling rectangles v0.1.0 (file:///projects/rectangles)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.61s
     Running `target/debug/rectangles`
[src/main.rs:10:16] 30 * scale = 60
[src/main.rs:14:5] &rect1 = Rectangle {
    width: 60,
    height: 50,
}
```

We can see the first bit of output came from *src/main.rs* line 10 where we're debugging the expression `30 * scale`, and its resultant value is `60` (the `Debug` formatting implemented for integers is to print only their value). The `dbg!` call on line 14 of *src/main.rs* outputs the value of `&rect1`, which is the `Rectangle` struct. This output uses the pretty `Debug` formatting of the `Rectangle` type. The `dbg!` macro can be really helpful when you're trying to figure out what your code is doing!

In addition to the `Debug` trait, Rust has provided a number of traits for us to use with the `derive` attribute that can add useful behavior to our custom types. Those traits and their behaviors are listed in Appendix C. We'll cover how to implement these traits with custom behavior as well as how to create your own traits in Chapter 10. There are also many attributes other than `derive`; for more information, see the "Attributes" section of the Rust Reference.

Our `area` function is very specific: it only computes the area of rectangles. It would be helpful to tie this behavior more closely to our `Rectangle` struct because it won't work with any other type. Let's look at how we can continue to refactor this code by turning the `area` function into an `area` *method* defined on our `Rectangle` type.

Method Syntax

Methods are similar to functions: we declare them with the `fn` keyword and a name, they can have parameters and a return value, and they contain some code that's run when the method is called from somewhere else. Unlike functions, methods are defined within the context of a struct (or an enum or a trait object, which we cover in Chapter 6 and Chapter 18, respectively), and their first parameter is always `self`, which represents the instance of the struct the method is being called on.

Defining Methods

Let's change the `area` function that has a `Rectangle` instance as a parameter and instead make an `area` method defined on the `Rectangle` struct, as shown in Listing 5-13.

Filename: src/main.rs

```
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Listing 5-13: Defining an `area` method on the `Rectangle` struct

To define the function within the context of `Rectangle`, we start an `impl` (implementation) block for `Rectangle`. Everything within this `impl` block will be associated with the `Rectangle` type. Then we move the `area` function within the `impl` curly brackets and change the first (and in this case, only) parameter to be `self` in the signature and everywhere within the body. In `main`, where we called the `area` function and passed `rect1` as an argument, we can instead use *method syntax* to call the `area` method on our `Rectangle` instance. The method syntax goes after an instance: we add a dot followed by the method name, parentheses, and any arguments.

In the signature for `area`, we use `&self` instead of `rectangle: &Rectangle`. The `&self` is actually short for `self: &Self`. Within an `impl` block, the type `Self` is an alias for the type that the `impl` block is for. Methods must have a parameter named `self` of type `Self` for their first parameter, so Rust lets you abbreviate this with only the name `self` in the first parameter spot. Note that we still need to use the `&` in front of the `self` shorthand to indicate that this method borrows the `Self` instance, just as we did in `rectangle: &Rectangle`. Methods can take ownership of `self`, borrow `self` immutably, as we've done here, or borrow `self` mutably, just as they can any other parameter.

We chose `&self` here for the same reason we used `&Rectangle` in the function version: we don't want to take ownership, and we just want to read the data in the struct, not write to it. If we wanted to change the instance that we've called the method on as part of what the method does, we'd use `&mut self` as the first parameter. Having a method that takes ownership of the instance by using just `self` as the first parameter is rare; this technique is usually used when the method transforms `self` into something else and you want to prevent the caller from using the original instance after the transformation.

The main reason for using methods instead of functions, in addition to providing method syntax and not having to repeat the type of `self` in every method's signature, is for organization. We've put all the things we can do with an instance of a type in one `impl` block rather than making future users of our code search for capabilities of `Rectangle` in various places in the library we provide.

Note that we can choose to give a method the same name as one of the struct's fields. For example, we can define a method on `Rectangle` that is also named `width`:

Filename: src/main.rs

```rust
impl Rectangle {
    fn width(&self) -> bool {
        self.width > 0
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    if rect1.width() {
        println!("The rectangle has a nonzero width; it is {}", rect1.width);
    }
}
```

Here, we're choosing to make the `width` method return `true` if the value in the instance's `width` field is greater than `0` and `false` if the value is `0` : we can use a field within a method of the same name for any purpose. In `main` , when we follow `rect1.width` with parentheses, Rust knows we mean the method `width` . When we don't use parentheses, Rust knows we mean the field `width` .

