Rust

Preface:

Rust is, (in my humble opinion,) *not* a beginners language. Despite the official documentation’s best efforts at welcoming new comers, the simple truth is that Rust expands upon languages of the past in numerous ways. It is syntactically rich and brings forth new concepts which are foreign to the average modern-day programmer. It is my recommendation that you learn at least three other languages prior to learning Rust. My recommendations are C, due to its low level concepts which lend well in regards to understanding how Rust works under the hood, C++, as it is a very syntax-heavy language, and a high level language such as Python, TypeScript, or Kotlin, as these will introduce you to modern programming idioms, most of which Rust adopts. This document will not waste time refreshing you on basics such as pointers, multithreading, enums, functions, classes, inheritance, or anything of the sort. Finally, note that at the time of writing this document, the future of Rust is largely unknown. Due to internal conflicts within the Rust Foundation, and turmoil/upstir in the community (hence the popular fork of Rust – Crablang), this document may very well become obsolete.   
  
Introduction:

I’m sure Rust requires little introduction, but I’ll recap in case you’re a bit behind. Rust is a procedural (and arguably OOP) language which offers incredible run-time speeds and memory safety. It is primarily targetted towards the systems programming crowd, but it has also been adopted widely in the web ecosystem thanks to its incredible compatibility with WASM. In order to implement memory-safe code and worry-free concurrency, Rust uses a system known as the borrow checker. We will discuss the borrow checker at nausium, as it is one of Rust’s core features. Assuming that you’ve heeded my warnings in the preface, and have some fundamental knowledge of programming, Rust’s learning curve will seem steep at first, but you will quickly begin to understand the reasoning behind the decisions made, and hopefully you will reap the benefits of the time sink it requires to learn the language by spending less time debugging difficult problems.

Types:

Rust is a type-safe language. One of my favorite aspects of Rust is that it forces the user to be precise about the size of types. Integer types are implicitly signed by default, but have unsigned variants which are prefixed by the letter ‘u’. The standard integer type is i32, but we can also declare a variable to be of type i8, i16, i64, i128, u8, u16, u32, u64, or u128. If you want a platform-dependent integer type, you may use isize or usize, which will occupy the length of a word. usize is actually very common in a lot of Rust’s APIs, so you may end up gravitating towards it more than you might expect. Integer literals can be represented in hexadecimal, octal, or binary using the standard prefixes (0x, 0o, 0b). Rust can also represent characters as bytes like so: b’A’, which will convert it to a u8. Decimals may include underscores to denote commas for readability e.g. 98\_222. In order to use floats and doubles, you may use f32 and f64, respectively. Rust actually does not support the ‘f’ postfix for floats. Floating point literals require that a digit preceed the decimal point, but they do not require that anything come after the decimal point. Characters in Rust (declared with the char keyword) are 4 bytes due to the fact that Rust ensures all strings are UTF-8 compliant. I will have a section dedicated to strings in Rust, as they are quite a complex and robust feature of the language.

Arrays and Vec<T>

Rust has built-in support for both statically sized arrays (stack-allocated) as well as dynamically sized arrays via the Vec struct (heap-allocated). A statically sized array is declared using square braces ([]). These braces require two arguments delimited by a semi-colon: the type and the length. Rust is very strict about knowing the length of stack-allocated variables at compile time. The only time that Rust will allow you to create dynamically sized objects is if they are heap-allocated. Therefore, to declare an array of bytes, we can do let bytes: [u8; 6] = [b’A’, b’B’, b’C’, b’D’, b’E’, b’F’]; If the array in this example did not contain exactly 6 elements at compile time, the compiler would throw an error, even if the number of elements was less than the length specified. It is at this point that I should bring up the fact that Rust supports type inference. Rather than explicitly providing the array’s type and length, we can instead create a dynamically sized array who’s type is inferred like so: let bytes = [b’A’, b’B’, b’C’, b’D’, b’E’, b’F’]; In this example, Rust infers the type to be a [u8; 6] (note that Rust treats different sizes of static arrays as completely different types, so [u8; 6] has a different type than [u8; 5], for example). We can provide, as an rvalue, an array initializer to quickly initialize or reinitialize an array with a specific value. The syntax for this is once again, to use square braces with two parameters: the value that the buffer will be initialized with, and the length of the initializer list. For example, we could say let a: [i32; 5] = [9; 5]; which would populate the array a with 5 9s. This is written more simplistically using type inferrence: let vals = [“Hello”; 10]; Vectors, as mentioned, are heap-allocated, and thus Rust allows them to be dynamically sized. A variable can explicitly be marked as a vector using Vec<T> e.g. let vector: Vec<bool>; In order to initialize a Vec in Rust, we use the vec! macro. Macros in Rust function similar to how they work in other languages, but are in many ways, more extensible. We can intialize a Vec like so: let vector = vec![1, 2, 3, 4, 5]; Rust will use type inferrence to infer the generic type of Vec in all of the examples where I’ve omitted the explicit type. If you want to use type inferrence for a Vec, but you don’t need to initialize it with any data, you can choose between assigning it a value of vec![] or Vec::new().

Strings

Strings in Rust can be quite daunting for most programmers due to some of the rules by which Rust abides. Unlike C or C++, strings in Rust are not null-terminated. To expound upon this, there are two primary ways of determining where a string terminates. The traditional way is to use a null-terminator, which is more memory efficient, but much less safe. The other method, which Rust uses, is to think of strings as a structure which first contains the string’s length, followed by the actual data. This occupies more memory since now we must store the length as a distinct field within each string object. In Rust, we also store the capacity of the string i.e. the maximum amount of bytes that the string can contain if it needs to grow. Strings in Rust are guaranteed to be valid UTF-8. If you’re not aware, UTF-8 is a text encoding standard which uses a variable-width encoding. Variable-width refers to the fact that a character may be represented as one or more bytes (in the case of UTF-8, up to 4). This allows it to be backwards compatible with ASCII, while also providing support for Unicode characters. Finally, Strings in Rust, alongside all datatypes in Rust, are immutable by default. As we’ll come to see, mutability is something that Rust takes very seriously. Creating a string requires slightly more effort than one may expect. If we attempt to make one e.g. let s: String = “hello world”; we get an error: “Expected String, found &str”. In Rust, the str type a.k.a. the string slice, represents a ‘view’ into a string. It represents the underlying string literal as a contiguous block of bytes. Unlike String, which is an owned type i.e. it owns the underlying data, str does not own the string literal that it points to, it simply has a read-only pointer to the data (thus it is considered to be a borrowed type). The str type also does not contain the capacity of the string, because unlike the String type, a str is not growable, though it still contains a length. Additionally, String always allocates its string literals in the heap, whereas str may reference string literals in heap memory, stack memory, or the .data section of the binary. The str type is usually seen in its borrowed form (&str) due to the fact that Rust requires that its size is known at compile time, meaning that we need to use either a reference or pointer (since references and pointers have known sizes). Although we’ll talk about lifetimes in more detail later, it is important to note that string literals in Rust have an implicit static lifetime, meaning that they live for the duration of the program. We can type the lifetime of a string literal explicitly like so: let s: &’static str = “hello world!”; Back to creating a String though. The error arose due to the fact that we were trying to assign a borrowed type (&str) to an owned type (String). There are a few ways to solve this dilemma, but I’ll cover three. The first is to use String::from() which will allocate some heap memory and perform a copy of the string slice if necessary. The second is to\_owned(), which will clone the string slice and return an owned String. Finally is into(), which is one way of performing type casting in Rust. We’ll look into type casting in more detail later, but essentially, the str type in Rust implements a trait which tells the compiler how to convert between str and String. The following examples all produce the same end result:

let s = String::from(“hello world”);

let s = “hello world”.to\_owned();

let s: String = “hello world”.into();

Tuples

The final type we’ll look at are tuples. In certain languages a tuple refers to a set of exactly two items. In Rust, a tuple is considered to be any set that contains *two or more* items. When we define a tuple, we’re really just defining multiple variables at the same time, each of which can be it’s own type, unlike arrays or Vecs. These variables do not necessarily need to be coupled to each other. Here is an example of a tuple: let tuple = (true, “word”, 4.5); The variable tuple contains three items. These three items are actually stored within the tuple struct as distinct member variables. The member variables are given a name according to their respective indices. For example, to access the first member variable of the tuple, we can do tuple.0; Note the use of the dot operator, indicating that this is not a list being indexed, but rather a struct with a distinct member variable, named 0, which is of type bool and stores the value true, as that is what we initialized it with. In the previous example, we used Rust’s type inferrence to determine the types for each member variable. In order to do this explicitly, it would look something like: let tuple: (bool, &str, f64) = (true, “word”, 4.5); Recall that I mentioned the items in a tuple did not necessarily need to be coupled. In the examples I’ve given thus far, each item in the tuple is bound to the tuple variable, meaning that they are dependent (or coupled) to the same variable. We can avoid this by using tuples to perform multiple assignment. For example, we could do: let (x, y, z) = (10.5, 456.32, 358.3); In this case, three distinct variables are initialized at the same time (x is assigned 10.5, y is assigned 456.32, and z is assigned 358.3). What we’ve done here is use the tuple (10.5, 456.32, 358.3) to perform multiple assignment on distinct variables. We can cast a regular tuple into multiple variables as well:

let tuple = (true, “word”, 4.5);

let (x, y, z) = tuple; // Assigns true to x, “word” to y, and 4.5 to z

Writing a basic Function:

A basic function in Rust looks like the following:

fn hello\_world() -> () {

println!(“Hello, World!”);

}

Rust does not have a void type like many other languages. Instead, we return what is known as the ‘unit’ type, denoted by an empty set of parentheses. In this example, returning the unit type was actually optional (we could have omitted the arrow operator altogether), but it is used frequently when returning a Result or Optional (which we will discuss soon). Note that in rust, println! is a macro (demonstrated by the exclamation mark). We will discuss the println! macro in more detail later. In terms of the semantics used to actually return values, Rust does not use the return keyword except in the case of early returns. Statements which are not terminated with a semi-colon are considered to be return statements. For example:

fn add(a: i32, b: i32) -> i32 {

a + b

}

When a function does not have a return type, but requires and early return, we can use the return keyword:

fn foo(list: Vec<i32>) {

for item in list {

if item == 8 return

}

}

Sometimes we have a function which never returns e.g. a thread that executes for the duration of the program or perhaps some microprocessor which spins until shutdown. In these situations, the appropriate “return type” is ‘!’. This indicates to the compiler that it is expected that the function will never return. Below is an example:  
fn run\_forever() -> ! {

loop { // The loop keyword in Rust is equivallent to while (true)

// Do something...

