Advanced Features

By now, you've learned the most commonly used parts of the Rust programming language. Before we do one more project in Chapter 20, we'll look at a few aspects of the language you might run into every once in a while, but may not use every day. You can use this chapter as a reference for when you encounter any unknowns. The features covered here are useful in very specific situations. Although you might not reach for them often, we want to make sure you have a grasp of all the features Rust has to offer.

In this chapter, we'll cover:

- Unsafe Rust: how to opt out of some of Rust's guarantees and take responsibility for manually upholding those guarantees
- Advanced traits: associated types, default type parameters, fully qualified syntax, supertraits, and the newtype pattern in relation to traits
- Advanced types: more about the newtype pattern, type aliases, the never type, and dynamically sized types
- Advanced functions and closures: function pointers and returning closures
- Macros: ways to define code that defines more code at compile time

It's a panoply of Rust features with something for everyone! Let's dive in!

Unsafe Rust

All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time. However, Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees: it's called *unsafe Rust* and works just like regular Rust, but gives us extra superpowers.

Unsafe Rust exists because, by nature, static analysis is conservative. When the compiler tries to determine whether or not code upholds the guarantees, it's better for it to reject some valid programs than to accept some invalid programs. Although the code *might* be okay, if the Rust compiler doesn't have enough information to be confident, it will reject the code. In these cases, you can use unsafe code to tell the compiler, "Trust me, I know what I'm doing." Be warned, however, that you use unsafe Rust at your own risk: if you use unsafe code incorrectly, problems can occur due to memory unsafety, such as null pointer dereferencing.

Another reason Rust has an unsafe alter ego is that the underlying computer hardware is inherently unsafe. If Rust didn't let you do unsafe operations, you couldn't do certain tasks. Rust needs to allow you to do low-level systems programming, such as directly interacting with the operating system or even writing your own operating system. Working with low-level systems programming is one of the goals of the language. Let's explore what we can do with unsafe Rust and how to do it.

Unsafe Superpowers

To switch to unsafe Rust, use the `unsafe` keyword and then start a new block that holds the unsafe code. You can take five actions in unsafe Rust that you can't in safe Rust, which we call *unsafe superpowers*. Those superpowers include the ability to:

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of a `union`

It's important to understand that `unsafe` doesn't turn off the borrow checker or disable any other of Rust's safety checks: if you use a reference in unsafe code, it will still be checked. The `unsafe` keyword only gives you access to these five features that are then not checked by the compiler for memory safety. You'll still get some degree of safety inside of an unsafe block.

In addition, `unsafe` does not mean the code inside the block is necessarily dangerous or that it will definitely have memory safety problems: the intent is that as the programmer, you'll ensure the code inside an `unsafe` block will access memory in a valid way.

People are fallible, and mistakes will happen, but by requiring these five unsafe operations to be inside blocks annotated with `unsafe` you'll know that any errors related to memory safety must be within an `unsafe` block. Keep `unsafe` blocks small; you'll be thankful later when you investigate memory bugs.

To isolate unsafe code as much as possible, it's best to enclose unsafe code within a safe abstraction and provide a safe API, which we'll discuss later in the chapter when we examine unsafe functions and methods. Parts of the standard library are implemented as safe abstractions over unsafe code that has been audited. Wrapping unsafe code in a safe abstraction prevents uses of `unsafe` from leaking out into all the places that you or your users might want to use the functionality implemented with `unsafe` code, because using a safe abstraction is safe.

Let's look at each of the five unsafe superpowers in turn. We'll also look at some abstractions that provide a safe interface to unsafe code.

Dereferencing a Raw Pointer

In Chapter 4, in the "Dangling References" section, we mentioned that the compiler ensures references are always valid. Unsafe Rust has two new types called *raw pointers* that are similar to references. As with references, raw pointers can be immutable or mutable and are written as `*const T` and `*mut T`, respectively. The asterisk isn't the dereference operator; it's part of the type name. In the context of raw pointers, *immutable* means that the pointer can't be directly assigned to after being dereferenced.

Different from references and smart pointers, raw pointers:

- Are allowed to ignore the borrowing rules by having both immutable and mutable pointers or multiple mutable pointers to the same location
- Aren't guaranteed to point to valid memory
- Are allowed to be null
- Don't implement any automatic cleanup

By opting out of having Rust enforce these guarantees, you can give up guaranteed safety in exchange for greater performance or the ability to interface with another language or hardware where Rust's guarantees don't apply.

Listing 19-1 shows how to create an immutable and a mutable raw pointer from references.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;
```

Listing 19-1: Creating raw pointers from references

Notice that we don't include the `unsafe` keyword in this code. We can create raw pointers in safe code; we just can't dereference raw pointers outside an unsafe block, as you'll see in a bit.

We've created raw pointers by using `as` to cast an immutable and a mutable reference into their corresponding raw pointer types. Because we created them directly from references guaranteed to be valid, we know these particular raw pointers are valid, but we can't make that assumption about just any raw pointer.

To demonstrate this, next we'll create a raw pointer whose validity we can't be so certain of. Listing 19-2 shows how to create a raw pointer to an arbitrary location in memory. Trying to use arbitrary memory is undefined: there might be data at that address or there might not, the compiler might optimize the code so there is no memory access, or the program might error with a segmentation fault. Usually, there is no good reason to write code like this, but it is possible.

```
let address = 0x012345usize;
let r = address as *const i32;
```

Listing 19-2: Creating a raw pointer to an arbitrary memory address

Recall that we can create raw pointers in safe code, but we can't *dereference* raw pointers and read the data being pointed to. In Listing 19-3, we use the dereference operator `*` on a raw pointer that requires an `unsafe` block.

```
let mut num = 5;

let r1 = &num as *const i32;
let r2 = &mut num as *mut i32;

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

Listing 19-3: Dereferencing raw pointers within an `unsafe` block

Creating a pointer does no harm; it's only when we try to access the value that it points at that we might end up dealing with an invalid value.

Note also that in Listing 19-1 and 19-3, we created `*const i32` and `*mut i32` raw pointers that both pointed to the same memory location, where `num` is stored. If we instead tried to create an immutable and a mutable reference to `num`, the code would not have compiled because Rust's ownership rules don't allow a mutable reference at the same time as any immutable references. With raw pointers, we can create a mutable pointer and an immutable pointer to the same location and change data through the mutable pointer, potentially creating a data race. Be careful!

With all of these dangers, why would you ever use raw pointers? One major use case is when interfacing with C code, as you'll see in the next section, "Calling an Unsafe Function or Method." Another case is when building up safe abstractions that the borrow checker doesn't understand. We'll introduce unsafe functions and then look at an example of a safe abstraction that uses unsafe code.

Calling an Unsafe Function or Method

The second type of operation you can perform in an unsafe block is calling unsafe functions. Unsafe functions and methods look exactly like regular functions and methods, but they have an extra `unsafe` before the rest of the definition. The `unsafe` keyword in this context indicates the function has requirements we need to uphold when we call this function, because Rust can't guarantee we've met these requirements. By calling an unsafe function within an `unsafe` block, we're saying that we've read this function's documentation and take responsibility for upholding the function's contracts.

Here is an unsafe function named `dangerous` that doesn't do anything in its body:

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

We must call the `dangerous` function within a separate `unsafe` block. If we try to call `dangerous` without the `unsafe` block, we'll get an error:

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0133]: call to unsafe function `dangerous` is unsafe and requires unsafe
function or block
 --> src/main.rs:4:5
  |
4 |     dangerous();
  |     ^^^^^^^^^^^ call to unsafe function
  |
  = note: consult the function's documentation for information on how to avoid
undefined behavior

For more information about this error, try `rustc --explain E0133`.
error: could not compile `unsafe-example` (bin "unsafe-example") due to 1 previous
error
```

With the `unsafe` block, we're asserting to Rust that we've read the function's documentation, we understand how to use it properly, and we've verified that we're fulfilling the contract of the function.

Bodies of unsafe functions are effectively `unsafe` blocks, so to perform other unsafe operations within an unsafe function, we don't need to add another `unsafe` block.

Creating a Safe Abstraction over Unsafe Code

Just because a function contains unsafe code doesn't mean we need to mark the entire function as unsafe. In fact, wrapping unsafe code in a safe function is a common abstraction. As an example, let's study the `split_at_mut` function from the standard library, which requires some unsafe code. We'll explore how we might implement it. This safe method is defined on mutable slices: it takes one slice and makes it two by splitting the slice at the index given as an argument. Listing 19-4 shows how to use `split_at_mut`.

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

Listing 19-4: Using the safe `split_at_mut` function

We can't implement this function using only safe Rust. An attempt might look something like Listing 19-5, which won't compile. For simplicity, we'll implement `split_at_mut` as a function rather than a method and only for slices of `i32` values rather than for a generic type `T`.

