A

Keywords

The following list contains keywords that are reserved for current or future use by the Rust language. As such, they cannot be used as identifiers, such as names of functions, variables, parameters, struct fields, modules, crates, constants, macros, static values, attributes, types, traits, or lifetimes.

Keywords Currently in Use

as—perform primitive casting, disambiguate the specific trait containing an item, or rename items in use and extern crate statements

break—exit a loop immediately

const—define constant items or constant raw pointers

continue—continue to the next loop iteration

crate—link an external crate or a macro variable representing the crate in which the macro is defined

else—fallback for if and if let control flow constructs

enum—define an enumeration

extern—link an external crate, function, or variable

false—Boolean false literal

fn—define a function or the function pointer type

for—loop over items from an iterator, implement a trait, or specify a higher-ranked lifetime

if—branch based on the result of a conditional expression

impl—implement inherent or trait functionality

in—part of for loop syntax

let—bind a variable

loop—loop unconditionally

match—match a value to patterns

mod—define a module

move—make a closure take ownership of all its captures

mut—denote mutability in references, raw pointers, or pattern bindings

pub—denote public visibility in struct fields, impl blocks, or modules

ref—bind by reference

return—return from function

Self—a type alias for the type implementing a trait

self—method subject or current module

static—global variable or lifetime lasting the entire program execution

struct—define a structure

super—parent module of the current module

trait—define a trait

true—Boolean true literal

type—define a type alias or associated type

unsafe—denote unsafe code, functions, traits, or implementations

use—import symbols into scope

where—denote clauses that constrain a type

while—loop conditionally based on the result of an expression

Keywords Reserved for Future Use

The following keywords do not have any functionality but are reserved by Rust for potential future use.

abstract

alignof

become

box

do

final

macro

offsetof

override

priv

proc

pure

sizeof

typeof

unsized

virtual

yield

B

Operators and Symbols

This appendix contains a glossary of Rust’s syntax, including operators and other symbols that appear by themselves or in the context of paths, generics, trait bounds, macros, attributes, comments, tuples, and brackets.

Operators

The following list contains the operators in Rust, an example of how the operator would appear in context, a short explanation, and whether that operator is overloadable. If an operator is overloadable, the relevant trait to use to overload that operator is listed.

PROD: I’m not sure how to handle this, would it be too big for a table? I think some structure with aligned columns would make it a great reference

! (ident!(...), ident!{...}, ident![...]): denotes macro expansion.

! (!expr): bitwise or logical complement. Overloadable (Not).

!= (var != expr): nonequality comparison. Overloadable (PartialEq).

% (expr % expr): arithmetic remainder. Overloadable (Rem).

%= (var %= expr): arithmetic remainder and assignment. Overloadable (RemAssign).

& (&expr, &mut expr): borrow.

& (&type, &mut type, &'a type, &'a mut type): borrowed pointer type.

& (expr & expr): bitwise AND. Overloadable (BitAnd).

&= (var &= expr): bitwise AND and assignment. Overloadable (BitAndAssign).

&& (expr && expr): logical AND.

\* (expr \* expr): arithmetic multiplication. Overloadable (Mul).

\* (\*expr): dereference.

\* (\*const type, \*mut type): raw pointer.

\*= (var \*= expr): arithmetic multiplication and assignment. Overloadable (MulAssign).

+ (trait + trait, 'a + trait): compound type constraint.

+ (expr + expr): arithmetic addition. Overloadable (Add).

+= (var += expr): arithmetic addition and assignment. Overloadable (AddAssign).

,: argument and element separator.

- (- expr): arithmetic negation. Overloadable (Neg).

- (expr - expr): arithmetic subtraction. Overloadable (Sub).

-= (var -= expr): arithmetic subtraction and assignment. Overloadable (SubAssign).

-> (fn(...) -> type, |...| -> type): function and closure return type.

. (expr.ident): member access.

.. (.., expr.., ..expr, expr..expr): right-exclusive range literal.