}

}

The Prelude:

Before we continue, I want to quickly make one thing clear about namespaces in Rust. Usually to utilize structures, functions, macros, etc. we need to include them with the use keyword. However, the Rust foundation decided that certain structs, functions, macros, etc. were used so frequently that it became a pain to always include their crates. Therefore, we have a prelude, which is a list of libraries that are automatically imported into your project. The prelude contains essentially everything from the core::\* package; primitive types; traits such as Clone, Copy, Debug, Default; macros such as println!, fmt!, vec!; and standard library items like String, Vec, Result, Option, etc.

Mutability:

Most languages provide mechanisms for controlling mutability. Conventionally, we’ve used the const keyword to restrict mutability. The const keyword indicates to the compiler that the value which is held by a variable should not be altered after initialization. The const keyword is a way for programmers to opt *into* immutability if they desire it. In Rust, variables are implicitly immutable, and we must opt *out* of immutability (in other words, opt into becoming mutable). This approach is arguably better because it really encourages programmers to only make variables mutable only when actually necessary. Take a look at this example:

let a = 3;

a = 4;

This code produces the error message “cannot assign twice to immutable variable”. This is because the variable a is immutable, so when we attempt to reassign it to another value, the compiler throws an error. The fix in this case is to qualify the variable a with the mut keyword. The mut keyword (short for mutable) will allow a to be altered, so the reassignment will succeed:

let mut a = 3;

a = 4;

Rust still has both the const and static keywords, though slightly repurposed. These keywords replace the let keyword when defining a variable in Rust. In Rust, we cannot create global variables using the let keyword, therefore, we must use either const or static. Both of these keywords create a global and immutable constant by default. The difference is that static does technically allow for the mut qualifier so long as all uses of the variable are placed within an unsafe {} block. Also, the rust compiler will enforce some semantics on you by forcing you to label const and static constants as uppercase. Ex:

const PI: f64 = 3.14159265358979323846264338327950288419716939;

Structs:

Rust does not have support for classes in the same manner that most OOP languages do, however, structs mimic many of the same design patterns typically attributed to classes. Structs are private by default, and so are their members. A struct declaration might look something like:

struct Foo {

val1: u64,

val2: f64,

val3: i32,

}

We can initialize a variable of type Foo using a C-style initializer list:

let foo = Foo { val1: 99, val2: 3.141, val3: -99, };

Structs in Rust may contain both methods and functions. We create a special scope for struct methods/functions using the impl (short for implement) keyword. Methods/functions are defined within this special impl scope, separate from the struct’s declaration. For example:

impl Foo { // Special implementation scope for defining methods that belong to Foo

fn new() -> Self {

Self {

val1: 0,

val2: 0.0,

val3: 0,

}

}

fn to\_string(&self) {

println!(“val1 = {}, val2 = {}, val3 = {}”, self.val1, self.val2, self.val3);

}

}

Rust makes a clear distinction between methods and functions. A function in Rust is equivallent to a static method in OOP languages. A method, on the other hand, is a function invoked upon an instance of the struct/object. In this example, the new() function is invoked statically e.g. Foo::new(). Methods use a reference to &self, which works similar to the this keyword in other languages, to implicate that the function should be called from an instance of Foo. Another imporant fact is that “&self” is syntactic sugar that expands to self: &Self, meaning that it’s a borrowed reference to the outer class with the name self.

Copying the contents of a struct in Rust is trivial. Rust provides us with the ‘..’ operator to allow for copying values from one struct to another. This is really useful for creating copy constructors. For example, let’s say we have a struct Person which has a name, gender, and career. Now say that we want to make a copy of an existing person with all of the same fields except for name. This can be accomplished like so:  
  
use std::error::Error;

#[derive(Debug, Copy, Clone)]

enum Career {

Firefighter,

Doctor,

Banker

}

#[derive(Debug, Copy, Clone)]

enum Gender {

Boy,

Girl,

}

#[derive(Debug, Copy, Clone)]

struct Person {

name: &'static str,

gender: Gender,

career: Career,

}

impl Person {

fn copy(&self, new\_name: &'static str) -> Self {

Person {

name: &new\_name,

..\*self

}

}

}

fn main() -> Result<(), Box<dyn Error>> {

let p1 = Person { name: "Steve", gender: Gender::Boy, career: Career::Firefighter };

let p2 = p1.copy("Myles");

dbg!(p1);

dbg!(p2);

Ok(())

}

This sample code contains a few unintroduced idioms, but the jist lies in the copy method. Here we take in the static string literal new\_name as the sole parameter (self is passed in implicitly), and then use ..\*self to use the callee’s data to fill in the remaining fields.

Rust provides us with some syntactic sugar to reduce the amount of boilerplate we need when gathering parameters from methods or functions. This is deemed “field init shorthand”, and is especially nice when creating a new() function (a very common pattern in Rust). Here is an example:  
  
struct Foo {

bar: i32,

baz: Option<bool>,

}

imp Foo {

fn new(bar: i32, baz: Option<bool>) -> Self {

Foo { bar, baz }

}

}

If you didn’t catch it, note that in the return statement, instead of having to explicitly declare each member variable in Foo followed by the assignment of bar and baz, we can instead use bar and baz directly. This is because the parameter names match the names given to the member variables of Foo. Without field init shorthand, we’d have to redundantly do the following:  
  
Foo { bar: bar, baz: baz }

Structs can also be declared with anonymous member variables in the form of a tuple, or just not contain any member variables at all. The following lines are all valid definitions:

struct Foo; // This is called a unit struct

struct Bar(i32);

struct Baz(u8, u8);

In order to intialize Bar and Baz, we can use a constructor-like syntax. We can access the values as we would with a normal :

let bar = Bar(10);

println!(“Value of bar: {}”, bar.0); // Prints “Value of bar: 10”

Enums:

In Rust, structs and enums are more or less the same. The major difference between them is the manner in which they are stored in memory. Enums are more akin to something like a union in C. When a variable is initialized using an enum, it can only contain one of the enum’s values, whereas a variable containing a struct allocates enough memory to store each member of the struct. Here is an example definition of an enum in Rust:

enum MyEnum {

Foo,

Bar(i32, u8),

Baz{ x: f64, y: f64 },

}

As you can see, enum values can optionally contain data. Foo does not have any associated data coupled to it, which is what you’re likely used to seeing when working with enums. We can also associate tuples containing 1 or more items, as seen in Bar. Interestingly, we cannot access the items of the tuple using the dot operator followed by the index, unlike structs. Enums can also contain struct data, as seen with Baz. These we can access using the dot operator followed by the member variable.

Match Statements:

Match statements replace switch statements from other languages. A conventional switch statements checks equality between two values, but the match statement can perform pattern matching between two items. In this regard, it is a bit more powerful than a traditional switch statement. We can perform a match to ensure that the values of each enum item are as we expect:

use std::error::Error;

enum Coin {

Penny,

Dime(f32),

Quarter(f32),

Dollar(i32),

Toonie(i32),

}

fn main() -> Result<(), Box<dyn Error>> {

match quart {

Coin::Penny => println!(“Coin is a penny”),

Coin::Dime(0.10) => println!(“Coin is a dime”),

Coin::Quarter(0.25) => println!(“Coin is a quarter”),

Coin::Dollar(1) => println!(“Coin is a dollar”),

Coin::Toonie(2) => println!(“Coin is a toonie”),

\_ => println!(“You picked up a foreign coin”),

}

Ok(())

}

In this example, each match expression checks not only if the value of quart matches with the specific enum sub-type, but also that their tuple values match. Therefore, it is not enough for quart to just be of type Coins::Quarter, because it also must contain the value 0.25. In match statements, the final case will be treated as the default case. This can either be the underscore operator (like in the example above), or a variable name which will be bound to the value of whatever is being matched against. Rust will force you to provide a default case if you do not check against all subtypes of an enum. This is actually a terrible example because at the time of writing this, Rust has stated that it will no longer support floating point comparisons in match statemtents, which sort of invalidates this example, but hopefully you understand the concept.

Matching against specific tuple values is one option that we have for enums, but we can also create new variables in our case statements which will be bound to the values held by the variable being matched against. This is essentially identical to what the aforementioned if-let statement does. For example:

match quart {

Coins::Penny => println!(“Coin is a penny”),

Coins::Dime(val) => println!(“Coin is a dime, and its value is {}”, val),

Coins::Quarter(val) => println!(“Coin is a quarter, and its value is {}”, val),

Coins::Dollar(val) => println!(“Coin is a dollar, and its value is {}”, val),

Coins::Toonie(val) => println!(“Coin is a toonie, and its value is {}”, val),

}

Here, we first match quart with the subtype Coins::Quarter. Then, we declare a new variable, val, which is bound to the tuple value held by quart. Exactly as I stated for the if-let statement, if quart had multiple tuple values, some of which we did not wish to bind, we could use the underscore operator to assign them to nothing, effectively ignoring them. Note that we don’t require the use of a default case here, since no matter what value is contained in any of the enum’s sub types, it will be bound to the local variable val. This means that if quart contains the value 0.3, this match statement will still consider it as a quarter. Of course, that doesn’t make much sense in our case, since I’m not good at creating meaningful examples, but there are many times where we want to capture the values of an enum’s value.