```
fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();

    assert!(mid <= len);

    (&mut values[..mid], &mut values[mid..])
}
```

Listing 19-5: An attempted implementation of `split_at_mut` using only safe Rust

This function first gets the total length of the slice. Then it asserts that the index given as a parameter is within the slice by checking whether it's less than or equal to the length. The assertion means that if we pass an index that is greater than the length to split the slice at, the function will panic before it attempts to use that index.

Then we return two mutable slices in a tuple: one from the start of the original slice to the `mid` index and another from `mid` to the end of the slice.

When we try to compile the code in Listing 19-5, we'll get an error.

```
$ cargo run
   Compiling unsafe-example v0.1.0 (file:///projects/unsafe-example)
error[E0499]: cannot borrow `*values` as mutable more than once at a time
 --> src/main.rs:6:31
  |
1 | fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
  |                         - let's call the lifetime of this reference `'1`
...
6 |     (&mut values[..mid], &mut values[mid..])
  |     -------------------------^^^^^^--------
  |     |     |                  |
  |     |     |                  second mutable borrow occurs here
  |     |     first mutable borrow occurs here
  |     returning this value requires that `*values` is borrowed for `'1`

For more information about this error, try `rustc --explain E0499`.
error: could not compile `unsafe-example` (bin "unsafe-example") due to 1 previous
error
```

Rust's borrow checker can't understand that we're borrowing different parts of the slice; it only knows that we're borrowing from the same slice twice. Borrowing different parts of a slice is fundamentally okay because the two slices aren't overlapping, but Rust isn't smart enough to know this. When we know code is okay, but Rust doesn't, it's time to reach for unsafe code.

Listing 19-6 shows how to use an `unsafe` block, a raw pointer, and some calls to unsafe functions to make the implementation of `split_at_mut` work.

```
use std::slice;

fn split_at_mut(values: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = values.len();
    let ptr = values.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (
            slice::from_raw_parts_mut(ptr, mid),
            slice::from_raw_parts_mut(ptr.add(mid), len - mid),
        )
    }
}
```

Listing 19-6: Using unsafe code in the implementation of the `split_at_mut` function

Recall from "The Slice Type" section in Chapter 4 that slices are a pointer to some data and the length of the slice. We use the `len` method to get the length of a slice and the `as_mut_ptr` method to access the raw pointer of a slice. In this case, because we have a mutable slice to `i32` values, `as_mut_ptr` returns a raw pointer with the type `*mut i32`, which we've stored in the variable `ptr`.

We keep the assertion that the `mid` index is within the slice. Then we get to the unsafe code: the `slice::from_raw_parts_mut` function takes a raw pointer and a length, and it creates a slice. We use this function to create a slice that starts from `ptr` and is `mid` items long. Then we call the `add` method on `ptr` with `mid` as an argument to get a raw pointer that starts at `mid`, and we create a slice using that pointer and the remaining number of items after `mid` as the length.

The function `slice::from_raw_parts_mut` is unsafe because it takes a raw pointer and must trust that this pointer is valid. The `add` method on raw pointers is also unsafe, because it must trust that the offset location is also a valid pointer. Therefore, we had to put an `unsafe` block around our calls to `slice::from_raw_parts_mut` and `add` so we could call them. By looking at the code and by adding the assertion that `mid` must be less than or equal to `len`, we can tell that all the raw pointers used within the `unsafe` block will be valid pointers to data within the slice. This is an acceptable and appropriate use of `unsafe`.

Note that we don't need to mark the resulting `split_at_mut` function as `unsafe`, and we can call this function from safe Rust. We've created a safe abstraction to the unsafe code with an implementation of the function that uses `unsafe` code in a safe way, because it creates only valid pointers from the data this function has access to.

In contrast, the use of `slice::from_raw_parts_mut` in Listing 19-7 would likely crash when the slice is used. This code takes an arbitrary memory location and creates a slice 10,000 items long.

```
use std::slice;

let address = 0x01234usize;
let r = address as *mut i32;

let values: &[i32] = unsafe { slice::from_raw_parts_mut(r, 10000) };
```

Listing 19-7: Creating a slice from an arbitrary memory location

We don't own the memory at this arbitrary location, and there is no guarantee that the slice this code creates contains valid `i32` values. Attempting to use `values` as though it's a valid slice results in undefined behavior.

Using `extern` Functions to Call External Code

Sometimes, your Rust code might need to interact with code written in another language. For this, Rust has the keyword `extern` that facilitates the creation and use of a *Foreign Function Interface (FFI)*. An FFI is a way for a programming language to define functions and enable a different (foreign) programming language to call those functions.

Listing 19-8 demonstrates how to set up an integration with the `abs` function from the C standard library. Functions declared within `extern` blocks are always unsafe to call from Rust code. The reason is that other languages don't enforce Rust's rules and guarantees, and Rust can't check them, so responsibility falls on the programmer to ensure safety.

Filename: src/main.rs

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe {
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}
```

Listing 19-8: Declaring and calling an `extern` function defined in another language

Within the `extern "C"` block, we list the names and signatures of external functions from another language we want to call. The `"C"` part defines which *application binary interface (ABI)*

the external function uses: the ABI defines how to call the function at the assembly level. The `"C"` ABI is the most common and follows the C programming language's ABI.

Calling Rust Functions from Other Languages

We can also use `extern` to create an interface that allows other languages to call Rust functions. Instead of creating a whole `extern` block, we add the `extern` keyword and specify the ABI to use just before the `fn` keyword for the relevant function. We also need to add a `#[no_mangle]` annotation to tell the Rust compiler not to mangle the name of this function. *Mangling* is when a compiler changes the name we've given a function to a different name that contains more information for other parts of the compilation process to consume but is less human readable. Every programming language compiler mangles names slightly differently, so for a Rust function to be nameable by other languages, we must disable the Rust compiler's name mangling.

In the following example, we make the `call_from_c` function accessible from C code, after it's compiled to a shared library and linked from C:

```
#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
}
```

This usage of `extern` does not require `unsafe`.

Accessing or Modifying a Mutable Static Variable

In this book, we've not yet talked about *global variables*, which Rust does support but can be problematic with Rust's ownership rules. If two threads are accessing the same mutable global variable, it can cause a data race.

In Rust, global variables are called *static* variables. Listing 19-9 shows an example declaration and use of a static variable with a string slice as a value.

Filename: src/main.rs

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {HELLO_WORLD}");
}
```

Listing 19-9: Defining and using an immutable static variable

Static variables are similar to constants, which we discussed in the "Differences Between Variables and Constants" section in Chapter 3. The names of static variables are in `SCREAMING_SNAKE_CASE` by convention. Static variables can only store references with the `'static` lifetime, which means the Rust compiler can figure out the lifetime and we aren't required to annotate it explicitly. Accessing an immutable static variable is safe.

A subtle difference between constants and immutable static variables is that values in a static variable have a fixed address in memory. Using the value will always access the same data. Constants, on the other hand, are allowed to duplicate their data whenever they're used. Another difference is that static variables can be mutable. Accessing and modifying mutable static variables is *unsafe*. Listing 19-10 shows how to declare, access, and modify a mutable static variable named `COUNTER`.

Filename: src/main.rs

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {COUNTER}");
    }
}
```

Listing 19-10: Reading from or writing to a mutable static variable is unsafe

As with regular variables, we specify mutability using the `mut` keyword. Any code that reads or writes from `COUNTER` must be within an `unsafe` block. This code compiles and prints `COUNTER: 3` as we would expect because it's single threaded. Having multiple threads access `COUNTER` would likely result in data races.

With mutable data that is globally accessible, it's difficult to ensure there are no data races, which is why Rust considers mutable static variables to be unsafe. Where possible, it's preferable to use the concurrency techniques and thread-safe smart pointers we discussed in Chapter 16 so the compiler checks that data accessed from different threads is done safely.

Implementing an Unsafe Trait

We can use `unsafe` to implement an unsafe trait. A trait is unsafe when at least one of its methods has some invariant that the compiler can't verify. We declare that a trait is `unsafe` by adding the `unsafe` keyword before `trait` and marking the implementation of the trait as `unsafe` too, as shown in Listing 19-11.