.. (..expr): struct literal update syntax.

.. (variant(x, ..), struct\_type { x, .. }): “and the rest” pattern binding.

... (expr...expr) in a pattern: inclusive range pattern.

/ (expr / expr): arithmetic division. Overloadable (Div).

/= (var /= expr): arithmetic division and assignment. Overloadable (DivAssign).

: (pat: type, ident: type): constraints.

: (ident: expr): struct field initializer.

: ('a: loop {...}): loop label.

;: statement and item terminator.

; ([...; len]): part of fixed-size array syntax

<< (expr << expr): left-shift. Overloadable (Shl).

<<= (var <<= expr): left-shift and assignment. Overloadable (ShlAssign).

< (expr < expr): less-than comparison. Overloadable (PartialOrd).

<= (expr <= expr): less-than or equal-to comparison. Overloadable (PartialOrd).

= (var = expr, ident = type): assignment/equivalence.

== (expr == expr): equality comparison. Overloadable (PartialEq).

=> (pat => expr): part of match arm syntax.

> (expr > expr): greater-than comparison. Overloadable (PartialOrd).

>= (expr >= expr): greater-than or equal-to comparison. Overloadable (PartialOrd).

>> (expr >> expr): right-shift. Overloadable (Shr).

>>= (var >>= expr): right-shift and assignment. Overloadable (ShrAssign).

@ (ident @ pat): pattern binding.

^ (expr ^ expr): bitwise exclusive OR. Overloadable (BitXor).

^= (var ^= expr): bitwise exclusive OR and assignment. Overloadable (BitXorAssign).

| (pat | pat): pattern alternatives.

| (|…| expr): closures.

| (expr | expr): bitwise OR. Overloadable (BitOr).

|= (var |= expr): bitwise OR and assignment. Overloadable (BitOrAssign).

|| (expr || expr): logical OR.

\_: “ignored” pattern binding. Also used to make integer literals readable.

? (expr?): error propagation.

Non-operator Symbols

The following list contains all non-letters that don’t function as operators; that is, they don’t behave like a function or method call.

Stand-Alone Syntax

'ident: named lifetime or loop label.

...u8, ...i32, ...f64, ...usize, etc.: numeric literal of specific type.

"...": string literal.

r"...", r#"..."#, r##"..."##, etc.: raw string literal, escape characters are not processed.

b"...": byte string literal, constructs a [u8] instead of a string.

br"...", br#"..."#, br##"..."##, etc.: raw byte string literal, combination of raw and byte string literal.

'...': character literal.

b'...': ASCII byte literal.

|...| expr: closure.

!: always empty bottom type for diverging functions.

Path-Related Syntax

ident::ident: namespace path.

::path: path relative to the crate root (i.e., an explicitly absolute path).

self::path: path relative to the current module (i.e., an explicitly relative path).

super::path: path relative to the parent of the current module.

type::ident, <type as trait>::ident: associated constants, functions, and types.

<type>::...: associated item for a type that cannot be directly named (e.g., <&T>::..., <[T]>::..., etc.).

trait::method(...): disambiguating a method call by naming the trait that defines it.

type::method(...): disambiguating a method call by naming the type for which it’s defined.

<type as trait>::method(...): disambiguating a method call by naming the trait and type.

Generics

path<...> (e.g., Vec<u8>): specifies parameters to generic type in a type.

path::<...>, method::<...> (e.g., "42".parse::<i32>()): specifies parameters to generic type, function, or method in an expression. Often referred to as turbofish.

fn ident<...> ...: define generic function.

struct ident<...> ...: define generic structure.

enum ident<...> ...: define generic enumeration.

impl<...> ...: define generic implementation.

for<...> type: higher-ranked lifetime bounds.

type<ident=type> (e.g., Iterator<Item=T>): a generic type where one or more associated types have specific assignments.

Trait Bound Constraints

T: U: generic parameter T constrained to types that implement U.

T: 'a: generic type T must outlive lifetime 'a. When we say that a type “outlives” the lifetime, we mean it cannot transitively contain any references with lifetimes shorter than 'a.