The if-let Statement:

The if-let statement performs pattern matching on an item. The boolean expression is considered true if the pattern matches. Not only this, but the ‘let’ part of the if-let statement creates and binds a temporary variable to be used within the if block. Let’s look at an example written in C that we can compare. The C version would be something like the following:

char c1 = ‘z’;

if ((char c2 = ‘z’) == c1) {

// Do something

}

The Rust version looks like this:

let c1 = ‘z’;

if let c2 = c1 {

// Do something

}

Of course the above example isn’t a very interesting one, however, we can perform any of the pattern matching that would normally be used in a match block. For example:

use std::ops::RangeInclusive;

enum LoadCapacities<T> {

Truck(RangeInclusive<T>),

Car(RangeInclusive<T>),

Motorcycle(RangeInclusive<T>),

}

...

let car\_load = LoadCapacities::Car(2000..=3000);

if let LoadCapacities::Car(range) = car\_load {

println!("The load capacity of a car is {} to {} lbs", range.start(), range.end());

}

The Borrow Checker:

We should have enough basic knowledge to begin discussing the borrow checker. It is critical that we understand how borrow checker works now, as it will apply to pretty much everything else going forward. In Rust, we have the concept of ownership. Whenever an assignment operation occurs, the lvalue (variable) is considered to become the owner of the rvalue. For primitive types that are stack-allocated, assigning a value owned by variable A to variable B will not transfer ownership from A to B. This is because primitive types in Rust (which are in fact structs in-and-of themselves) implement something known as the Copy trait. Thus, within the following code snippter, the assignment operator will actually copy the value owned by A into B so that they both contain the same value without transfering ownership of the value:

fn main() {

let a = 3;

{

let b = a;

}

println!(“Value of a = {}”, a);

}

This code will compile successfully, because variables a and b are of type i32, which is a struct that implements the Copy trait. The statement let b = a; when compiled, will expand to a memcpy() from a to b and perform a bitwise copy. Structs that we create which do not implement the Copy trait will be moved into the lvalue upon assignment by default. Upon reassignment of a value, the old owner is invalidated, since the borrow checker only allows there to be one owner of an rvalue at a time. Take the following code for example:

fn main() {

let a = Foo::new();

{

let b = a; // Ownership of the Foo instance is transferred from a to b

} // b goes out of scope

a.to\_string();

}

The error message produced will be “a.to\_string(); value borrowed here after move”. This is because the value held by a is moved into b within the inner scope, making b the new owner. Remember, our Foo struct does not implement the Copy trait, so a is moved, not copied, to b. The moment that ownership is transferred from a to b, a is invalidated. Once b goes out of scope the rvalue that it held is dropped.

As an aside, if you’re like me, you may be asking yourself how Rust actually “invalidates” a. After all, the most basic instruction in assembly for transferring values is the mov operation, which actually copies the contents of one register to another. If this was your thought, you’d be correct. When Rust “moves” a value to another location, it still just copies the value. It is the compiler that actually enforces the fact that a is now “invalidated”, even though technically speaking it is still a valid memory address with valid content. In other words, the terms copy and move in Rust are semantic distinctions, but in terms of the underlying implementation, they both perform bitwise copies. The compiler is what enforces the restrictions which prevent us from referring to a after it has been “invalidated”.

Let’s discuss how we deal with this in Rust. One method is to implement either the Copy or Clone trait. The primary difference between Copy and Clone is that structs which implement Copy implicitly invoke memcpy() whenever the assignment operator is used, whereas structs which have the Clone attribute inherit the default implementation of the clone() function, which is a function that must be explicitly invoked. Note that the Copy trait inherits the Clone trait, therefore, in order to use the Copy trait, we must also implement the Clone trait.

// Example 1: Using Clone trait and explicit call to clone() method

#[derive(Clone)]

struct Foo {

...

}

fn main() {

let a = Foo::new();

{

let b = a.clone();

}

a.to\_string();

}

// Example 2: Using Copy trait and implicit copy via assignment operator

#[derive(Copy, Clone)]

struct Foo {

...

}

fn main() {

let a = Foo::new();

{

let b = a;

}

a.to\_string();

}

These solutions work for certain structs, but not all structs, therefore theare not the ideal way of handling ownership. Rather than assign b the value of a directly, making b the new owner, we can instead grant b with a reference to the value held by a. By doing so, a still owns the value of the Foo struct, but b owns a reference to the same underlying memory. We do this using the ampersand, which you’ve already seen in previous examples. The new code looks like this:

fn main() {

let a = Foo::new();

{

let b = &a;

}

a.to\_string();

}

When we give b a reference to a, we say that b “borrows” a. That is to say, for the duration of the inner scope, b borrows access to the data held by a. Note that in this example, b is borrowing an immutable reference to a, meaning that b is not allowed to modify a (or in other words, the reference to a is read-only). We can allow b to modify a by giving it a *mutable* reference to a like so:

fn main() {

let mut a = Foo::new();

{

let b = &mut a;

b.val1 = 20;

}

a.to\_string();

}

Note that in order to let b borrow a mutable reference to a, a itself needs to be mutable. After all, it wouldn’t make sense if b could mutate a when a is declared immutable.

So far I’ve demonstrated the same example of a main function that contains the declaration of variable a in the outer scope and the declaration of variable b in the inner scope. Of course, considering that functions have their own scopes, the same ownership principles apply when we invoke a function. Therefore, values which are passed as arguments to a function will transfer ownership to the parameters of that function (i.e., ownership of the rvalue held by the original owner is transferred to the parameter created for the function’s stack frame). If you desire to be able to still use the lvalue of a variable after passing it to a function, then the function’s parameter should borrow a reference to said variable, exactly like in the previous two examples:

fn do\_something(list: &Vec<i32>) {

for item in list {

println!("Item: {}", item);

}

}

fn main() {

let mut list = Vec::from\_iter(1..=100);

do\_something(&list);

list.pop();

}

In this example we create a Vec<i32> using the Vec::from\_iter() function. We pass in the range 1..=100. The equals sign indicates that 100 should be included within the range, otherwise it would be excluded. The do\_something() function borrows a reference to list, meaning that it takes ownership of the reference to list, but not the actual Vec<i32> that is owned by list. This is so that the do\_something() function may use list while still allowing us to call pop() on list afterwards. If do\_something() took ownership over the rvalue of list, the statement list.pop() would produce a compiler error.

One of the fundamental rules of Rust is that we can either hold several immutable references to an object, or a single mutable reference to an object, but not both at the same time. This may sound slightly arbitrary at first, but it actually makes a lot of sense if you think about it. If variable a holds an immutable reference to some data, but another variable b holds a mutable reference to the same data, then a is not, in fact, truly immutable, since b can mutate it at any time. Likewise, Rust does not allow for multiple mutable references because then it becomes unclear who owns the data and therefore, who must clean it up once it goes out of scope. There are exceptions to these rules which we’ll look into later.

Traits:

Traits are Rust’s answer to interfaces, abstract classes, and polymorphism in OOP languages. Rust does not allow for inheritance in the traditional sense. In a language like C++, a class which inherits from another class gains all the properties of the parent including member variables and member functions. Rust cannot do this, and yet we still want to reap the benefits of polymorphism. We can use traits to force a struct to implement a specified set of functions and/or methods. The difference between traits and traditional interfaces is that traits can define a default implementation for a function. In this regard, they act a little more like abstract classes. Creating a trait is as simple as defining it, along with the functions that must be implemented by it:

pub trait Summary {

fn summarize(&self) -> String;

}

Then, in order to implement this trait for an existing struct, we use the same impl keyword that we’ve used before for struct methods, but we add the for keyword:

pub struct Tweet {

pub username: String,

pub content: String,

pub reply: bool,

pub retweet: bool,

}

impl Summary for Tweet {

fn summarize(&self) -> String {

format!(“{}: {}”, self.username, self.content)

}

}

Notice the statement “impl Summary for Tweet { ... }”. We create this special scope specifically for methods that require an implementation from the Summary trait. We need to create a unique scope like this for each trait that is implemented by the struct in question. Additionally, if we want to have methods that are unique to our struct, we need a regular impl {} block that is separate from the other “impl trait for” {} blocks.

As I mentioned, we can also create default implementations for traits in order to remove redundancy when implementing a trait for multiple structs which perform the same behavior. This works as you’d expect: define the trait, and then define the method rather than simply declaring it.

pub trait Summary {

fn summarize(&self) -> String {

String::from(“(Read more...)”)

}

}

If all methods within a trait have default implementations that we don’t wish to override, then all we have to do to implement them for a specific struct is create an empty impl for block:

impl Summary for Tweet {}

If you wanted to override the functionality for a default trait method, you would simply redefine that specific method within the impl-trait-for block.

Generics:

Rust, like most other languages, allows us to use type generics to help us avoid redundant code. As I’m sure you’re aware, generics allow us to perform a sort of meta-programming, where we specify the concrete type at the call site of a function, and then replace all of the references to the generic type with the concrete type. If you’re *not* familiar with generics, then what I said likely went over your head, but I promise that generics are actually quite simple. The notation for generics in Rust is pretty much the same as in any other language. A pair of angle brackets specifies the list of generic types that we want to accept into our struct, enum, method, or trait. The convension is to use a single upper case letter such as T (for type), though you’re permitted to use anything. Defining a struct looks something like the following:

struct Foo<T> {

a: T,

b: T,

}

...

let foo = Foo{ a: 5, b: 9 };

The type T is infered when we initialize the object foo (in this case, the compiler will infer T to be an i32). Attempting to initialize b with a type that differs from a will result in a compile-time error since a and b should both of the same type T. Likewise, we can use generics on functions like so:

fn func<T, U>do\_something(arg1: T, arg2: U) {

...

}

For structs that accept generic types, we can implement traits for a specified set of concrete types. For example, perhaps we only want to implement a method called return\_fractional() for instances of a struct Foo<T> if T is either the concrete type f32 or f64. This is accomplished like so:

struct Foo<T> {

x: T

}

trait Fractional {

fn return\_fractional(&self);

}

impl Fractional for Foo<f32> {

fn return\_fractional(&self) {

self.x.fract();

}

}

impl Fractional for Foo<f64> {

fn return\_fractional(&self) {

self.x.fract();

}

}

If we want to create an impl block for functions that work on any concrete type, then we need to use a bit of a special notation. It would look something like this:

impl<T> Foo<T> {

...