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}

fn main() {}
```

Listing 19-11: Defining and implementing an unsafe trait

By using `unsafe impl`, we're promising that we'll uphold the invariants that the compiler can't verify.

As an example, recall the `Sync` and `Send` marker traits we discussed in the "Extensible Concurrency with the `Sync` and `Send` Traits" section in Chapter 16: the compiler implements these traits automatically if our types are composed entirely of `Send` and `Sync` types. If we implement a type that contains a type that is not `Send` or `Sync`, such as raw pointers, and we want to mark that type as `Send` or `Sync`, we must use `unsafe`. Rust can't verify that our type upholds the guarantees that it can be safely sent across threads or accessed from multiple threads; therefore, we need to do those checks manually and indicate as such with `unsafe`.

Accessing Fields of a Union

The final action that works only with `unsafe` is accessing fields of a *union*. A `union` is similar to a `struct`, but only one declared field is used in a particular instance at one time. Unions are primarily used to interface with unions in C code. Accessing union fields is unsafe because Rust can't guarantee the type of the data currently being stored in the union instance. You can learn more about unions in the Rust Reference.

When to Use Unsafe Code

Using `unsafe` to take one of the five actions (superpowers) just discussed isn't wrong or even frowned upon. But it is trickier to get `unsafe` code correct because the compiler can't help

uphold memory safety. When you have a reason to use `unsafe` code, you can do so, and having the explicit `unsafe` annotation makes it easier to track down the source of problems when they occur.

Advanced Traits

We first covered traits in the "Traits: Defining Shared Behavior" section of Chapter 10, but we didn't discuss the more advanced details. Now that you know more about Rust, we can get into the nitty-gritty.

Specifying Placeholder Types in Trait Definitions with Associated Types

Associated types connect a type placeholder with a trait such that the trait method definitions can use these placeholder types in their signatures. The implementor of a trait will specify the concrete type to be used instead of the placeholder type for the particular implementation. That way, we can define a trait that uses some types without needing to know exactly what those types are until the trait is implemented.

We've described most of the advanced features in this chapter as being rarely needed. Associated types are somewhere in the middle: they're used more rarely than features explained in the rest of the book but more commonly than many of the other features discussed in this chapter.

One example of a trait with an associated type is the `Iterator` trait that the standard library provides. The associated type is named `Item` and stands in for the type of the values the type implementing the `Iterator` trait is iterating over. The definition of the `Iterator` trait is as shown in Listing 19-12.

```
pub trait Iterator {
    type Item;

    fn next(&mut self) -> Option<Self::Item>;
}
```

Listing 19-12: The definition of the `Iterator` trait that has an associated type `Item`

The type `Item` is a placeholder, and the `next` method's definition shows that it will return values of type `Option<Self::Item>`. Implementors of the `Iterator` trait will specify the concrete type for `Item`, and the `next` method will return an `Option` containing a value of that concrete type.

Associated types might seem like a similar concept to generics, in that the latter allow us to define a function without specifying what types it can handle. To examine the difference

between the two concepts, we'll look at an implementation of the `Iterator` trait on a type named `Counter` that specifies the `Item` type is `u32`:

Filename: src/lib.rs

```
impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // --snip--
```

This syntax seems comparable to that of generics. So why not just define the `Iterator` trait with generics, as shown in Listing 19-13?

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}
```

Listing 19-13: A hypothetical definition of the `Iterator` trait using generics

The difference is that when using generics, as in Listing 19-13, we must annotate the types in each implementation; because we can also implement `Iterator<String> for Counter` or any other type, we could have multiple implementations of `Iterator` for `Counter`. In other words, when a trait has a generic parameter, it can be implemented for a type multiple times, changing the concrete types of the generic type parameters each time. When we use the `next` method on `Counter`, we would have to provide type annotations to indicate which implementation of `Iterator` we want to use.

With associated types, we don't need to annotate types because we can't implement a trait on a type multiple times. In Listing 19-12 with the definition that uses associated types, we can only choose what the type of `Item` will be once, because there can only be one `impl Iterator for Counter`. We don't have to specify that we want an iterator of `u32` values everywhere that we call `next` on `Counter`.

Associated types also become part of the trait's contract: implementors of the trait must provide a type to stand in for the associated type placeholder. Associated types often have a name that describes how the type will be used, and documenting the associated type in the API documentation is good practice.

Default Generic Type Parameters and Operator Overloading

When we use generic type parameters, we can specify a default concrete type for the generic type. This eliminates the need for implementors of the trait to specify a concrete type if the

default type works. You specify a default type when declaring a generic type with the `<PlaceholderType=ConcreteType>` syntax.

A great example of a situation where this technique is useful is with *operator overloading*, in which you customize the behavior of an operator (such as `+`) in particular situations.

Rust doesn't allow you to create your own operators or overload arbitrary operators. But you can overload the operations and corresponding traits listed in `std::ops` by implementing the traits associated with the operator. For example, in Listing 19-14 we overload the `+` operator to add two `Point` instances together. We do this by implementing the `Add` trait on a `Point` struct:

Filename: src/main.rs

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point {
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(
        Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
        Point { x: 3, y: 3 }
    );
}
```

Listing 19-14: Implementing the `Add` trait to overload the `+` operator for `Point` instances

The `add` method adds the `x` values of two `Point` instances and the `y` values of two `Point` instances to create a new `Point`. The `Add` trait has an associated type named `Output` that determines the type returned from the `add` method.

The default generic type in this code is within the `Add` trait. Here is its definition:

```
trait Add<Rhs=Self> {
    type Output;

    fn add(self, rhs: Rhs) -> Self::Output;
}
```

This code should look generally familiar: a trait with one method and an associated type. The new part is `Rhs=Self`: this syntax is called *default type parameters*. The `Rhs` generic type parameter (short for "right hand side") defines the type of the `rhs` parameter in the `add` method. If we don't specify a concrete type for `Rhs` when we implement the `Add` trait, the type of `Rhs` will default to `Self`, which will be the type we're implementing `Add` on.

When we implemented `Add` for `Point`, we used the default for `Rhs` because we wanted to add two `Point` instances. Let's look at an example of implementing the `Add` trait where we want to customize the `Rhs` type rather than using the default.

We have two structs, `Millimeters` and `Meters`, holding values in different units. This thin wrapping of an existing type in another struct is known as the *newtype pattern*, which we describe in more detail in the "Using the Newtype Pattern to Implement External Traits on External Types" section. We want to add values in millimeters to values in meters and have the implementation of `Add` do the conversion correctly. We can implement `Add` for `Millimeters` with `Meters` as the `Rhs`, as shown in Listing 19-15.

Filename: src/lib.rs

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
}
```

Listing 19-15: Implementing the `Add` trait on `Millimeters` to add `Millimeters` to `Meters`

To add `Millimeters` and `Meters`, we specify `impl Add<Meters>` to set the value of the `Rhs` type parameter instead of using the default of `Self`.

You'll use default type parameters in two main ways:

- To extend a type without breaking existing code

- To allow customization in specific cases most users won't need

The standard library's `Add` trait is an example of the second purpose: usually, you'll add two like types, but the `Add` trait provides the ability to customize beyond that. Using a default type parameter in the `Add` trait definition means you don't have to specify the extra parameter most of the time. In other words, a bit of implementation boilerplate isn't needed, making it easier to use the trait.

The first purpose is similar to the second but in reverse: if you want to add a type parameter to an existing trait, you can give it a default to allow extension of the functionality of the trait without breaking the existing implementation code.

Fully Qualified Syntax for Disambiguation: Calling Methods with the Same Name

Nothing in Rust prevents a trait from having a method with the same name as another trait's method, nor does Rust prevent you from implementing both traits on one type. It's also possible to implement a method directly on the type with the same name as methods from traits.

When calling methods with the same name, you'll need to tell Rust which one you want to use. Consider the code in Listing 19-16 where we've defined two traits, `Pilot` and `Wizard`, that both have a method called `fly`. We then implement both traits on a type `Human` that already has a method named `fly` implemented on it. Each `fly` method does something different.

Filename: src/main.rs

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

Listing 19-16: Two traits are defined to have a `fly` method and are implemented on the `Human` type, and a `fly` method is implemented on `Human` directly

When we call `fly` on an instance of `Human`, the compiler defaults to calling the method that is directly implemented on the type, as shown in Listing 19-17.

Filename: src/main.rs

```
fn main() {
    let person = Human;
    person.fly();
}
```

Listing 19-17: Calling `fly` on an instance of `Human`

Running this code will print `*waving arms furiously*`, showing that Rust called the `fly` method implemented on `Human` directly.

To call the `fly` methods from either the `Pilot` trait or the `Wizard` trait, we need to use more explicit syntax to specify which `fly` method we mean. Listing 19-18 demonstrates this syntax.

Filename: src/main.rs

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

Listing 19-18: Specifying which trait's `fly` method we want to call

Specifying the trait name before the method name clarifies to Rust which implementation of `fly` we want to call. We could also write `Human::fly(&person)`, which is equivalent to the `person.fly()` that we used in Listing 19-18, but this is a bit longer to write if we don't need to disambiguate.