T : 'static: the generic type T contains no borrowed references other than 'static ones.

'b: 'a: generic lifetime 'b must outlive lifetime 'a.

T: ?Sized: allow generic type parameter to be a dynamically sized type.

'a + trait, trait + trait: compound type constraint.

Macros and Attributes

#[meta]: outer attribute.

#![meta]: inner attribute.

$ident: macro substitution.

$ident:kind: macro capture.

$(…)…: macro repetition.

Comments

//: line comment.

//!: inner line doc comment.

///: outer line doc comment.

/\*...\*/: block comment.

/\*!...\*/: inner block doc comment.

/\*\*...\*/: outer block doc comment.

Tuples

(): empty tuple (aka unit), both literal and type.

(expr): parenthesized expression.

(expr,): single-element tuple expression.

(type,): single-element tuple type.

(expr, ...): tuple expression.

(type, ...): tuple type.

expr(expr, ...): function call expression. Also used to initialize tuple structs and tuple enum variants.

ident!(...), ident!{...}, ident![...]: macro invocation.

expr.0, expr.1, etc.: tuple indexing.

Curly Brackets

{...}: block expression.

Type {...}: struct literal.

Square Brackets

[...]: array literal.

[expr; len]: array literal containing len copies of expr.

[type; len]: array type containing len instances of type.

expr[expr]: collection indexing. Overloadable (Index, IndexMut).

expr[..], expr[a..], expr[..b], expr[a..b]: collection indexing pretending to be collection slicing, using Range, RangeFrom, RangeTo, or RangeFull as the “index.”

C

Derivable Traits

In various places in the book, we’ve discussed the derive attribute that you can apply to a struct or enum definition.

The derive attribute generates code that will implement a trait with its own default implementation on the type you’ve annotated with the derive syntax. In this appendix, we provide a reference of all the traits in the standard library that you can use with derive. Each section covers:

What operators and methods deriving this trait will enable

What the implementation of the trait provided by derive does

What implementing the trait signifies about the type

The conditions in which you’re allowed or not allowed to implement the trait

Examples of operations that require the trait

If you want different behavior than that provided by the derive attribute, consult the standard library documentation for each trait for details on how to manually implement them.

The rest of the traits defined in the standard library can’t be implemented on your types using derive. These traits don’t have sensible default behavior, so it’s up to you to implement them in the way that makes sense for what you’re trying to accomplish.

An example of a trait that can’t be derived is Display, which handles formatting for end users. You should always consider the appropriate way to display a type to an end user: for example, what parts of the type should an end user be allowed to see? What parts would they find relevant? What format of the data would be most relevant to them? The Rust compiler doesn’t have this insight, so it can’t provide appropriate default behavior for you.

The list of derivable traits provided in this appendix is not comprehensive: libraries can implement derive for their own traits, making the list of traits you can use derive with truly open-ended. Implementing derive involves using a procedural macro, which is covered in Appendix D, “Macros.”

Debug for Programmer Output

The Debug trait enables debug formatting in format strings, which you indicate by adding :? within {} placeholders.

The Debug trait allows you to print instances of a type for debugging purposes, so you and other programmers using your type can inspect an instance at a particular point in a program’s execution.

The Debug trait is required, for example, in use of the assert\_eq! macro. This macro prints the values of instances given as arguments if the equality assertion fails so programmers can see why the two instances weren’t equal.

PartialEq and Eq for Equality Comparisons

The PartialEq trait allows you to compare instances of a type to check for equality and enables use of the == and != operators.

Deriving PartialEq implements the eq method. When PartialEq is derived on structs, two instances are equal only if all fields are equal and not equal if any fields are not equal. When derived on enums, each variant is equal to itself and not equal to the other variants.

The PartialEq trait is required, for example, with the use of the assert\_eq! macro, which needs to be able to compare two instances of a type for equality.