}

The reason that we require a generic type in the impl block as well as the struct is because we could potentially have more than one generic type, in which case the compiler needs to know which type is getting mapped to what. Again this will make more sense when we get to blanket implementations in a bit, but for now just know that we need to have the generic specified for both the impl block and the struct that we’re implementing it for.

Trait Bounds:

Interestingly, traits can be treated as types. This is an important feature because it allows us to be as broad or as specific as we want with the kinds of objects we accept as parameters. For instance, we can create a function called notify() which only accepts structs that implement the Summary trait like so:

pub fn notify(item1: &impl Summary, item2: &impl Summary) { ... }

This is called the impl syntax and works in cases where the traits that need to be implemented for item1 and item2 are allowed to be different from each other. In this example, item1 and item2 both need to implement Summary, but this doesn’t have to be the case. In order to enforce that item1 and item2 must implement the same trait, we can use something called trait bounds. Trait bounds operate under the guise of looking like generics, but are not actually true generics.

pub fn notify<T: Summary>(item1: &T, item2: &T) { ... }

We can add additional traits aside from just Summary so that item1 and item2 must implement Summary as well as the additional traits that we specify:

pub fn notify(item: &(impl Summary + Display)) { ... }

And here is the same statement using the trait-bound syntax:

pub fn notify<T: Summary + Display>(item: &T) { ... }

But wait, there’s more! Rust provides more syntax sugar to make writing trait-bounds even easier (this doesn’t apply to the impl syntax). For example, consider a case where we have two parameters, the first of which has a set of trait boundaries, and the second of which also has a set of trait boundaries, but distinct from the first set:

fn some\_function<T: Display + Clone, U: Clone + Debug>(t: &T, u: &U) -> i32 { ... }

As you can tell, this can become cumbersome to read pretty quickly. We can use the ‘where’ keyword to declare the types for our generics between the method’s signature and the method’s body, which reduces clutter in the signature:

fn some\_function<T, U>(item1: &T, item2: &U) -> i32

where

T: Display + Clone,

U: Clone + Debug,

{ ... }

Blanket Implementations:

Traits can be implemented on any type, even those external to our current package, using something called blanket implementations. The conditions for applying blanket implementations are that either 1) the trait must be defined in the current crate, or 2) the type upon which the trait is being implemented must be defined in the current crate. So for example, I can’t just apply the Copy trait on some external type defined in the standard library because neither the Copy trait, nor the external type are defined within the current crate. I can, however, implement a custom trait on a char, so long as that trait is defined within the current crate. This is super useful for creating extension methods which expand the functionality of a struct. Heres an example:

trait Palindrome {

fn is\_palindrome(&self) -> bool;

}

// Blanket implementation on str

impl Palindrome for str {

fn is\_palindrome(&self) -> bool {

self.chars().eq(self.chars().rev())

}

}

fn main() -> () {

let s = String::from("racecar");

if s.is\_palindrome() {

println!("{s} is a palindrome");

} else {

println!("{s} is not a palindrome");

}

}

Blanket implementations also allow us to implement traits for types that implement other traits. For example, we can implement a trait on all types that implement the Display trait like so:

impl<T: Display> ToString for T { ... }

Traits can contain generics themselves, which lets us do the following cool thing as shown in the Rust by example book:

struct Empty;

struct Null;

trait DoubleDrop<T> {

fn double\_drop(self, \_: T);

}

impl<T, U> DoubleDrop<T> for U {

// This method takes ownership of both passed arguments, deallocating both

fn double\_drop(self, \_: T) {}

}

fn main() {

let empty = Empty;

let null = Null;

empty.double\_drop(null);

}

In this example, the trait DoubleDrop takes in a generic type similar to what we do for impl blocks. We define a template for the double\_drop() method within the trait block, then do a blanket implement on the generic type U in the impl block. The Rust compiler can infer U based on the first caller of the double\_drop() method (in this case, it resolves to Empty). The generic T resolves to Null. In case you’re wondering why this function drops U, it’s because we use self rather than &self, so double\_drop() takes ownership over the empty variable, which invalidates it once the function returns.

Super Traits:

Super traits are similar to inherritance in OOP languages. They allow you to specify traits that must be implemented alongside the trait that is being defined. The syntax for this looks something like the following:

trait TraitName: RequiredTrait { ... }

Similar to trait bounds, we can specify multiple required traits using the + operator. The implementor will now be required to implement all functions/methods within TraitName *and* RequiredTrait that don’t have default implementations.

impl vs dyn:

We’ve seen how to create our own types from traits and utilize them when taking in parameters to a function, for example. To accomplish this, we used the ‘impl’ keyword. In Rust, there exists another keyword which we can utilize to accomplish the exact same thing, which is the ‘dyn’ keyword (short for dynamic dispatch). The difference between impl and dyn is not what they do (they both specify a trait as a type), but rather how they function under the hood. The impl keyword works by creating a unique copy of the function for each unique type that gets passed into it, similar to function overloading. This is called static dispatch, and it works because the compiler can use static analysis to find each call site of the function that accepts the trait(s) and then makes a copy of the function for each unique type that implements the required traits. This is in contrast to dynamic dispatch using the dyn keyword. Dynamic dispatch is essentially equivallent to marking an object as virtual in C++. A fat pointer (a struct that contains two pointers: one to the object and one to the vtable entry) will be implicitly passed to the function. The appropriate function will then be invoked from the vtable according to the type of the object that was passed into the function. When using the dyn keyword, you must either specify that the type will be passed as a reference or as a pointer using a wrapper like Box, but you can’t use the dyn keyword by itself like you can with impl. This is because, unlike impl, you can’t pass direct ownership to an object with the dyn keyword. People tend to gravitate towards the smart pointer approach since lifetimes aren’t always viable. For instance, the main() method cannot accept generics, and thus cannot accept lifetimes (though the ‘static lifetime is still permissible).

If you’re not aware, vtable lookups tend to incur a runtime cost due to pointer indirection. Static dispatch, however, causes the binary code to be larger in size due to the fact that it needs to create a unique method definition per type that the method is invoked with. Generally speaking, it’s best practice to use static dispatch whenever possible, though there are circumstances in which this is not possible, which is where dynamic dispatch may be necessary.

Associated Types for Traits:

We’ve seen in previous examples that traits are one of many constructs that can take advantage of generic types. The issue with generics is that they must satiate the conditions for all possible values of T. For example, a List structure would need to be able to perform it’s operations regardless of what T resolved to. Using associated types in traits allows us to pass in a concrete type when referencing the function or method. This functions similarly to doing a blanket implementation for a specific type T e.g. impl Foo<&[u8]> { ... }. We create a type alias using the ‘type’ keyword within the function definition. Default types can also be specified, though I’m getting a bit ahead of myself.

Iterators and Enumeration:

An iterator is a looping structure which is *lazy*. That is to say that elements within the iterator are retrieved at runtime only when being read, rather than being loaded into a structure prior to access like with Vec, for example. The primary difference between an *iterator* and an *enumerator* in Rust is that an iterator will only retrieve the next value in the collection, whereas an enumerator will retrieve both the next value, as well as its index and return both as a tuple. There are three methods that you need to be aware of when dealing with iterators in Rust, and these are: iter(), into\_iter(), and iter\_mut(). I don’t particularly feel like the names of these functions provide an accurate description of what they do, so allow me to explain. The iter() method, when applied to a collection that implements the Iterator trait, returns an iterator of *references* to the items within that collection. Occasionally, what we want instead is an iterator that returns the collection’s items as owned values, which is what into\_iter() will return. And as you may have guessed, mut\_iter() returns an iterator over the collection, but returns each item as a mutable reference.

In order to retrieve an enumerator for a collection, simply invoke the collection’s enumerate() method. All iterators will end with an Optional::None item indicating the end of the iterator. The next(), nth(), and last() methods get the next, nth, or last item (or for enumerators, index-item pair) respectively. The next\_back() method gets the last element of the iterator and pops it. step\_by() is how we change the step size of our iteration so that we jump n elements forward in the collection. The zip() method takes two iterators and turns them into one iterator of pairs. map() takes a closure and applies the operation to each item in the iterator, whereas filter() takes a closure and removes all items that return false when checked against some boolean logic. filter\_map() creates an iterator that both filters and maps at the same time. Very importantly are the collect() and collect\_into<E>() methods. These transform the iterator back into a collection, which is useful when you want to apply a map and/or filter but continue to iterate over the collection within a for loop, for example. Some other favorites of mine are the all() and any() methods. The all() method takes a closure with a boolean expression that each item in the iterator is tested against and returns true if all of the items in the iterator pass the condition. The any() method is similar, but returns true as soon as a single item matches the condition. There are many many other methods I could cover, however, for the sake of brevity, I will consider the methods listed here to be an adequate starting point.

In order to accept an Iterator as a parameter to a function, we check if the type implements the Iterator trait. The iterator trait uses an associated type, as we discussed earlier, to determine what type to use in its methods. Look at the next() method, for example.

pub trait Iterator {

type Item;

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

}

Normally when we use the type keyword we initialize the type at the same time we declare it, however, this does not have to be the case. In the case of traits, it is expected that the implementor will assign a value for each type alias within the impl block for that trait, or in the case of function/method parameters, within angle braces(<>). Let’s say we now want to accept an Iterator as input to a function. We do so like this:

use std::fmt::Display;

fn accepts\_iter<T: Display>(i: impl Iterator<Item = T>) {

for item in i {

println!("{item}");

}

}

fn main() -> () {

let it = vec![1, 2, 3].into\_iter();

accepts\_iter(it);

}

Ranges:

Iterators come in another form, which are ranges. The Range structure in Rust implements the Iterator trait, thus they are basically wrappers around an iterator. Ranges add the contains(), is\_empty(), start(), and end() methods on top of all of the methods implemented by the Iterator trait. The notation for creating a range is something like start..end, where the double dot operator can be read as “to” i.e. start *to* end. Ranges are by default, non-inclusive, meaning that end will not be included as a value within the iterator. If you want an inclusive range, we use the following notation: start..=end. We can do some really cool things using ranges. Here’s some code that prints the alphabet:

fn main() -> () {

let letters = 'a'..='z';

for c in letters {

print!("{c}");

}

}

We can also use this in our match statements to effectively filter out certain elements, such as in the following example where we filter out every item in the list aside from the first and last:

fn main() {

let numbers = (2, 4, 8, 16, 32);

match numbers {

(first, .., last) => {

println!(“Some numbers: {first}, {last}”);

}

}

}

Ranges are also used for creating slices. For instance, let’s say that we have a classic hello world string and we only want from the 7th letter/index onwards. We can do that like so:

fn main() -> () {

let greeting: String = String::from("Hello, World!");

let substr: &str = &greeting[7..];

println!("{substr}");

}

Note that in this example we have an existing collection (greeting) so we don’t need to explicitly define the end of our range. In other words, 7.. is read as “from index 7 to end”. Likewise, ..7 would yield everything from the beginning of the collection until the 7th element, and Simply putting .. would yield a slice containing all of the elements belonging to the original collection.