Often, but not always, when we give a method the same name as a field we want it to only return the value in the field and do nothing else. Methods like this are called *getters*, and Rust does not implement them automatically for struct fields as some other languages do. Getters are useful because you can make the field private but the method public, and thus enable read-only access to that field as part of the type's public API. We will discuss what public and private are and how to designate a field or method as public or private in Chapter 7.

Where's the -> Operator?

In C and C++, two different operators are used for calling methods: you use `.` if you're calling a method on the object directly and `->` if you're calling the method on a pointer to the object and need to dereference the pointer first. In other words, if `object` is a pointer, `object->something()` is similar to `(*object).something()` .

Rust doesn't have an equivalent to the `->` operator; instead, Rust has a feature called *automatic referencing and dereferencing*. Calling methods is one of the few places in Rust with this behavior.

Here's how it works: when you call a method with `object.something()` , Rust automatically adds in `&` , `&mut` , or `*` so `object` matches the signature of the method. In other words, the following are the same:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of `self`. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading (`&self`), mutating (`&mut self`), or consuming (`self`). The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.

Methods with More Parameters

Let's practice using methods by implementing a second method on the `Rectangle` struct. This time we want an instance of `Rectangle` to take another instance of `Rectangle` and return `true` if the second `Rectangle` can fit completely within `self` (the first `Rectangle`); otherwise, it should return `false`. That is, once we've defined the `can_hold` method, we want to be able to write the program shown in Listing 5-14.

Filename: src/main.rs

```rust
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };

    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

Listing 5-14: Using the as-yet-unwritten `can_hold` method

The expected output would look like the following because both dimensions of `rect2` are smaller than the dimensions of `rect1`, but `rect3` is wider than `rect1` :

```
Can rect1 hold rect2? true
Can rect1 hold rect3? false
```

We know we want to define a method, so it will be within the `impl Rectangle` block. The method name will be `can_hold`, and it will take an immutable borrow of another

`Rectangle` as a parameter. We can tell what the type of the parameter will be by looking at the code that calls the method: `rect1.can_hold(&rect2)` passes in `&rect2`, which is an immutable borrow to `rect2`, an instance of `Rectangle`. This makes sense because we only need to read `rect2` (rather than write, which would mean we'd need a mutable borrow), and we want `main` to retain ownership of `rect2` so we can use it again after calling the `can_hold` method. The return value of `can_hold` will be a Boolean, and the implementation will check whether the width and height of `self` are greater than the width and height of the other `Rectangle`, respectively. Let's add the new `can_hold` method to the `impl` block from Listing 5-13, shown in Listing 5-15.

Filename: src/main.rs

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }

    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Listing 5-15: Implementing the `can_hold` method on `Rectangle` that takes another `Rectangle` instance as a parameter

When we run this code with the `main` function in Listing 5-14, we'll get our desired output. Methods can take multiple parameters that we add to the signature after the `self` parameter, and those parameters work just like parameters in functions.

Associated Functions

All functions defined within an `impl` block are called *associated functions* because they're associated with the type named after the `impl`. We can define associated functions that don't have `self` as their first parameter (and thus are not methods) because they don't need an instance of the type to work with. We've already used one function like this: the `String::from` function that's defined on the `String` type.

Associated functions that aren't methods are often used for constructors that will return a new instance of the struct. These are often called `new`, but `new` isn't a special name and isn't built into the language. For example, we could choose to provide an associated function named `square` that would have one dimension parameter and use that as both width and height, thus making it easier to create a square `Rectangle` rather than having to specify the same value twice:

Filename: src/main.rs

```rust
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

The `Self` keywords in the return type and in the body of the function are aliases for the type that appears after the `impl` keyword, which in this case is `Rectangle`.

To call this associated function, we use the `::` syntax with the struct name; `let sq = Rectangle::square(3);` is an example. This function is namespaced by the struct: the `::` syntax is used for both associated functions and namespaces created by modules. We'll discuss modules in Chapter 7.

Multiple `impl` Blocks

Each struct is allowed to have multiple `impl` blocks. For example, Listing 5-15 is equivalent to the code shown in Listing 5-16, which has each method in its own `impl` block.

```rust
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

impl Rectangle {
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
```

Listing 5-16: Rewriting Listing 5-15 using multiple `impl` blocks

There's no reason to separate these methods into multiple `impl` blocks here, but this is valid syntax. We'll see a case in which multiple `impl` blocks are useful in Chapter 10, where we discuss generic types and traits.

Summary

Structs let you create custom types that are meaningful for your domain. By using structs, you can keep associated pieces of data connected to each other and name each piece to make your code clear. In `impl` blocks, you can define functions that are associated with your type, and methods are a kind of associated function that let you specify the behavior that instances of your structs have.

But structs aren't the only way you can create custom types: let's turn to Rust's enum feature to add another tool to your toolbox.