The Eq trait has no methods. Its purpose is to signal that for every value of the annotated type, the value is equal to itself. The Eq trait can only be applied to types that also implement PartialEq, although not all types that implement PartialEq can implement Eq. One example of this is floating point number types: the implementation of floating point numbers states that two instances of the not-a-number (NaN) value are not equal to each other.

An example of when Eq is required is for keys in a HashMap so the HashMap can tell whether two keys are the same.

PartialOrd and Ord for Ordering Comparisons

The PartialOrd trait allows you to compare instances of a type for sorting purposes. A type that implements PartialOrd can be used with the <, >, <=, and >= operators. You can only apply the PartialOrd trait to types that also implement PartialEq.

Deriving PartialOrd implements the partial\_cmp method, which returns an Option<Ordering> that will be None when the values given don’t produce an ordering. An example of a value that doesn’t produce an ordering, even though most values of that type can be compared, is the not-a-number (NaN) floating point value. Calling partial\_cmp with any floating point number and the NaN floating point value will return None.

When derived on structs, PartialOrd compares two instances by comparing the value in each field in the order in which the fields appear in the struct definition. When derived on enums, variants of the enum declared earlier in the enum definition are considered less than the variants listed later.

The PartialOrd trait is required, for example, for the gen\_range method from the rand crate that generates a random value in the range specified by a low value and a high value.

The Ord trait allows you to know that for any two values of the annotated type, a valid ordering will exist. The Ord trait implements the cmp method, which returns an Ordering rather than an Option<Ordering> because a valid ordering will always be possible. You can only apply the Ord trait to types that also implement PartialOrd and Eq (and Eq requires PartialEq). When derived on structs and enums, cmp behaves the same way as the derived implementation for partial\_cmp does with PartialOrd.

An example of when Ord is required is when storing values in a BTreeSet<T>, a data structure that stores data based on the sort order of the values.

Clone and Copy for Duplicating Values

The Clone trait allows you to explicitly create a deep copy of a value, and the duplication process might involve running arbitrary code and copying heap data. See “Ways Variables and Data Interact: Clone” on page XX for more information on Clone.

prod: confirm/link xref (ch4)

Deriving Clone implements the clone method, which when implemented for the whole type, calls clone on each of the parts of the type. This means all the fields or values in the type must also implement Clone to derive Clone.

An example of when Clone is required is when calling the to\_vec method on a slice. The slice doesn’t own the type instances it contains, but the vector returned from to\_vec will need to own its instances, so to\_vec calls clone on each item. Thus, the type stored in the slice must implement Clone.

The Copy trait allows you to duplicate a value by only copying bits stored on the stack; no arbitrary code is necessary. See “Stack-Only Data: Copy” on page XX for more information on Copy.

prod: confirm/link xref (ch 4)

The Copy trait doesn’t define any methods to prevent programmers from overloading those methods and violating the assumption that no arbitrary code is being run. That way, all programmers can assume that copying a value will be very fast.

You can derive Copy on any type whose parts all implement Copy. You can only apply the Copy trait to types that also implement Clone, because a type that implements Copy has a trivial implementation of Clone that performs the same task as Copy.

The Copy trait is rarely required; types that implement Copy have optimizations available, meaning you don’t have to call clone, which makes the code more concise.

Everything possible with Copy you can also accomplish with Clone, but the code might be slower or have to use clone in places.

Hash for Mapping a Value to a Value of Fixed Size

The Hash trait allows you to take an instance of a type of arbitrary size and map that instance to a value of fixed size, using a hash function. Deriving Hash implements the hash method. The derived implementation of the hash method combines the result of calling hash on each of the parts of the type, meaning all fields or values must also implement Hash to derive Hash.

An example of when Hash is required is in storing keys in a HashMap to store data efficiently.

Default for Default Values

The Default trait allows you to create a default value for a type. Deriving Default implements the default function. The derived implementation of the default function calls the default function on each part of the type, meaning all fields or values in the type must also implement Default to derive Default.