Depending on the notation used to construct the range, Rust will use one of several structs to represent the actual object. Here are the mappings:

* start..end = Range
* start.. = RangeFrom
* ..end = RangeTo
* ..=end = RangeToInclusive
* start..=end = RangeInclusive
* .. = RangeFull

Result and Option Enums:

Rust does not normally allow the use of NULL (though there are edge-cases, such as when initializing a raw-pointer within an unsafe block). Modern languages have been trying to find other paradigms to avoid the use of NULL. In languages like C# and Kotlin, types are implicitly non-nullable unless explicitly stated. C# and Kotlin both use a question mark operator to indicate that a type can be assigned NULL. This solution is okay, but I believe Rust has a better solution, which are the Result and Option enums. The Result enum in Rust has the following definition:

pub enum Result<T, E> {

Ok(T),

Err(E),

}

A Result is typically returned from any function which may error. If no errors occurred, the subtype Result::Ok will be returned with a mandatory value, and if an error occurred, the subtype Result::Err will be returned with a mandatory value. The Option enum is similar, but not identical:

pub enum Option<T> {

Some(T),

None,

}

The primary difference here is that the subtype None cannot contain a value. We use Option when a function may optionally return a value, but doesn’t necessitate returning a value. None effectively replaces NULL. Unlike NULL which is usually the value 0, None is an enum, which makes error handling more graceful.

Match statements are a great way of handling either of these types when they are returned from a function. We can directly do something in the case of success or failure:

match foo(arg) {

Ok(ok\_msg) => println!(“{}”, ok\_msg),

Err(err\_msg) => println!(“{}”, err\_msg),

}

Here is another example where we try opening a file using the [File::open](../../../../:open)() function which either returns an Ok with a handle to the file or an Err with an error message:

let file\_result = [File::open](../../../../:open)(“myfile.txt”);

let file = match file\_result {

Ok(file) => file,

Err(error) => panic!(“Error opening the file: {:?}”, error),

};

In the example above, file\_result is of type Result. We do a match on it to see if it contains the subtype Ok or Err. In the case of Err, we panic, meaning that we cause the program to abort. In the case that opening the file succeeded, we return the file handle, which gets assigned to the file variable. This example can be abbreviated using the unwrap() function. The unwrap() function essentially performs the same match statement that we created in the example above, either returning the value of Ok or calling panic!() with an error message. For example:

let file = [File::open](../../../../:open)(“myfile.txt”).unwrap();

If you want to provide your own custom error message, use expect() instead, which does the same thing as unwrap(), but accepts a string for the custom error message:

let file = [File::open](../../../../:open)(“myfile.txt”).expect(“Something went wrong opening myfile.txt”);

If you need more control than that, use unwrap\_or\_else(), which takes a closure and will execute if the Result was of type Err:

let file = [File::open](../../../../:open)(“myfile.txt”).unwrap\_or\_else(|| {

Cleanup or other logic here...

});

Sometimes we only care to branch on some code depending upon the type of either the Result or Option. Normally we’d probably use a match or if-let statement to handle this, but Result and Option also provide some methods for quickly checking the type. For Result, there are the is\_ok() and is\_err() methods and for Option, the is\_some() and is\_none() methods. These return a boolean indicating whether the Result or Option were of type Ok, Err, Some, and None, respectively.

Error Propogation:

In the previous section, we explored the handling of errors using the Result enum in conjunction with match statements, as well as the unwrap() and expect() methods. This works just fine if the error is able to be handled locally, but sometimes we don’t want to handle an error locally. Sometimes, it makes more sense for the error to be returned to the calling function to be handled. This is known as error propogation. Here is an example taken from the Rust documentation:

fn read\_username\_from\_file() -> Result<String, io::Error> {

let username\_file\_result = File::open("hello.txt");

let mut username\_file = match username\_file\_result {

Ok(file) => file,

Err(e) => return Err(e),

};

let mut username = String::new();

match username\_file.read\_to\_string(&mut username) {

Ok(\_) => Ok(username),

Err(e) => Err(e),

}

}

Here we try reading a username from a file. The first match statement checks the result of opening the file and early returns the error to the callee if it failed. Otherwise, we proceed to try and find the username. Once more, if we don’t find the username, we return the error, otherwise, we return the username. Rust provides an operator for error propogation, which is the question mark operator. It is placed after any statement that returns a Result. If the subtype of Result is Err, it performs an early return with the error, otherwise, it returns the value of Ok, and we proceed as normal. Here is the same code from above using the question mark operator:

use std::io::Error as IoError;

fn read\_uname\_from\_file() -> Result<String, ioError> {

let mut uname\_file = File::open("user.txt")?;

let mut uname = String::new();

uname\_file.read\_to\_string(&mut uname)?;

Ok(uname)

}

This is off topic from error propogation, but Rust knows that opening a file and then reading it as a String is a common operation, so Rust provides us with std::fs::read\_to\_string (different than std::fs::[File::read\_to\_string](../../../../:read_to_string)). This version of read\_to\_string() combines the action of opening a file and reading the file as a string into one operation:

use std::fs;

use std::io::Error as IoError;

fn read\_uname\_from\_file() -> Result<String, IoError> {

fs::read\_to\_string(“user.txt”)

}

The Error Trait:

Sometimes we want to use a custom error type to represent points of failure in our code. A naiive approach is to create a custom struct or enum and call it a day. Error types such as std::io::Error and std::fmt::Error implement the std::error::Error trait, which provides some benefits to us over a standard struct. The Error trait is defined as follows:

pub trait Error: Debug + Display {

fn source(&self) -> Option<&(Error + 'static)> { ... }

}

Error contains a super trait which requires that we implement std::Debug and std::Display. These traits both contain a method called fmt(). std::Debug is intended for programmer-facing output, meaning that the output is intended for the programmer while the project is still in development. We can access the debug output via the :? formatter in our println! statements, or via the dbg! macro, which is similar to println!, but takes ownership of the value passed in and outputs additional context such as the line and file location. On the other hand, Display is intended for user-facing output i.e. for the project’s release build. User-facing output is the standard when printing strings created with String::from(), to\_string(), or the fmt! macro.

Due to the fact that the Error trait inherits from both Debug and Display, in order to implement it within our own struct, we must at the very least implement both Debug and Display. Here is an example, taken from <https://learning-rust.github.io/docs/custom-error-types/>:  
   
use std::fmt;

struct AppError; // Unit struct that will implement std::error::Error

impl fmt::Display for AppError {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(f, "An Error Occurred, Please Try Again!") // User-facing output

}

}

impl fmt::Debug for AppError {

fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {

write!(f, "{{ file: {}, line: {} }}", file!(), line!()) // Programmer-facing output

}

}

fn throws\_error() -> Result<(), AppError> {

Err(AppError)

}

fn main() {

match throws\_error() {

Err(e) => eprintln!("{}", e),

\_ => println!("No error"),

}

eprintln!("{:?}", throws\_error());

}

In both cases, the fmt() method takes in a Formatter struct. In order to buffer custom input into the formatter we can use the write!() macro, like in the example above. For printing the fields of a struct, we can instead use the formatter’s built-in debug\_struct() method like so:  
  
let mut dbg\_builder = f.debug\_struct(“StructName”) // Used as the header

.field(“field1”, &self.field1)

.field(“field2”, &self.field2)

.finish() // Returns a fmt::Result

In a lot of examples, you’ll see people return std::error::Error wrapped within a Result. This makes sense since Error from the std::error crate is a trait which all other Error types implement, so by using trait bounds we can return any type of Error. The std::io::Error type is probably the most useful prebuilt error type which implements std::error::Error. We can construct a new I/O Error using the new() function, which requires an ErrorKind (an enum also defined in std::io) and a message as a String or string slice. I say that this is the most useful type due to the ErrorKind enum, which contains a whole host of predefined I/O related error types. Here’s an example:

use std::error::Error;

use std::io::Error as IOError;

use std::io::ErrorKind;

fn main() -> Result<(), impl Error> {

Err(IOError::new(ErrorKind::Other, “Whoops”))

}

The Sized Trait:

All type parameters in Rust have an implicit trait bound of Sized i.e. they all must implement the Sized trait. Structs and enums implement the Sized trait by default, whereas other traits do not. Sometimes we don’t actually want this implicit trait bound, which is why you’ll sometimes see the question mark operator being used. The question mark operator only applies to the Sized trait and cannot be used for any other traits. Here is the example given by the Rust doc:

struct Foo<T>(T);

struct Bar<T: ?Sized>(T);

// struct FooUse(Foo<[i32]>); // error: Sized is not implemented for [i32]

struct BarUse(Bar<[i32]>); // OK

Recall that we can explicitly declare the size of a static array like so: [type; len]. In this example, however, we are passing a static array without specifying the length, which is normally illegal. By using the question mark operator to make Sized optional for Bar, we are then allowed to pass an object whos size is not known at compile time. This is not considered unsafe Rust because Rust will prevent us from indexing a static array whos size is not known at compile time.

The Derive Attribute:

In some of our examples, we’ve made use of the #[derive(x)] attribute, but we haven’t really discussed how it is used. Rust allows us to provide basic implementations of certain traits using this attribute, rather than needing to use a blanket implementation. The traits which are permitted to be used in the #[derive(x)] attribute are as follows: Eq, PartialEq, Ord, PartialOrd, Clone, Copy, Hash, and Debug. If we wish to implement more than one of these traits at once, we can do so by delimiting them with a comma separator e.g. #[derive(Copy, Clone)]. Of course, if you desire to override any of the methods/functions defined within these traits, then you ought to use a blanket implementation.