Running this code prints the following:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.46s
     Running `target/debug/traits-example`
This is your captain speaking.
Up!
*waving arms furiously*
```

Because the `fly` method takes a `self` parameter, if we had two *types* that both implement one *trait*, Rust could figure out which implementation of a trait to use based on the type of `self`.

However, associated functions that are not methods don't have a `self` parameter. When there are multiple types or traits that define non-method functions with the same function name, Rust doesn't always know which type you mean unless you use *fully qualified syntax*. For example, in Listing 19-19 we create a trait for an animal shelter that wants to name all baby dogs *Spot*. We make an `Animal` trait with an associated non-method function `baby_name`. The `Animal` trait is implemented for the struct `Dog`, on which we also provide an associated non-method function `baby_name` directly.

Filename: src/main.rs

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
}
```

Listing 19-19: A trait with an associated function and a type with an associated function of the same name that also implements the trait

We implement the code for naming all puppies Spot in the `baby_name` associated function that is defined on `Dog`. The `Dog` type also implements the trait `Animal`, which describes characteristics that all animals have. Baby dogs are called puppies, and that is expressed in the implementation of the `Animal` trait on `Dog` in the `baby_name` function associated with the `Animal` trait.

In `main`, we call the `Dog::baby_name` function, which calls the associated function defined on `Dog` directly. This code prints the following:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.54s
     Running `target/debug/traits-example`
A baby dog is called a Spot
```

This output isn't what we wanted. We want to call the `baby_name` function that is part of the `Animal` trait that we implemented on `Dog` so the code prints `A baby dog is called a puppy`. The technique of specifying the trait name that we used in Listing 19-18 doesn't help here; if we change `main` to the code in Listing 19-20, we'll get a compilation error.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
}
```

Listing 19-20: Attempting to call the `baby_name` function from the `Animal` trait, but Rust doesn't know which implementation to use

Because `Animal::baby_name` doesn't have a `self` parameter, and there could be other types that implement the `Animal` trait, Rust can't figure out which implementation of `Animal::baby_name` we want. We'll get this compiler error:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0790]: cannot call associated function on trait without specifying the
corresponding `impl` type
  --> src/main.rs:20:43
   |
2  |     fn baby_name() -> String;
   |     ------------------------- `Animal::baby_name` defined here
...
20 |     println!("A baby dog is called a {}", Animal::baby_name());
   |                                           ^^^^^^^^^^^^^^^^^^ cannot call
associated function of trait
   |
help: use the fully-qualified path to the only available implementation
   |
20 |     println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
   |                                           +++++++       +

For more information about this error, try `rustc --explain E0790`.
error: could not compile `traits-example` (bin "traits-example") due to 1 previous
error
```

To disambiguate and tell Rust that we want to use the implementation of `Animal` for `Dog` as opposed to the implementation of `Animal` for some other type, we need to use fully qualified syntax. Listing 19-21 demonstrates how to use fully qualified syntax.

Filename: src/main.rs

```
fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Listing 19-21: Using fully qualified syntax to specify that we want to call the `baby_name` function from the `Animal` trait as implemented on `Dog`

We're providing Rust with a type annotation within the angle brackets, which indicates we want to call the `baby_name` method from the `Animal` trait as implemented on `Dog` by saying that we want to treat the `Dog` type as an `Animal` for this function call. This code will now print what we want:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
    Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.48s
     Running `target/debug/traits-example`
A baby dog is called a puppy
```

In general, fully qualified syntax is defined as follows:

```
<Type as Trait>::function(receiver_if_method, next_arg, ...);
```

For associated functions that aren't methods, there would not be a `receiver`: there would only be the list of other arguments. You could use fully qualified syntax everywhere that you call functions or methods. However, you're allowed to omit any part of this syntax that Rust can figure out from other information in the program. You only need to use this more verbose syntax in cases where there are multiple implementations that use the same name and Rust needs help to identify which implementation you want to call.

Using Supertraits to Require One Trait's Functionality Within Another Trait

Sometimes, you might write a trait definition that depends on another trait: for a type to implement the first trait, you want to require that type to also implement the second trait. You would do this so that your trait definition can make use of the associated items of the second trait. The trait your trait definition is relying on is called a *supertrait* of your trait.

For example, let's say we want to make an `OutlinePrint` trait with an `outline_print` method that will print a given value formatted so that it's framed in asterisks. That is, given a `Point` struct that implements the standard library trait `Display` to result in `(x, y)`, when we call `outline_print` on a `Point` instance that has `1` for `x` and `3` for `y`, it should print the following:

```
**********
*        *
* (1, 3) *
*        *
**********
```

In the implementation of the `outline_print` method, we want to use the `Display` trait's functionality. Therefore, we need to specify that the `OutlinePrint` trait will work only for types

that also implement `Display` and provide the functionality that `OutlinePrint` needs. We can do that in the trait definition by specifying `OutlinePrint: Display`. This technique is similar to adding a trait bound to the trait. Listing 19-22 shows an implementation of the `OutlinePrint` trait.

Filename: src/main.rs

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {output} *");
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}
```

Listing 19-22: Implementing the `OutlinePrint` trait that requires the functionality from `Display`

Because we've specified that `OutlinePrint` requires the `Display` trait, we can use the `to_string` function that is automatically implemented for any type that implements `Display`. If we tried to use `to_string` without adding a colon and specifying the `Display` trait after the trait name, we'd get an error saying that no method named `to_string` was found for the type `&Self` in the current scope.

Let's see what happens when we try to implement `OutlinePrint` on a type that doesn't implement `Display`, such as the `Point` struct:

Filename: src/main.rs

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

impl OutlinePrint for Point {}
```

We get an error saying that `Display` is required but not implemented:

```
$ cargo run
   Compiling traits-example v0.1.0 (file:///projects/traits-example)
error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src/main.rs:20:23
   |
20 | impl OutlinePrint for Point {}
   |                       ^^^^^ `Point` cannot be formatted with the default
formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
note: required by a bound in `OutlinePrint`
  --> src/main.rs:3:21
   |
3  | trait OutlinePrint: fmt::Display {
   |                     ^^^^^^^^^^^^ required by this bound in `OutlinePrint`

error[E0277]: `Point` doesn't implement `std::fmt::Display`
  --> src/main.rs:24:7
   |
24 |     p.outline_print();
   |       ^^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
   = note: in format strings you may be able to use `{:?}` (or {:#?} for pretty-
print) instead
note: required by a bound in `OutlinePrint::outline_print`
  --> src/main.rs:3:21
   |
3  | trait OutlinePrint: fmt::Display {
   |                     ^^^^^^^^^^^^ required by this bound in
`OutlinePrint::outline_print`
4  |     fn outline_print(&self) {
   |        ------------- required by a bound in this associated function

For more information about this error, try `rustc --explain E0277`.
error: could not compile `traits-example` (bin "traits-example") due to 2 previous
errors
```

To fix this, we implement `Display` on `Point` and satisfy the constraint that `OutlinePrint` requires, like so:

Filename: src/main.rs

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

Then implementing the `OutlinePrint` trait on `Point` will compile successfully, and we can call `outline_print` on a `Point` instance to display it within an outline of asterisks.

Using the Newtype Pattern to Implement External Traits on External Types

In Chapter 10 in the "Implementing a Trait on a Type" section, we mentioned the orphan rule that states we're only allowed to implement a trait on a type if either the trait or the type are local to our crate. It's possible to get around this restriction using the *newtype pattern*, which involves creating a new type in a tuple struct. (We covered tuple structs in the "Using Tuple Structs without Named Fields to Create Different Types" section of Chapter 5.) The tuple struct will have one field and be a thin wrapper around the type we want to implement a trait for. Then the wrapper type is local to our crate, and we can implement the trait on the wrapper. *Newtype* is a term that originates from the Haskell programming language. There is no runtime performance penalty for using this pattern, and the wrapper type is elided at compile time.

As an example, let's say we want to implement `Display` on `Vec<T>`, which the orphan rule prevents us from doing directly because the `Display` trait and the `Vec<T>` type are defined outside our crate. We can make a `Wrapper` struct that holds an instance of `Vec<T>`; then we can implement `Display` on `Wrapper` and use the `Vec<T>` value, as shown in Listing 19-23.

Filename: src/main.rs

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {w}");
}
```

Listing 19-23: Creating a `Wrapper` type around `Vec<String>` to implement `Display`

The implementation of `Display` uses `self.0` to access the inner `Vec<T>`, because `Wrapper` is a tuple struct and `Vec<T>` is the item at index 0 in the tuple. Then we can use the functionality of the `Display` trait on `Wrapper`.

The downside of using this technique is that `Wrapper` is a new type, so it doesn't have the methods of the value it's holding. We would have to implement all the methods of `Vec<T>` directly on `Wrapper` such that the methods delegate to `self.0`, which would allow us to treat `Wrapper` exactly like a `Vec<T>`. If we wanted the new type to have every method the inner type has, implementing the `Deref` trait (discussed in Chapter 15 in the "Treating Smart Pointers Like Regular References with the `Deref` Trait" section) on the `Wrapper` to return the inner type would be a solution. If we don't want the `Wrapper` type to have all the methods of the inner type—for example, to restrict the `Wrapper` type's behavior—we would have to implement just the methods we do want manually.