The Default::default function is commonly used in combination with the struct update syntax discussed in “Creating Instances from Other Instances with Struct Update Syntax” on page XX. You can customize a few fields of a struct and then set and use a default value for the rest of the fields by using ..Default::default().

prod: confirm/link xref (ch5)

The Default trait is required when, for example, you use the unwrap\_or\_default method on Option<T> instances. If the Option<T> is None, the unwrap\_or\_default method will return the result of Default::default for the type T stored in the Option<T>.

D

Macros

We’ve used macros like println! throughout this book but haven’t fully explored what a macro is and how it works. This appendix explains macros as follows:

What macros are and how they differ from functions

How to define a declarative macro to do metaprogramming

How to define a procedural macro to create custom derive traits

We’re covering the details of macros in an appendix because they’re still evolving in Rust. Macros have changed and, in the near future, will change at a quicker rate than the rest of the language and standard library since Rust 1.0, so this section is more likely to date than the rest of the book. Due to Rust’s stability guarantees, the code shown here will continue to work with future versions. But there may be additional capabilities or easier ways to write macros that weren’t available at the time of this publication. Bear that in mind when you try to implement anything from this appendix.

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 discussed 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 have.

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 difference between macros and functions is that macro definitions aren’t namespaced within modules like function definitions are. To prevent unexpected name clashes when using external crates, you have to explicitly bring the macros into the scope of your project at the same time as you bring the external crate into scope, using the #[macro\_use] annotation. The following example would bring all the macros defined in the serde crate into the scope of the current crate:

#[macro\_use]

extern crate serde;

If extern crate was able to bring macros into scope by default without this explicit annotation, you would be prevented from using two crates that happened to define macros with the same name. In practice, this conflict doesn’t occur often, but the more crates you use, the more likely it is.

There is one last important difference between macros and functions: you must define or bring macros into scope before you call them in a file, whereas you can define functions anywhere and call them anywhere.

Declarative Macros with macro\_rules! for General Metaprogramming

The most widely used form of macros in Rust are 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 have code associated with them; 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 is the code that replaces the code passed to the macro. This all happens during compilation.

Prod: confirm xref

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 with three integers inside:

prod: confirm xref

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.

Let’s look at a slightly simplified definition of the vec! macro in Listing D-1:

#[macro\_export]

macro\_rules! vec {

( $( $x:expr ),\* ) => {

{

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

$(

temp\_vec.push($x);

)\*

temp\_vec

}

};

}

Listing D-1: 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 we’re defining the macro is imported. Without this annotation, even if someone depending on this crate uses the #[macro\_use] annotation, the macro wouldn’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 will be 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 pieces of the pattern in Listing D-1 mean; for the full macro pattern syntax, see the reference at <https://doc.rust-lang.org/stable/reference/macros.html>.

prod: confirm xref

First, a set of parentheses encompasses the whole pattern. Next comes a dollar sign ($) followed by a set of parentheses, which 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 captured in $(). The \* following the comma 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: the temp\_vec.push() code within the $()\* part 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.

Given 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 Custom derive

The second form of macros is called procedural macros because they’re more like functions (which are a type of procedure). Procedural macros accept some Rust code as an input, operate on that code, and produce some Rust code as an output rather than matching against patterns and replacing the code with other code as declarative macros do. At the time of this writing, you can only define procedural macros to allow your traits to be implemented on a type by specifying the trait name in a derive annotation.

We’ll 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 D-2 using our crate.

src/main.rs

extern crate hello\_macro;

#[macro\_use]

extern crate hello\_macro\_derive;

use hello\_macro::HelloMacro;

#[derive(HelloMacro)]

struct Pancakes;

fn main() {

Pancakes::hello\_macro();

}

Listing D-2: 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:

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:

extern crate hello\_macro;

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 a default implementation for the hello\_macro function 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. But 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:

hello\_macro\_derive/Cargo.toml

[lib]

proc-macro = true

[dependencies]

syn = "0.11.11"

quote = "0.3.15"

To start defining the procedural macro, place the code in Listing D-3 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.