The Eq and PartialEq Traits:

The Eq and PartialEq traits provide functions for checking equality between two items. PartialEq is a super trait of Eq, since Eq builds on top of PartialEq. PartialEq provides the eq() and ne() methods which are mapped to the == and != operators respectively. PartialEq and Eq both require transitivity, meaning that a == b must imply b == a. This might seem obvious, however, we can override the eq() and ne() methods to check for equality between two objects which are not of the same type. In this case, PartialEq must be implemented for both types, and the comparison between both ought to provide the same result.

The Default Trait:

Default constructors are sometimes quite beneficial for creating an object with an initial set of conditions. Rust does not support constructors, as we already know, however, it does give us something known as the Default trait. The Default trait provides us with the default() function, which can, of course, be overriden. Here is an example of its usage:

#[derive(Default)]

enum Engine {

#[default]

V6,

V8

}

#[derive(Default)]

struct Car {

engine: Engine,

make: &'static str,

model: &'static str,

wheels: u8,

}

impl Car {

fn default() -> Self {

Self {

engine: Engine::V6,

make: "Toyota",

model: "Corolla",

wheels: 4

}

}

}

fn main() -> Result<(), &'static str> {

let default\_car = Car::default();

let fmt\_str = format!(

"Engine: {}\n Make: {}\n Model: {}\n Wheels: {}\n",

match default\_car.engine {

Engine::V6 => "V6",

Engine::V8 => "V8"

},

default\_car.make,

default\_car.model,

default\_car.wheels);

println!("{}", fmt\_str);

Ok(())

}

Note that since Car derives the Default trait, the Engine enum must also implement it. Assuming that all member variables belonging to the struct already implement the Default trait (all primitive types implement it), then we don’t actually need to override the default() function whatsoever.

Lifetimes:

Every reference in Rust has an implicit lifetime, which is the duration (or more precisely, the scope) for which the reference is valid. For the most part, Rust can infer the lifetime of a reference. For example, granting a reference to a variable which goes out of scope before it is able to be used would produce a dangling pointer i.e., a pointer which points to invalid data. We’ve seen this in previous examples, but here’s a refresher:

fn main() {

let r;

{

let x = 5;

r = &x;

}

println!(“r: {}”, r);

}

Of course, the Rust compiler does not approve of this, and gives us the error “borrowed value does not live long enough” on the statement r = &x;. The way that the borrow checker checks this is by using lifetimes. It recognizes that r exists in a larger scope than x, meaning that r has the bigger lifetime. The borrow checker simply compares the size of these lifetimes and determines that the lifetime of x is smaller than the lifetime of r, so the assignment is invalid. Now let’s look at a function which takes in two references and returns one of them back:

fn longest(x: &str, y: &str) -> &str {

if x.len() > y.len() {

x

} else {

y

}

}

This function compares the length of two string slices and then returns one of them back. The borrow checker takes issue with this code because is cannot determine whether x lives longer or y lives longer, since that’s dependent on what x and y are, which the compiler doesn’t know. The variable x will live longer if it is returned since y will be dropped, but y will live longer if it is returned since x will be dropped. The error you will get in this case will be something like “this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `x` or `y`”. In order to fix this, we must hint to the borrow checker that the lifetimes of x and y should be the same as that of the return value. Whenever parameters are involved in a function or method, at least one of them must be tied to the return value, as we’ll see with the lifetime elision rules. We declare explicit lifetimes in Rust with an apostraphe followed by a unique name for the lifetime. Typically, we just assign letters as the names for our lifetimes, but you can give them another name if you desire. For example, ‘a is a valid lifetime, or ‘foo is a valid lifetime. The signature of the example above must be changed to the following in order to work:

fn longest<’a>(x: &’a str, y: &’a str) -> &’a str { ... }

The lifetimes of x and y have nothing to do with how they live inside of the function, but rather how long they live outside of the function. The syntax above indicates to the borrow checker that the reference returned by this function must live for the shorter of the two parameters. Consider the following example, where we invoke the longest() function:

fn main() {

let string1 = String::from("long string is long");

{

let string2 = String::from("xyz");

let result = longest(string1.as\_str(), string2.as\_str());

println!("The longest string is {}", result);

}

}

This code will compile successfully because we’ve obeyed the lifetime rules that we specified in the longest() function. Namely, that the return value of longest() (which is stored in the variable result) will live at least as long as the smaller of the two lifetimes between string1 and string2. Remember, result will be holding a reference to either string1 or string2. The println!() statement is within the inner scope, so whether result references string1 or string2 doesn’t matter since neither of them will be dropped before result is dropped (because again, result lives for the smaller of the two lifetimes). But now consider this alternative code:

fn main() {

let string1 = String::from("long string is long");

let result;

{

let string2 = String::from("xyz");

result = longest(string1.as\_str(), string2.as\_str());

}

println!("The longest string is {}", result);

}

This does not compile, because the lifetime of result is larger than the lifetime of string2. In the example above, string1 happens to be longer, so result would contain a reference to string1, which would actually be perfectly fine. The borrow checker has no way of checking this however. It could very well be that string2 contains the longer string, in which case result would contain a dangling pointer since string2 is dropped before the println!() statement.

Now consider the case where we accept two or more references in our function, but we only actually ever return one of them:

fn longest(x: &str, y: &str) -> &str {

x

}

Rust will still expect us to supply an explicit lifetime for this function because it doesn’t parse the entire thing to find out whether only one of the parameters is returned. Basically, if there is a possibility that either x or y are returned, we must explicitly specify the lifetime(s). In this case, we indicate to the compiler that the lifetime of the return value must be equal to the lifetime of x, and that we can safely disregard y, since we know that y has no relation to the return value:

fn longest<’a>(x: &’a str, y: &str) -> &’a str {

x

}

Another common time where explicit lifetimes are required is when defining structs which contain references to other data. For example, it is easy enough to create a struct which contains a String since String is an owned value. If, however, you wanted to have a struct which contains a string slice, the compiler will throw an error:

struct Foo {

slice: &str, // missing lifetime specifier expected named lifetime parameter

}

This is because the borrowed value may be dropped prior to the instance of Foo, meaning that when we try to access the field member slice, we’re going to segfault, since slice is now a dangling reference. In most cases we know that the referenced value should live at least as long as the struct though, so this is a pretty simple fix:

struct Foo<’a> {

slice: &’a str,

}

Lifetime Elision:

Historically, it was the case that all lifetimes had to be declared explicitly. This became quite tedious, however. The Rust community eventually discovered frequent patterns that the compiler could infer regarding lifetimes, so that developers would not need to explicitly type out the lifetimes of each parameter. The current set of rules that the compiler follows to determine whether lifetimes can be implicitly defined are called the lifetime elision rules. The rules are as follows:  
  
1. Each parameter in the function/method which is a reference gets a unique lifetime specifier.

2. If the function/method only contains one reference parameter, the return value shall share the same lifetime of said parameter.

3. If operating on a method (which has &self or &mut self as its first parameter) that has multiple reference parameters, the return value shall share the same lifetime of the &self parameter.

If the compiler checks each of the lifetime elision rules and cannot determine the lifetime of the return value, then it becomes your responsibility to state the lifetime explicitly.

Smart Pointers:

Up until this point, we’ve only focused on references in Rust. References are called borrowed types in Rust, since they never own the underlying data that they refer to. Rust has support for raw pointers, like in C, though they are practically never used Rust, except for embedded programing, and even then, they must be marked as unsafe. Instead, if we require a pointer to memory within the heap, for example, we use smart pointers. There are 4 primary types of smart pointer in Rust. These include Box, Rc, Arc, and RefCell. We’ll look at each of these in detail in a bit. All smart pointers in Rust implement both the Drop and Deref traits. The Drop trait makes the implementer implement the drop() method. This method is not actually meant to be invoked manually, however, as the Rust compiler already invokes it for you once the object goes out of scope so calling it manually will result in a double free error. If you actually need to prematurely drop something e.g. a synchronization lock you should call core::mem::drop(), which will prevent a double free. Being that drop() is a core function, it’s included in the prelude, and therefore requires no import to use. The Deref trait is a bit odd, but in essence, it allows us to override the behavior of using the dereference operator. Up until now, you probably haven’t had to use the dereference operator much if at all since references are typically dereferenced for you automatically. The dereference operator is actually syntactic sugar that invokes the deref() method prior to actually dereferencing. For example, \*y expands to \*(y.deref()) in Rust. The method signature for deref() looks like the following:

fn deref(&self) -> &Self::Target { ... }

As you can see, deref() actually returns a reference to Self::Target. Target is defined within the Deref trait as a trait type. Here’s an example from the Deref trait help page:

use std::ops::Deref;

struct DerefExample<T> {

value: T

}

impl<T> Deref for DerefExample<T> {

type Target = T;

fn deref(&self) -> &Self::Target {

&self.value

}

}

let x = DerefExample { value: 'a' };

assert\_eq!('a', \*x);

The Deref trait is interesting because it allows us to do what is called Deref coercion. We’ve already breifly touched on this when I mentioned that String could be coerced into str. The reason for this is that String (as well as Vec, though that’s besides the point) are smart pointers that implement the Deref trait. Without deref coercion, to convert a String to a str, we’d have to write code like so:

fn takes\_str\_ref(s: &str) {

// Do something...

}

fn main() {

let m = MyBox::new(String::from("Rust"));

takes\_str\_ref(&(\*m)[..]);

}

I know we haven’t looked at Box just yet, but it’s coming soon after this. Without deref coercion, to get from a Box to a str, we have to first dereference the Box (\*m) to get the String, then convert it to a slice with the slice operator ([..]), and finally, get a reference to that data to pass to takes\_str\_ref(). With deref coercion, however, we can just directly pass a Box to a str reference and the compiler will invoke the deref() method to get from smart pointer to reference.