This newtype pattern is also useful even when traits are not involved. Let's switch focus and look at some advanced ways to interact with Rust's type system.

Advanced Types

The Rust type system has some features that we've so far mentioned but haven't yet discussed. We'll start by discussing newtypes in general as we examine why newtypes are useful as types. Then we'll move on to type aliases, a feature similar to newtypes but with slightly different semantics. We'll also discuss the `!` type and dynamically sized types.

Using the Newtype Pattern for Type Safety and Abstraction

> Note: This section assumes you've read the earlier section "Using the Newtype Pattern to Implement External Traits on External Types."

The newtype pattern is also useful for tasks beyond those we've discussed so far, including statically enforcing that values are never confused and indicating the units of a value. You saw an example of using newtypes to indicate units in Listing 19-15: recall that the `Millimeters` and `Meters` structs wrapped `u32` values in a newtype. If we wrote a function with a parameter of type `Millimeters`, we couldn't compile a program that accidentally tried to call that function with a value of type `Meters` or a plain `u32`.

We can also use the newtype pattern to abstract away some implementation details of a type: the new type can expose a public API that is different from the API of the private inner type.

Newtypes can also hide internal implementation. For example, we could provide a `People` type to wrap a `HashMap<i32, String>` that stores a person's ID associated with their name. Code using `People` would only interact with the public API we provide, such as a method to add a name string to the `People` collection; that code wouldn't need to know that we assign an `i32` ID to names internally. The newtype pattern is a lightweight way to achieve encapsulation to hide implementation details, which we discussed in the "Encapsulation that Hides Implementation Details" section of Chapter 17.

Creating Type Synonyms with Type Aliases

Rust provides the ability to declare a *type alias* to give an existing type another name. For this we use the `type` keyword. For example, we can create the alias `Kilometers` to `i32` like so:

```
type Kilometers = i32;
```

Now, the alias `Kilometers` is a *synonym* for `i32`; unlike the `Millimeters` and `Meters` types we created in Listing 19-15, `Kilometers` is not a separate, new type. Values that have the type `Kilometers` will be treated the same as values of type `i32`:

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y);
```

Because `Kilometers` and `i32` are the same type, we can add values of both types and we can pass `Kilometers` values to functions that take `i32` parameters. However, using this method, we don't get the type checking benefits that we get from the newtype pattern discussed earlier. In other words, if we mix up `Kilometers` and `i32` values somewhere, the compiler will not give us an error.

The main use case for type synonyms is to reduce repetition. For example, we might have a lengthy type like this:

```
Box<dyn Fn() + Send + 'static>
```

Writing this lengthy type in function signatures and as type annotations all over the code can be tiresome and error prone. Imagine having a project full of code like that in Listing 19-24.

```
let f: Box<dyn Fn() + Send + 'static> = Box::new(|| println!("hi"));

fn takes_long_type(f: Box<dyn Fn() + Send + 'static>) {
    // --snip--
}

fn returns_long_type() -> Box<dyn Fn() + Send + 'static> {
    // --snip--
}
```

Listing 19-24: Using a long type in many places

A type alias makes this code more manageable by reducing the repetition. In Listing 19-25, we've introduced an alias named `Thunk` for the verbose type and can replace all uses of the type with the shorter alias `Thunk`.

```
type Thunk = Box<dyn Fn() + Send + 'static>;

let f: Thunk = Box::new(|| println!("hi"));

fn takes_long_type(f: Thunk) {
    // --snip--
}

fn returns_long_type() -> Thunk {
    // --snip--
}
```

Listing 19-25: Introducing a type alias `Thunk` to reduce repetition

This code is much easier to read and write! Choosing a meaningful name for a type alias can help communicate your intent as well (*thunk* is a word for code to be evaluated at a later time, so it's an appropriate name for a closure that gets stored).

Type aliases are also commonly used with the `Result<T, E>` type for reducing repetition. Consider the `std::io` module in the standard library. I/O operations often return a `Result<T, E>` to handle situations when operations fail to work. This library has a `std::io::Error` struct that represents all possible I/O errors. Many of the functions in `std::io` will be returning `Result<T, E>` where the `E` is `std::io::Error`, such as these functions in the `Write` trait:

```
use std::fmt;
use std::io::Error;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}
```

The `Result<..., Error>` is repeated a lot. As such, `std::io` has this type alias declaration:

```
type Result<T> = std::result::Result<T, std::io::Error>;
```

Because this declaration is in the `std::io` module, we can use the fully qualified alias `std::io::Result<T>`; that is, a `Result<T, E>` with the `E` filled in as `std::io::Error`. The `Write` trait function signatures end up looking like this:

```
pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<()>;
}
```

The type alias helps in two ways: it makes code easier to write *and* it gives us a consistent interface across all of `std::io`. Because it's an alias, it's just another `Result<T, E>`, which means we can use any methods that work on `Result<T, E>` with it, as well as special syntax like the `?` operator.

The Never Type that Never Returns

Rust has a special type named `!` that's known in type theory lingo as the *empty type* because it has no values. We prefer to call it the *never type* because it stands in the place of the return type when a function will never return. Here is an example:

```
fn bar() -> ! {
    // --snip--
}
```

This code is read as "the function `bar` returns never." Functions that return never are called *diverging functions*. We can't create values of the type `!` so `bar` can never possibly return.

But what use is a type you can never create values for? Recall the code from Listing 2-5, part of the number guessing game; we've reproduced a bit of it here in Listing 19-26.

```
        let guess: u32 = match guess.trim().parse() {
            Ok(num) => num,
            Err(_) => continue,
        };
```

Listing 19-26: A `match` with an arm that ends in `continue`

At the time, we skipped over some details in this code. In Chapter 6 in "The `match` Control Flow Operator" section, we discussed that `match` arms must all return the same type. So, for example, the following code doesn't work:

```
    let guess = match guess.trim().parse() {
        Ok(_) => 5,
        Err(_) => "hello",
    };
```

The type of `guess` in this code would have to be an integer *and* a string, and Rust requires that `guess` have only one type. So what does `continue` return? How were we allowed to return a `u32` from one arm and have another arm that ends with `continue` in Listing 19-26?

As you might have guessed, `continue` has a `!` value. That is, when Rust computes the type of `guess`, it looks at both match arms, the former with a value of `u32` and the latter with a `!` value. Because `!` can never have a value, Rust decides that the type of `guess` is `u32`.

The formal way of describing this behavior is that expressions of type `!` can be coerced into any other type. We're allowed to end this `match` arm with `continue` because `continue` doesn't return a value; instead, it moves control back to the top of the loop, so in the `Err` case, we never assign a value to `guess`.

The never type is useful with the `panic!` macro as well. Recall the `unwrap` function that we call on `Option<T>` values to produce a value or panic with this definition:

```
impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}
```

In this code, the same thing happens as in the `match` in Listing 19-26: Rust sees that `val` has the type `T` and `panic!` has the type `!`, so the result of the overall `match` expression is `T`. This code works because `panic!` doesn't produce a value; it ends the program. In the `None` case, we won't be returning a value from `unwrap`, so this code is valid.

One final expression that has the type `!` is a `loop`:

```
    print!("forever ");

    loop {
        print!("and ever ");
    }
```

Here, the loop never ends, so `!` is the value of the expression. However, this wouldn't be true if we included a `break`, because the loop would terminate when it got to the `break`.

Dynamically Sized Types and the `Sized` Trait

Rust needs to know certain details about its types, such as how much space to allocate for a value of a particular type. This leaves one corner of its type system a little confusing at first: the concept of *dynamically sized types*. Sometimes referred to as *DSTs* or *unsized types*, these types let us write code using values whose size we can know only at runtime.

Let's dig into the details of a dynamically sized type called `str`, which we've been using throughout the book. That's right, not `&str`, but `str` on its own, is a DST. We can't know how long the string is until runtime, meaning we can't create a variable of type `str`, nor can we take an argument of type `str`. Consider the following code, which does not work:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
```

Rust needs to know how much memory to allocate for any value of a particular type, and all values of a type must use the same amount of memory. If Rust allowed us to write this code, these two `str` values would need to take up the same amount of space. But they have different lengths: `s1` needs 12 bytes of storage and `s2` needs 15. This is why it's not possible to create a variable holding a dynamically sized type.