Notice the way we’ve split the functions in D-3; this will be the same for almost every procedural macro crate you see or create, because it makes writing a procedural macro more convenient. What you choose to do in the place where the impl\_hello\_macro function is called will be different depending on your procedural macro’s purpose.

hello\_macro\_derive/src/lib.rs

extern crate proc\_macro;

extern crate syn;

#[macro\_use]

extern crate quote;

use proc\_macro::TokenStream;

#[proc\_macro\_derive(HelloMacro)]

pub fn hello\_macro\_derive(input: TokenStream) -> TokenStream {

// Construct a string representation of the type definition

let s = input.to\_string();

// Parse the string representation

let ast = syn::parse\_derive\_input(&s).unwrap();

// Build the impl

let gen = impl\_hello\_macro(&ast);

// Return the generated impl

gen.parse().unwrap()

}

Listing D-3: Code that most procedural macro crates will need to have for processing Rust code

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 allows us to convert Rust code into a string containing that Rust code. The syn crate parses Rust code from a string into a data structure that we can perform operations on. The quote crate takes syn data structures and turns them 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 get called when a user of our library specifies #[derive(HelloMacro)] on a type. The reason is that we’ve annotated the hello\_macro\_derive function here with proc\_macro\_derive and specified the name, HelloMacro, which matches our trait name; that’s the convention most procedural macros follow.

This function first converts the input from a TokenStream to a String by calling to\_string. This String is a string representation of the Rust code for which we are deriving HelloMacro. In the example in Listing D-2, s will have the String value struct Pancakes; because that is the Rust code we added the #[derive(HelloMacro)] annotation to.

Note At the time of this writing, you can only convert a TokenStream to a string. A richer API will exist in the future.

Now we need to parse the Rust code String into a data structure that we can then interpret and perform operations on. This is where syn comes into play. The parse\_derive\_input function in syn takes a String and returns a DeriveInput struct representing the parsed Rust code. The following code shows the relevant parts of the DeriveInput struct we get from parsing the string struct Pancakes;:

DeriveInput {

// --snip--

ident: Ident(

"Pancakes"

),

body: Struct(

Unit

)

}

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/0.11.11/syn/struct.DeriveInput.html for more information.

At this point, we haven’t defined 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 last part of this hello\_macro\_derive function uses the parse function from the quote crate to turn the output of the impl\_hello\_macro function back into a TokenStream. The returned TokenStream is added to the code that our crate users write, so when they compile their crate, they get extra functionality that we provide.

You might have noticed that we’re calling unwrap to panic if the calls to the parse\_derive\_input or parse functions fail here. Panicking on errors is necessary in procedural macro code because proc\_macro\_derive functions must return TokenStream rather than Result to conform to the procedural macro API. We’ve chosen to simplify 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 String and a DeriveInput instance, let’s generate the code that implements the HelloMacro trait on the annotated type:

hello\_macro\_derive/src/lib.rs

fn impl\_hello\_macro(ast: &syn::DeriveInput) -> quote::Tokens {

let name = &ast.ident;

quote! {

impl HelloMacro for #name {

fn hello\_macro() {

println!("Hello, Macro! My name is {}", stringify!(#name));

}

}

}

}

We get an Ident struct instance containing the name (identifier) of the annotated type using ast.ident. The code in Listing D-2 specifies that the name will be Ident("Pancakes").

The quote! macro lets us write the Rust code that we want to return and convert it into quote::Tokens. This macro also provides some very cool templating mechanics; we can write #name and quote! will replace it with the value in the variable named 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!, 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 D-2 to see the procedural macro in action! Create a new binary project in your projects directory using cargo new --bin 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 from Listing D-2 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.

The Future of Macros

In the future, Rust will expand declarative and procedural macros. Rust will use a better declarative macro system with the macro keyword and will add more types of procedural macros for more powerful tasks than just derive. These systems are still under development at the time of this publication; please consult the online Rust documentation for the latest information.