Implementing the Deref trait will only work for read-only smart pointers. If you need to dereference mutable smart pointers, implement DerefMut. The compiler performs deref coercion only in these cases:

- From &T to &U when T: Deref<Target=U>  
- From &mut T to &mut U when T: DerefMut<Target=U>  
- From &mut T to &U when T: Deref<Target=U>

Note that in the last case, we can go from a mutable reference to a non-mutable reference, but not vice versa.

Box:

Box is the simplest form of smart pointer. It simply points to some memory within the heap and drops/frees it automatically when it goes out of scope. I suppose you could say that it is the equivallent of std::unique\_ptr in C++ if you’re familiar with that. The most common use-case for Box is when dealing with data that is self-referencing or with recursive data structures. Box is included in the standard library, so we don’t need to add any use statements to use it. To create a Box, simply invoke Box::new(). The function description for Box::new() reads: “Allocates memory on the heap, then places x into it.” The variable x in this case refers to whatever we pass in as a parameter. “Boxing” a variable is the act of wrapping it in a pointer, whereas “unboxing” means dereferencing the pointer to get the original value.

Rc and Arc:

Rc and Arc are where smart pointers get really interesting. The Rc smart pointer stands for “Reference Count”. This is the equivallent of std::shared\_ptr in C++. Rc pointers differ from Box in a few ways. Primarily, Rcs allow multiple data sources to point to the same underlying memory. But wait, doesn’t this violate Rust’s rules about only having one owner per value? Yes. Rc, in fact, allows for multiple owners over the same value. This is precisely why it is not thread safe. Rc also differs from Box in that it does not call drop() when it goes out of scope unless the reference count drops to 0. For each Rc that points to the same memory on the heap, an internal counter is incremented. This is called the strong count, because it indicates how many owners share the data. When an Rc goes out of scope the counter is decremented. Once the counter reaches 0 i.e. the final Rc goes out of scope, that is when drop() is invoked. In order to use Rc, we initially use Rc::new(), similar to Box::new(). This will create a new Rc and store the value on the heap, but does not increment the strong count. Next, in order to create another pointer to the same data, we use Rc::clone() and pass in a reference to our original Rc created with Rc::new(). Rc::clone() is *not* related to the clone() method implemented by the Clone trait! Rc::clone() simply creates a new Rc which points to the same underlying data as the &Rc parameter and increments the strong count. Here is an example:

use std::rc::Rc;

fn main() -> Result<(), Box<dyn Error>> {

let a = 10;

let rc1 = Rc::new(a);

println!("{}", Rc::strong\_count(&rc1)); // Prints 1

let rc2 = Rc::clone(&rc1);

println!("{}", Rc::strong\_count(&rc1)); // Prints 2

drop(rc2);

println!("{}", Rc::strong\_count(&rc1)); // Prints 1

Ok(())

}

Arc stands for “Atomic Reference Count”. It is the thread-safe equivallent of Rc. Arc utilizes atomic operations for data access and mutation, making it much slower than regular smart pointers. On a more technical level, Arc implements the Send trait (whereas Rc does not), which is the trait that Rust uses to determine if it’s safe to send data across thread boundaries. The compiler will auto-implement the Send trait on a type whenever it determines it would be appropriate to do so.

Weak:

Akin to C++’s std::weak\_ptr, Rust has weak pointers as well. If you are not familiar with smart pointers, perhaps you were a bit confused as to why we printed out something called the “strong” count. Well this is because there is also such a thing as a weak count. Weak pointers may point to the same data that an Rc is pointing to, but it only increments the weak count, not the strong count. Since the strong count is what matters when determining if the data needs to be dropped, this means that a weak pointer can be invalidated at any point. Therefore, you must be careful when using weak pointers, ensuring that at least one Rc is still valid when accessing the underlying data throught the weak pointer. Weak pointers should be used for brief segments of code where we don’t want to affect the strong count. In order to use weak pointers, we use std::rc::Weak and rather than calling Rc::clone(), we use Rc::downgrade(). Here is the same example from the previous Rc segment, but this time, we print both the weak and strong count:  
  
fn main() -> Result<(), &'static dyn Error> {

let a = 10;

let rc1 = Rc::new(a);

println!("strong: {}", Rc::strong\_count(&rc1)); // Prints 1

println!("weak: {}", Rc::weak\_count(&rc1)); // Prints 0

let rc2 = Rc::clone(&rc1);

let weak = Rc::downgrade(&rc1);

println!("strong: {}", Rc::strong\_count(&rc1)); // Prints 2

println!("weak: {}", Rc::weak\_count(&rc1)); // Prints 1

drop(rc2);

println!("strong: {}", Rc::strong\_count(&rc1)); // Prints 1

println!("weak: {}", Rc::weak\_count(&rc1)); // Prints 1

Ok(())

}

RefCell:

RefCell is the spookiest smart pointer of them all. Rc broke one of Rust’s fundamental ownership rules; namely that values should only have one owner at a time. With Rc, we could at least rest easy rest easy knowing that we hadn’t violated any rules of the borrow checker. RefCell, however, breaks the rules of the borrow checker, namely that we may have either one mutable reference or many immutable references. But why would Rust let us break this rule? Well, in a sense, it doesn’t, the compiler still performs checks at runtime to see if we’ve violated these rules, but the issue is that in many cases, we might have code that is actually safe, and that we, the programmer, know is safe, but the compiler lacks the foreknowledge to see this. The Rust compiler is conservative by default, meaning that if it is unsure of something, it will simply default to not allowing the code to compile. It is in these scenarios that RefCell becomes useful. RefCell can be quite dangerous because it can lead to pretty cryptic bugs if we attempt to abuse it. Here is an example, though note that nothing about this code actually requires the use of RefCell:  
  
fn main() -> Result<(), Box<dyn Error>> {

let a = 10;

let r#ref = RefCell::new(a);

let mut ref\_mut = r#ref.borrow\_mut();

\*ref\_mut += 1;

println!("{}", ref\_mut); // Prints 11

Ok(())

}

To demistify a few things, the r# prefix on the ref variable is necessary since ref is a keyword in Rust so we tell the compiler to interpret it as a raw variable name. We create a new RefCell which contains the value of a and then on the next line we use the borrow\_mut() method. This method returns a RefMut, which is just a wrapper for \*mut<T> i.e. a mutable raw pointer. This is why we must dereference ref\_mut on the following line prior to incrementing it’s value. Note that a is not declared mutable in this code, and yet, since we create a RefCell containing a, we can blissfully ignore this fact. This is coined as Interior Mutability, because the value is mutated, not through a direct reference, but through a pointer to the value. A very common pattern in Rust is to encapsulate types with Rc<RefCell<T>>>. This is a powerful idiom which lets us have both multiple owners and the ability to interpret the underlying data as either immutable or mutable. Of course, it should go without saying that it is quite easy to create bugs in your code using this idiom.

Closures:

Closures are Rust’s answer to lambda functions in most other programming languages. We must be precise about the differences between closures and functions. According to the Rust documentation: “Functions coerce 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.” So function pointers have a type: fn, and can be assigned to a reference of a function. Closures, on the other hand, can be one of three types: Fn, FnOnce, or FnMut. Importantly, closures may implicitly capture variables in their enclosing scope, unlike the fn type. If the closure will be passed data by a caller from a scope that it does not have access to, the closure must accept said data as an argument rather than by capture. Unlike fn functions, closures may optionally omit explicit types for its parameters. The Rust compiler will implicitly infer the types for its parameters by looking at the initial call site and using those arguments as the basis for type inference. Here is an example from the Rust docs:

let closure = |x| x;

let s = closure(String::from(“hello”));

let n = closure(5);

This code produces an error, because on the second line (which is the initial call site for the closure) the parameter x (parameters within closures exist between two pipes (|)) is inferred to be of type String. After this point, when we try invoking closure() again with the argument of 5, it is rejected since x was already inferred to be of type String. Note also that closures do not require an explicit scope {} like fn functions do. Here is another example demonstrating the implicit borrow of an external variable within a closure:

fn main() -> Result<(), Box<dyn Error>> {

let var = String::from("Fun with closures");

let closure = || { println!("{}", var); };

closure();

println!("{}", var);

Ok(())

}

Note that we do not get an error on the second println! statement, which would normally happen if the value of var had moved into the first println statement, proving that var is being borrowed as an immutable reference here. In order to mutate an external variable, we must declare the closure as mutable (and of course also the value that were mutating):  
  
fn main() -> Result<(), Box<dyn Error>> {

let mut var = String::from("fun with closures");

let mut closure = || {

var.insert\_str(9, "rust "); // Inserts string slice “rust “ at index 9

println!("{}", var);

};

closure();

println!("{}", var);

Ok(())

}

The type of our closure (once again, it can be Fn, FnOnce, or FnMut) depends upon how we want to capture external variables. The Fn type denotes a closure which either captures external variables as immutable references, or does not capture anything at all. The FnMut type denotes a closure which will take in all externals as mutable references. The most confusing type, however, is FnOnce, which (as the name implies) may only be invoked one time (unlike Fn and FnMut which can be invoked multiple times). The trade-off here is that closures of type FnOnce can consume external variables i.e. take ownership of them. In order to implicitly create a closure of type FnOnce, we use the move keyword to indicate that all externally captured variables are moved into the closure. This is often used during thread creation:

use std::thread;

fn main() {

let v = vec![1, 2, 3];

let handle = thread::spawn(move || {

println!("Here's a vector: {:?}", v);

});

handle.join().unwrap();

}

We need to use move here so that we can pass ownership of v to the newly spawned thread.

Type Casting:

Type casting in Rust is pretty standard in comparison to other modern programming languages. You should already understand the difference between down-casting and up-casting. When up-casting, we can either use the ‘as’ keyword or the into() function to create a copy of the value with the new type. The difference between these keywords is that the as keyword requires you to explicitly declare the target type, whereas into() is inferred based upon the context (for example, if the lvalue is of a specific type in an assignment, Rust will infer that it must be cast to match that type). For down-casting, there is the try\_into() variant, which returns a Result<T, Self::Error>. The as keyword will also work for down-casting, but it will not throw an error or even a warning if bytes are discarded. Casting in Rust is possible thanks to both the From and Into traits (as well as TryFrom and TryInto for the try\_into() method). The From trait defines how other types can be cast into the type implementing the From trait, whereas the Into trait defines how the type implementing the Into trait can be cast to other types. Implementors are recommended to simply implement the From trait for your own custom types. This is because implementing From automatically implements Into. Confusingly, it also recommends using Into rather than From when declaring trait boundaries.