So what do we do? In this case, you already know the answer: we make the types of `s1` and `s2` a `&str` rather than a `str`. Recall from the "String Slices" section of Chapter 4 that the slice data structure just stores the starting position and the length of the slice. So although a `&T` is a single value that stores the memory address of where the `T` is located, a `&str` is *two* values: the address of the `str` and its length. As such, we can know the size of a `&str` value at compile time: it's twice the length of a `usize`. That is, we always know the size of a `&str`, no matter how long the string it refers to is. In general, this is the way in which dynamically sized types are used in Rust: they have an extra bit of metadata that stores the size of the dynamic information. The golden rule of dynamically sized types is that we must always put values of dynamically sized types behind a pointer of some kind.

We can combine `str` with all kinds of pointers: for example, `Box<str>` or `Rc<str>`. In fact, you've seen this before but with a different dynamically sized type: traits. Every trait is a dynamically sized type we can refer to by using the name of the trait. In Chapter 17 in the "Using Trait Objects That Allow for Values of Different Types" section, we mentioned that to use traits as trait objects, we must put them behind a pointer, such as `&dyn Trait` or `Box<dyn Trait>` (`Rc<dyn Trait>` would work too).

To work with DSTs, Rust provides the `Sized` trait to determine whether or not a type's size is known at compile time. This trait is automatically implemented for everything whose size is known at compile time. In addition, Rust implicitly adds a bound on `Sized` to every generic function. That is, a generic function definition like this:

```
fn generic<T>(t: T) {
    // --snip--
}
```

is actually treated as though we had written this:

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

By default, generic functions will work only on types that have a known size at compile time. However, you can use the following special syntax to relax this restriction:

```
fn generic<T: ?Sized>(t: &T) {
    // --snip--
}
```

A trait bound on `?Sized` means "`T` may or may not be `Sized`" and this notation overrides the default that generic types must have a known size at compile time. The `?Trait` syntax with this meaning is only available for `Sized`, not any other traits.

Also note that we switched the type of the `t` parameter from `T` to `&T`. Because the type might not be `Sized`, we need to use it behind some kind of pointer. In this case, we've chosen a reference.

Next, we'll talk about functions and closures!

Advanced Functions and Closures

This section explores some advanced features related to functions and closures, including function pointers and returning closures.

Function Pointers

We've talked about how to pass closures to functions; you can also pass regular functions to functions! This technique is useful when you want to pass a function you've already defined rather than defining a new closure. Functions coerce to the type `fn` (with a lowercase f), not to be confused with the `Fn` closure trait. The `fn` type is called a *function pointer*. Passing functions with function pointers will allow you to use functions as arguments to other functions.

The syntax for specifying that a parameter is a function pointer is similar to that of closures, as shown in Listing 19-27, where we've defined a function `add_one` that adds one to its parameter. The function `do_twice` takes two parameters: a function pointer to any function that takes an `i32` parameter and returns an `i32`, and one `i32` value. The `do_twice` function calls the function `f` twice, passing it the `arg` value, then adds the two function call results together. The `main` function calls `do_twice` with the arguments `add_one` and `5`.

Filename: src/main.rs

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {answer}");
}
```

Listing 19-27: Using the `fn` type to accept a function pointer as an argument

This code prints `The answer is: 12`. We specify that the parameter `f` in `do_twice` is an `fn` that takes one parameter of type `i32` and returns an `i32`. We can then call `f` in the body of

`do_twice`. In `main`, we can pass the function name `add_one` as the first argument to `do_twice`.

Unlike closures, `fn` is a type rather than a trait, so we specify `fn` as the parameter type directly rather than declaring a generic type parameter with one of the `Fn` traits as a trait bound.

Function pointers implement all three of the closure traits (`Fn`, `FnMut`, and `FnOnce`), meaning you can always pass a function pointer as an argument for a function that expects a closure. It's best to write functions using a generic type and one of the closure traits so your functions can accept either functions or closures.

That said, one example of where you would want to only accept `fn` and not closures is when interfacing with external code that doesn't have closures: C functions can accept functions as arguments, but C doesn't have closures.

As an example of where you could use either a closure defined inline or a named function, let's look at a use of the `map` method provided by the `Iterator` trait in the standard library. To use the `map` function to turn a vector of numbers into a vector of strings, we could use a closure, like this:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
    list_of_numbers.iter().map(|i| i.to_string()).collect();
```

Or we could name a function as the argument to `map` instead of the closure, like this:

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> =
    list_of_numbers.iter().map(ToString::to_string).collect();
```

Note that we must use the fully qualified syntax that we talked about earlier in the "Advanced Traits" section because there are multiple functions available named `to_string`. Here, we're using the `to_string` function defined in the `ToString` trait, which the standard library has implemented for any type that implements `Display`.

Recall from the "Enum values" section of Chapter 6 that the name of each enum variant that we define also becomes an initializer function. We can use these initializer functions as function pointers that implement the closure traits, which means we can specify the initializer functions as arguments for methods that take closures, like so:

```
enum Status {
    Value(u32),
    Stop,
}

let list_of_statuses: Vec<Status> = (0u32..20).map(Status::Value).collect();
```

Here we create `Status::Value` instances using each `u32` value in the range that `map` is called on by using the initializer function of `Status::Value`. Some people prefer this style, and some people prefer to use closures. They compile to the same code, so use whichever style is clearer to you.

Returning Closures

Closures are represented by traits, which means you can't return closures directly. In most cases where you might want to return a trait, you can instead use the concrete type that implements the trait as the return value of the function. However, you can't do that with closures because they don't have a concrete type that is returnable; you're not allowed to use the function pointer `fn` as a return type, for example.

The following code tries to return a closure directly, but it won't compile:

```
fn returns_closure() -> dyn Fn(i32) -> i32 {
    |x| x + 1
}
```

The compiler error is as follows:

```
$ cargo build
   Compiling functions-example v0.1.0 (file:///projects/functions-example)
error[E0746]: return type cannot have an unboxed trait object
 --> src/lib.rs:1:25
  |
1 | fn returns_closure() -> dyn Fn(i32) -> i32 {
  |                         ^^^^^^^^^^^^^^^^^^ doesn't have a size known at
compile-time
  |
help: return an `impl Trait` instead of a `dyn Trait`, if all returned values are
the same type
  |
1 | fn returns_closure() -> impl Fn(i32) -> i32 {
  |                         ~~~~
help: box the return type, and wrap all of the returned values in `Box::new`
  |
1 ~ fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
2 ~     Box::new(|x| x + 1)
  |

For more information about this error, try `rustc --explain E0746`.
error: could not compile `functions-example` (lib) due to 1 previous error
```

The error references the `Sized` trait again! Rust doesn't know how much space it will need to store the closure. We saw a solution to this problem earlier. We can use a trait object:

```
fn returns_closure() -> Box<dyn Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

This code will compile just fine. For more about trait objects, refer to the section "Using Trait Objects That Allow for Values of Different Types" in Chapter 17.

Next, let's look at macros!

Macros

We've used macros like `println!` throughout this book, but we haven't fully explored what a macro is and how it works. The term *macro* refers to a family of features in Rust: *declarative* macros with `macro_rules!` and three kinds of *procedural* macros:

- Custom `#[derive]` macros that specify code added with the `derive` attribute used on structs and enums
- Attribute-like macros that define custom attributes usable on any item
- Function-like macros that look like function calls but operate on the tokens specified as their argument

We'll talk about each of these in turn, but first, let's look at why we even need macros when we already have functions.

The Difference Between Macros and Functions

Fundamentally, macros are a way of writing code that writes other code, which is known as *metaprogramming*. In Appendix C, we discuss the `derive` attribute, which generates an implementation of various traits for you. We've also used the `println!` and `vec!` macros throughout the book. All of these macros *expand* to produce more code than the code you've written manually.

Metaprogramming is useful for reducing the amount of code you have to write and maintain, which is also one of the roles of functions. However, macros have some additional powers that functions don't.

A function signature must declare the number and type of parameters the function has. Macros, on the other hand, can take a variable number of parameters: we can call `println!("hello")` with one argument or `println!("hello {}", name)` with two arguments. Also, macros are expanded before the compiler interprets the meaning of the code, so a macro can, for example, implement a trait on a given type. A function can't, because it gets called at runtime and a trait needs to be implemented at compile time.

The downside to implementing a macro instead of a function is that macro definitions are more complex than function definitions because you're writing Rust code that writes Rust code. Due to this indirection, macro definitions are generally more difficult to read, understand, and maintain than function definitions.

Another important difference between macros and functions is that you must define macros or bring them into scope *before* you call them in a file, as opposed to functions you can define anywhere and call anywhere.

