This paragraph is the same as the one on page 448; I’m including it here to show where the new content should go. /Carol

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” on page XX.

This is the start of the new content on macros, some of which used to be in Appendix D. /Carol

Prod: this text should be added just before the summary for the end of Ch 19

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 declarative macros. 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 19-36 shows a slightly simplified definition of the vec! macro.

au: please check my added wingdings here

Filename: src/lib.rs

 #[macro\_export]

 macro\_rules! vec {

 ( $( $x:expr ),\* ) => {

{

let mut temp\_vec = Vec::new();

 $(

 temp\_vec.push($x);

)\*

 temp\_vec

}

};

}

Listing 19-36: 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 19-36 mean; for the full macro pattern syntax, see the reference at <https://doc.rust-lang.org/stable/reference/macros.html>.

First, a set of parentheses encompasses the whole pattern. A dollar sign ($) is next, followed by 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.

There are some strange edge cases with macro\_rules!. In the future, Rust will have a second kind of declarative macro that will work in a similar fashion but fix some of these edge cases. After that update, macro\_rules! will be effectively deprecated. With this in mind, as well as the fact that most Rust programmers will use macros more than write macros, we won’t discuss macro\_rules! any further. To learn more about how to write macros, consult the online documentation or other resources, such as “The Little Book of Rust Macros” at <https://danielkeep.github.io/tlborm/book/index.html>.

Procedural Macros for Generating Code from Attributes

This section is mostly different from what’s in Appendix D. /Carol

The second form of macros is procedural macros, which act more like functions (and are 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: custom derive, attribute-like, and function-like, 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. Using procedural macros looks like the code in Listing 19-37, where some\_attribute is a placeholder for using a specific macro.

Filename: src/lib.rs

use proc\_macro;

#[some\_attribute]

pub fn some\_name(input: TokenStream) -> TokenStream {

}

Listing 19-37: An example of using 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 crate 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 19-38 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 19-38: 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 reexport 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 = "0.14.4"

quote = "0.6.3"

To start defining the procedural macro, place the code in Listing 19-39 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

extern crate proc\_macro;

use crate::proc\_macro::TokenStream;

use quote::quote;

use syn;

#[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 19-39: 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 responsible for parsing the TokenStream and the impl\_hello\_macro function 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 (available from <https://crates.io/crates/syn>), and quote (available from <https://crates.io/crates/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 19-40 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 19-40: The DeriveInput instance we get when parsing the code that has the macro’s attribute in Listing 19-38

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 at <https://docs.rs/syn/0.14.4/syn/struct.DeriveInput.html> 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 19-41.

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 19-41: 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 19-40 shows that when we run the impl\_hello\_macro function on the code in Listing 19-38, the ident we get will have the ident field with a value of "Pancakes". Thus, the name variable in Listing 19-41 will contain an Ident struct instance that, when printed, will be the string "Pancakes", the name of the struct in Listing 19-38.

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 at https://docs.rs/quote 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 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 19-38 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 https://crates.io/, they would be regular dependencies; if not, you can specify them as path dependencies as follows:

[dependencies]

hello\_macro = { path = "../hello\_macro" }

hello\_macro\_derive = { path = "../hello\_macro/hello\_macro\_derive" }

Put the code in Listing 19-38 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 in that they can take an unknown number of arguments, for example. However, macro\_rules! macros can only be defined using the match-like syntax we discussed in the section “Declarative Macros with macro\_rules! for General Metaprogramming” on page XX. 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

This section is the same as the existing summary on page 448 and is included here to show how the new content should fit in. /Carol

Whew! Now you have some Rust features in your toolbox that you won’t use often, but you’ll know they’re available in particular circumstances. We’ve introduced several complex topics, so when you encounter them in error message suggestions or in other peoples’ code, you’ll 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!