Declarative Macros with `macro_rules!` for General Metaprogramming

The most widely used form of macros in Rust is the *declarative macro*. These are also sometimes referred to as "macros by example," " `macro_rules!` macros," or just plain "macros." At their core, declarative macros allow you to write something similar to a Rust `match` expression. As discussed in Chapter 6, `match` expressions are control structures that take an expression, compare the resulting value of the expression to patterns, and then run the code associated with the matching pattern. Macros also compare a value to patterns that are associated with particular code: in this situation, the value is the literal Rust source code passed to the macro; the patterns are compared with the structure of that source code; and the code associated with each pattern, when matched, replaces the code passed to the macro. This all happens during compilation.

To define a macro, you use the `macro_rules!` construct. Let's explore how to use `macro_rules!` by looking at how the `vec!` macro is defined. Chapter 8 covered how we can use the `vec!` macro to create a new vector with particular values. For example, the following macro creates a new vector containing three integers:

```
let v: Vec<u32> = vec![1, 2, 3];
```

We could also use the `vec!` macro to make a vector of two integers or a vector of five string slices. We wouldn't be able to use a function to do the same because we wouldn't know the number or type of values up front.

Listing 19-28 shows a slightly simplified definition of the `vec!` macro.

Filename: src/lib.rs

```
#[macro_export]
macro_rules! vec {
    ( $( $x:expr ),* ) => {
        {
            let mut temp_vec = Vec::new();
            $(
                temp_vec.push($x);
            )*
            temp_vec
        }
    };
}
```

Listing 19-28: A simplified version of the `vec!` macro definition

---

> Note: The actual definition of the `vec!` macro in the standard library includes code to preallocate the correct amount of memory up front. That code is an optimization that we don't include here to make the example simpler.

The `#[macro_export]` annotation indicates that this macro should be made available whenever the crate in which the macro is defined is brought into scope. Without this annotation, the macro can't be brought into scope.

We then start the macro definition with `macro_rules!` and the name of the macro we're defining *without* the exclamation mark. The name, in this case `vec`, is followed by curly brackets denoting the body of the macro definition.

The structure in the `vec!` body is similar to the structure of a `match` expression. Here we have one arm with the pattern `( $( $x:expr ),* )`, followed by `=>` and the block of code associated with this pattern. If the pattern matches, the associated block of code will be emitted. Given that this is the only pattern in this macro, there is only one valid way to match; any other pattern will result in an error. More complex macros will have more than one arm.

Valid pattern syntax in macro definitions is different than the pattern syntax covered in Chapter 18 because macro patterns are matched against Rust code structure rather than values. Let's walk through what the pattern pieces in Listing 19-28 mean; for the full macro pattern syntax, see the Rust Reference.

First, we use a set of parentheses to encompass the whole pattern. We use a dollar sign (`$`) to declare a variable in the macro system that will contain the Rust code matching the pattern. The dollar sign makes it clear this is a macro variable as opposed to a regular Rust variable. Next comes a set of parentheses that captures values that match the pattern within the parentheses for use in the replacement code. Within `$()` is `$x:expr`, which matches any Rust expression and gives the expression the name `$x`.

The comma following `$()` indicates that a literal comma separator character could optionally appear after the code that matches the code in `$()`. The `*` specifies that the pattern matches zero or more of whatever precedes the `*`.

When we call this macro with `vec![1, 2, 3];`, the `$x` pattern matches three times with the three expressions `1`, `2`, and `3`.

Now let's look at the pattern in the body of the code associated with this arm: `temp_vec.push()` within `$()*` is generated for each part that matches `$()` in the pattern zero or more times depending on how many times the pattern matches. The `$x` is replaced with

each expression matched. When we call this macro with `vec![1, 2, 3];`, the code generated that replaces this macro call will be the following:

```
{
    let mut temp_vec = Vec::new();
    temp_vec.push(1);
    temp_vec.push(2);
    temp_vec.push(3);
    temp_vec
}
```

We've defined a macro that can take any number of arguments of any type and can generate code to create a vector containing the specified elements.

To learn more about how to write macros, consult the online documentation or other resources, such as "The Little Book of Rust Macros" started by Daniel Keep and continued by Lukas Wirth.

Procedural Macros for Generating Code from Attributes

The second form of macros is the *procedural macro*, which acts more like a function (and is a type of procedure). Procedural macros accept some code as an input, operate on that code, and produce some code as an output rather than matching against patterns and replacing the code with other code as declarative macros do. The three kinds of procedural macros are custom derive, attribute-like, and function-like, and all work in a similar fashion.

When creating procedural macros, the definitions must reside in their own crate with a special crate type. This is for complex technical reasons that we hope to eliminate in the future. In Listing 19-29, we show how to define a procedural macro, where `some_attribute` is a placeholder for using a specific macro variety.

Filename: src/lib.rs

```
use proc_macro;

#[some_attribute]
pub fn some_name(input: TokenStream) -> TokenStream {
}
```

Listing 19-29: An example of defining a procedural macro

The function that defines a procedural macro takes a `TokenStream` as an input and produces a `TokenStream` as an output. The `TokenStream` type is defined by the `proc_macro` crate that is included with Rust and represents a sequence of tokens. This is the core of the macro: the

source code that the macro is operating on makes up the input `TokenStream`, and the code the macro produces is the output `TokenStream`. The function also has an attribute attached to it that specifies which kind of procedural macro we're creating. We can have multiple kinds of procedural macros in the same crate.

Let's look at the different kinds of procedural macros. We'll start with a custom derive macro and then explain the small dissimilarities that make the other forms different.

How to Write a Custom `derive` Macro

Let's create a crate named `hello_macro` that defines a trait named `HelloMacro` with one associated function named `hello_macro`. Rather than making our users implement the `HelloMacro` trait for each of their types, we'll provide a procedural macro so users can annotate their type with `#[derive(HelloMacro)]` to get a default implementation of the `hello_macro` function. The default implementation will print `Hello, Macro! My name is TypeName!` where `TypeName` is the name of the type on which this trait has been defined. In other words, we'll write a crate that enables another programmer to write code like Listing 19-30 using our crate.

Filename: src/main.rs

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

#[derive(HelloMacro)]
struct Pancakes;

fn main() {
    Pancakes::hello_macro();
}
```

Listing 19-30: The code a user of our crate will be able to write when using our procedural macro

This code will print `Hello, Macro! My name is Pancakes!` when we're done. The first step is to make a new library crate, like this:

```
$ cargo new hello_macro --lib
```

Next, we'll define the `HelloMacro` trait and its associated function:

Filename: src/lib.rs

```
pub trait HelloMacro {
    fn hello_macro();
}
```

We have a trait and its function. At this point, our crate user could implement the trait to achieve the desired functionality, like so:

```
use hello_macro::HelloMacro;

struct Pancakes;

impl HelloMacro for Pancakes {
    fn hello_macro() {
        println!("Hello, Macro! My name is Pancakes!");
    }
}

fn main() {
    Pancakes::hello_macro();
}
```

However, they would need to write the implementation block for each type they wanted to use with `hello_macro`; we want to spare them from having to do this work.

Additionally, we can't yet provide the `hello_macro` function with default implementation that will print the name of the type the trait is implemented on: Rust doesn't have reflection capabilities, so it can't look up the type's name at runtime. We need a macro to generate code at compile time.

The next step is to define the procedural macro. At the time of this writing, procedural macros need to be in their own crate. Eventually, this restriction might be lifted. The convention for structuring crates and macro crates is as follows: for a crate named `foo`, a custom derive procedural macro crate is called `foo_derive`. Let's start a new crate called `hello_macro_derive` inside our `hello_macro` project:

```
$ cargo new hello_macro_derive --lib
```

Our two crates are tightly related, so we create the procedural macro crate within the directory of our `hello_macro` crate. If we change the trait definition in `hello_macro`, we'll have to change the implementation of the procedural macro in `hello_macro_derive` as well. The two crates will need to be published separately, and programmers using these crates will need to add both as dependencies and bring them both into scope. We could instead have the `hello_macro` crate use `hello_macro_derive` as a dependency and re-export the procedural macro code. However, the way we've structured the project makes it possible for programmers to use `hello_macro` even if they don't want the `derive` functionality.

We need to declare the `hello_macro_derive` crate as a procedural macro crate. We'll also need functionality from the `syn` and `quote` crates, as you'll see in a moment, so we need to add them as dependencies. Add the following to the *Cargo.toml* file for `hello_macro_derive`:

Filename: hello_macro_derive/Cargo.toml

```
[lib]
proc-macro = true

[dependencies]
syn = "2.0"
quote = "1.0"
```

To start defining the procedural macro, place the code in Listing 19-31 into your *src/lib.rs* file for the `hello_macro_derive` crate. Note that this code won't compile until we add a definition for the `impl_hello_macro` function.

Filename: hello_macro_derive/src/lib.rs

```
use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(HelloMacro)]
pub fn hello_macro_derive(input: TokenStream) -> TokenStream {
    // Construct a representation of Rust code as a syntax tree
    // that we can manipulate
    let ast = syn::parse(input).unwrap();

    // Build the trait implementation
    impl_hello_macro(&ast)
}
```

Listing 19-31: Code that most procedural macro crates will require in order to process Rust code

Notice that we've split the code into the `hello_macro_derive` function, which is responsible for parsing the `TokenStream`, and the `impl_hello_macro` function, which is responsible for transforming the syntax tree: this makes writing a procedural macro more convenient. The code in the outer function (`hello_macro_derive` in this case) will be the same for almost every procedural macro crate you see or create. The code you specify in the body of the inner function (`impl_hello_macro` in this case) will be different depending on your procedural macro's purpose.

We've introduced three new crates: `proc_macro`, `syn`, and `quote`. The `proc_macro` crate comes with Rust, so we didn't need to add that to the dependencies in *Cargo.toml*. The `proc_macro` crate is the compiler's API that allows us to read and manipulate Rust code from our code.

The `syn` crate parses Rust code from a string into a data structure that we can perform operations on. The `quote` crate turns `syn` data structures back into Rust code. These crates make it much simpler to parse any sort of Rust code we might want to handle: writing a full parser for Rust code is no simple task.

The `hello_macro_derive` function will be called when a user of our library specifies `#[derive(HelloMacro)]` on a type. This is possible because we've annotated the `hello_macro_derive` function here with `proc_macro_derive` and specified the name `HelloMacro`, which matches our trait name; this is the convention most procedural macros follow.

The `hello_macro_derive` function first converts the `input` from a `TokenStream` to a data structure that we can then interpret and perform operations on. This is where `syn` comes into play. The `parse` function in `syn` takes a `TokenStream` and returns a `DeriveInput` struct representing the parsed Rust code. Listing 19-32 shows the relevant parts of the `DeriveInput` struct we get from parsing the `struct Pancakes;` string:

```
DeriveInput {
    // --snip--

    ident: Ident {
        ident: "Pancakes",
        span: #0 bytes(95..103)
    },
    data: Struct(
        DataStruct {
            struct_token: Struct,
            fields: Unit,
            semi_token: Some(
                Semi
            )
        }
    )
}
```

Listing 19-32: The `DeriveInput` instance we get when parsing the code that has the macro's attribute in Listing 19-30

The fields of this struct show that the Rust code we've parsed is a unit struct with the `ident` (identifier, meaning the name) of `Pancakes`. There are more fields on this struct for describing all sorts of Rust code; check the `syn` documentation for `DeriveInput` for more information.

Soon we'll define the `impl_hello_macro` function, which is where we'll build the new Rust code we want to include. But before we do, note that the output for our derive macro is also a `TokenStream`. The returned `TokenStream` is added to the code that our crate users write, so

when they compile their crate, they'll get the extra functionality that we provide in the modified `TokenStream`.

You might have noticed that we're calling `unwrap` to cause the `hello_macro_derive` function to panic if the call to the `syn::parse` function fails here. It's necessary for our procedural macro to panic on errors because `proc_macro_derive` functions must return `TokenStream` rather than `Result` to conform to the procedural macro API. We've simplified this example by using `unwrap`; in production code, you should provide more specific error messages about what went wrong by using `panic!` or `expect`.

Now that we have the code to turn the annotated Rust code from a `TokenStream` into a `DeriveInput` instance, let's generate the code that implements the `HelloMacro` trait on the annotated type, as shown in Listing 19-33.

Filename: hello_macro_derive/src/lib.rs

```
fn impl_hello_macro(ast: &syn::DeriveInput) -> TokenStream {
    let name = &ast.ident;
    let gen = quote! {
        impl HelloMacro for #name {
            fn hello_macro() {
                println!("Hello, Macro! My name is {}!", stringify!(#name));
            }
        }
    };
    gen.into()
}
```

Listing 19-33: Implementing the `HelloMacro` trait using the parsed Rust code

We get an `Ident` struct instance containing the name (identifier) of the annotated type using `ast.ident`. The struct in Listing 19-32 shows that when we run the `impl_hello_macro` function on the code in Listing 19-30, the `ident` we get will have the `ident` field with a value of `"Pancakes"`. Thus, the `name` variable in Listing 19-33 will contain an `Ident` struct instance that, when printed, will be the string `"Pancakes"`, the name of the struct in Listing 19-30.

The `quote!` macro lets us define the Rust code that we want to return. The compiler expects something different to the direct result of the `quote!` macro's execution, so we need to convert it to a `TokenStream`. We do this by calling the `into` method, which consumes this intermediate representation and returns a value of the required `TokenStream` type.

The `quote!` macro also provides some very cool templating mechanics: we can enter `#name`, and `quote!` will replace it with the value in the variable `name`. You can even do some repetition similar to the way regular macros work. Check out the `quote` crate's docs for a thorough introduction.

We want our procedural macro to generate an implementation of our `HelloMacro` trait for the type the user annotated, which we can get by using `#name`. The trait implementation has the one function `hello_macro`, whose body contains the functionality we want to provide: printing `Hello, Macro! My name is` and then the name of the annotated type.

The `stringify!` macro used here is built into Rust. It takes a Rust expression, such as `1 + 2`, and at compile time turns the expression into a string literal, such as `"1 + 2"`. This is different than `format!` or `println!`, macros which evaluate the expression and then turn the result into a `String`. There is a possibility that the `#name` input might be an expression to print literally, so we use `stringify!`. Using `stringify!` also saves an allocation by converting `#name` to a string literal at compile time.

At this point, `cargo build` should complete successfully in both `hello_macro` and `hello_macro_derive`. Let's hook up these crates to the code in Listing 19-30 to see the procedural macro in action! Create a new binary project in your *projects* directory using `cargo new pancakes`. We need to add `hello_macro` and `hello_macro_derive` as dependencies in the `pancakes` crate's *Cargo.toml*. If you're publishing your versions of `hello_macro` and `hello_macro_derive` to crates.io, they would be regular dependencies; if not, you can specify them as `path` dependencies as follows:

```
hello_macro = { path = "../hello_macro" }
hello_macro_derive = { path = "../hello_macro/hello_macro_derive" }
```

Put the code in Listing 19-30 into *src/main.rs*, and run `cargo run`: it should print `Hello, Macro! My name is Pancakes!` The implementation of the `HelloMacro` trait from the procedural macro was included without the `pancakes` crate needing to implement it; the `#[derive(HelloMacro)]` added the trait implementation.

Next, let's explore how the other kinds of procedural macros differ from custom derive macros.

Attribute-like macros

Attribute-like macros are similar to custom derive macros, but instead of generating code for the `derive` attribute, they allow you to create new attributes. They're also more flexible: `derive` only works for structs and enums; attributes can be applied to other items as well, such as functions. Here's an example of using an attribute-like macro: say you have an attribute named `route` that annotates functions when using a web application framework:

```
#[route(GET, "/")]
fn index() {
```

This `#[route]` attribute would be defined by the framework as a procedural macro. The signature of the macro definition function would look like this:

```
#[proc_macro_attribute]
pub fn route(attr: TokenStream, item: TokenStream) -> TokenStream {
```

Here, we have two parameters of type `TokenStream`. The first is for the contents of the attribute: the `GET, "/"` part. The second is the body of the item the attribute is attached to: in this case, `fn index() {}` and the rest of the function's body.

Other than that, attribute-like macros work the same way as custom derive macros: you create a crate with the `proc-macro` crate type and implement a function that generates the code you want!

Function-like macros

Function-like macros define macros that look like function calls. Similarly to `macro_rules!` macros, they're more flexible than functions; for example, they can take an unknown number of arguments. However, `macro_rules!` macros can be defined only using the match-like syntax we discussed in the section "Declarative Macros with `macro_rules!` for General Metaprogramming" earlier. Function-like macros take a `TokenStream` parameter and their definition manipulates that `TokenStream` using Rust code as the other two types of procedural macros do. An example of a function-like macro is an `sql!` macro that might be called like so:

```
let sql = sql!(SELECT * FROM posts WHERE id=1);
```

This macro would parse the SQL statement inside it and check that it's syntactically correct, which is much more complex processing than a `macro_rules!` macro can do. The `sql!` macro would be defined like this:

```
#[proc_macro]
pub fn sql(input: TokenStream) -> TokenStream {
```

This definition is similar to the custom derive macro's signature: we receive the tokens that are inside the parentheses and return the code we wanted to generate.

Summary

Whew! Now you have some Rust features in your toolbox that you likely won't use often, but you'll know they're available in very particular circumstances. We've introduced several complex topics so that when you encounter them in error message suggestions or in other peoples' code, you'll be able to recognize these concepts and syntax. Use this chapter as a reference to guide you to solutions.

Next, we'll put everything we've discussed throughout the book into practice and do one more project!