Parsing primitive types from Strings is a special case and may not technically be considered as type-casting, but for simplicity, I’m adding it here. In order to parse a String into another type, we have the parse() method. This method returns a Result containing the parsed value if Ok, or an Err otherwise. The parse() method in particular is usually seen with the use of what Rustaceans call the “turbofish syntax”, which looks like “::<>”. Here is an example code to analyse:

use std::io;

use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {

let mut input = String::new();

io::stdin().read\_line(&mut input)?;

let num = input.trim().parse::<i32>().unwrap();

println!("Your number was {num}");

Ok(())

}

Note the turbofish syntax I mentioned in the parse() method. We supply the type i32 to the generic type T, which parse() will then use to determine what it should try to convert the output of trim() into. This is especially useful when handling user input because the compiler won’t be able to infer the size or signedness if parsing to a numeric type.

Operator Overloading:

Similar to a language like C++, operators expand to actual functions when compiled, meaning that a + b really expands to something like a.add(b). We can overload these operator functions with traits. It’s actually really simple to do operator overloading in Rust. We just implement the correct trait from std::ops, specify a generic type as the RHS operator, specify the return type, and implement the function. This looks something like the following:

use std::ops;

struct MyFloat {

f: f32,

}

impl MyFloat {

fn new(f: f32) -> Self {

Self{ f }

}

}

impl ops::Add<MyFloat> for MyFloat {

type Output = MyFloat;

fn add(self, rhs: MyFloat) -> MyFloat {

MyFloat::new(self.f + rhs.f)

}

}

fn main() -> () {

let a = MyFloat::new(6.4);

let b = MyFloat::new(3.4);

let c = a + b;

println!("Result: {}", c.f);

}

Foreign Function Interface (FFI):

Since most operating systems are written in C it is beneficial when doing systems programming to be able to directly invoke C APIs directly in Rust rather than having to create our own versions of these APIs in Rust. Rust makes this process pretty easy by declaring a ton of wrapper functions that translate our data types into types that are compatible with C and then invoking the underlying C APIs for us. These wrapper functions begin with the letter ‘c’, but otherwise, the names match with their C counterparts e.g. sqrtf -> csqrtf. These wrapper functions must be accessed through an extern block like so:

#[link(name = “m”)]

extern {

fn csqrtf(z: Complex) -> Complex;

fn ccosf(z: Complex) -> Complex;

}

// We must wrap FFI functions in an unsafe block, so it is common   
// to create another wrapper function to avoid using unsafe repeatedly

fn cos(z: Complex) -> Complex {

unsafe { ccosf(z) }

}

Note the use of the link attribute which tells the FFI that the C implementations can be found within libm.

Macros:

There are 3 distinct types of macro in Rust. These are a) macros which are able to be used within the #[derive(x)] attribute, b) attribute-like macros that define custom attributes usable on any item, and c) function-like macros which have the ‘!’ postfix. Here we will be discussing creating custom function-like macros. We define macros in Rust with Rust, meaning that macros are a type of metaprogramming. Rust provides us with the macro\_rules! macro, which is what we use to begin a macro defintion. Here is a simplified implementation of the vec! macro, taken from the Rust docs:

#[macro\_export]

macro\_rules! vec {

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

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

$(

temp\_vec.push($x);

)\*

temp\_vec

}

}

The definition of the vec macro begins with a statement that contains syntax which we’ve not seen before. The dollar sign in the context of defining a macro indicates that we are defining a pattern with which to match. The statement $x: expr accepts any expression and assigns it to the variable $x. The comma indicates a literal comma, and the kleane star indicates that the pattern matches 0 or more of the preceding statement.

PhantomData:

PhantomData is one of those strange types in Rust that you’ll see every now and again, but probably be confused by unless you’re already familiar with it. Essentially, PhantomData is a type which consumes no memory. It is primarily used to satisfy certain conditions during lexical analysis. One such example which is given in the Rustonomicon (<https://doc.rust-lang.org/nomicon/phantom-data.html>) demonstrates the use of PhantomData for structs which contain lifetime parameters that go unused in unsafe code. As shown in the example, the Iter struct looks roughly like the following:

struct Iter<’a, T: ‘a> {

ptr \*const T,

end \*const T,

}

In this scenario, the lifetime ‘a is unbound, which (for reasons that the Rustonomicon explains) is forbidden by the compiler. We can solve this by introducing an unused field which binds to the lifetime like so:

struct Iter<’a, T: ‘a> {

ptr \*const T,

end \*const T,

\_marker: marker::PhantomData<&’a T>,

}

The cfg Attribute/Macro:

The cfg attribute and cfg! macro allow you to check for external conditions. The most common use for this attribute/macro is when writing platform-dependent code. Rust provides us with the target\_os conditional, as well as a few others, though we can also pass custom ones by using the --cfg flag when compiling. Here’s the example code given by the Rust by Example doc:

// This function only gets compiled if the target OS is linux

#[cfg(target\_os = "linux")]

fn are\_you\_on\_linux() {

println!("You are running linux!");

}

// And this function only gets compiled if the target OS is \*not\* linux

#[cfg(not(target\_os = "linux"))]

fn are\_you\_on\_linux() {

println!("You are \*not\* running linux!");

}

fn main() {

are\_you\_on\_linux();

println!("Are you sure?");

if cfg!(target\_os = "linux") {

println!("Yes. It's definitely linux!");

} else {

println!("Yes. It's definitely \*not\* linux!");

}

}

Packages, Crates, and Modules:

One of the hardest parts of Rust to master (in my opinion) is actually how it handles project structure. I’ve talked about packages and crates, but haven’t really explained what these concepts mean (though I assume you have some ideas). First, let’s discuss packages. A package is essentially synonymous with the project as a whole. A package must have a Cargo.toml file, and at least one crate. Crates come in two forms: they can either be a binary crate, or a library crate. A package can contain at most one library crate, but as many binary crates as you desire. The <project>/src/ directory is the home of what we call the “crate root”. The crate root must either be named main.rs for a binary crate, or lib.rs for a library crate. Sometimes we want to produce multiple executables for a single project, which is why we’re able to have multiple binary crates. Additional binary crates which are not the crate root must go in <project>/src/bin/, though these can be named whatever you want, unlike the crate root which must be main.rs or lib.rs. If using cargo for building (which we’ll discuss more in the next section), then you’ll need to supply the --bin option to specify which binary crate you wish to build if Rust finds more than one. Note that a project is allowed to have one or more binary crates as well as the library crate at the same time if you wish.

Now we’ll delve into modules, which are a bit more confusing. Modules are essentially like namespaces in other languages. They encapsulate functions or methods within a block to separate and organize code. Modules can be nested of course. Also, modules are by default private, unless the pub keyword is used to specify otherwise. Before I get ahead of myself, let’s look at how Rust searches for modules. The Rust docs use the example of a garden module and a vegetable module. Within the crate root, we declare the garden module with the statement mod garden;. First, the compiler checks to see if the module is declared inline i.e. if we had use curly braces after mod garden instead of the semi-colon. If not, it then searches for either src/garden.rs or src/garden/mod.rs. Here’s where it gets really strange: let’s say that we want the garden module to exist within <project>/src/backyard/garden.rs. Well, the solution that the Rust book would suggest is to make an intermittent backyard module which references the garden module, and then our crate root references the backyard module. So our directory structure now has <project>/src/backyard/garden.rs and <project>/src/backyard/mod.rs. In main.rs we refer to the backyard module with mod backyard; and in src/backyard/mod.rs, we refer to the garden module with mod garden;. To me, this seems a little silly. Instead, we can refer directly to the garden module in the crate root by declaring the backyard module inline like so:

mod backyard {

mod garden;

}

Still not ideal, but in my opinion better than making a whole file just to reference a single module.

Another facet of modules is that they are private by default, as are pretty much everything in Rust. We use the pub modifier to make them public. Just because we make a module public, however, does not make the functions within the module public. Let’s say in our garden.rs file we have two functions; plant() and harvest():

use std::error::Error;

pub fn plant() -> Result<(), ()> {

println!("Planted crops!");

Ok(())

}

pub fn harvest() -> Result<usize, Box<dyn Error>> {

let harvest\_size: usize = 543;

println!("Harvested {} plants!", harvest\_size);

Ok(harvest\_size)

}

If we try compiling this code, it will fail since in main.rs we didn’t make the garden module public. The changes look like this:

mod backyard {

pub mod garden;

}

use backyard::garden::\*;

fn main() -> () {

plant();

harvest();

}

The way of doing things that I’ve described with the garden module being declared inline in the crate root does have limitations though. Doing things the Rust recommended way, we can expose public functions to the user even if the enclosing module is not public. Doing things this way requires a bit of a rewrite. First, we remove the nested backyard/garden module declarations in main and replace them with just mod backyard;. We reintroduce the mod.rs file in the backyard directory and give it a single statement: pub mod garden;. Then in garden.rs, we declare our functions:

use std::error::Error;

// plant() definition ...

// harvest() definition ...

By doing things the way that the Rust foundation recommends, we can take advantage of the pub-use syntax, sometimes also referred to as a re-export. The ‘use’ keyword allows us to import an item into the current module/namespace. Prepending a use statement with ‘use’ will work as a normal use statement, but also re-export the module so that we can propogate its usage. For instance, our backyard module could have a use statement like so:

// mod.rs

mod garden;

use crate::backyard::garden::\*;

This would allow the backyard module to access both the plant() and harvest() functions from the garden module, however, our crate root (main.rs) would not be able to access them using the same use statement since the backyard module is private in this example. Instead, since the plant() and harvest() functions are public, we’re allowed to re-export them and propogate their usage to main.rs. Now in main.rs all we have to do is the following:

// main.rs

mod backyard;

use crate::backyard::\*;

To recap, modules are like namespaces. We declare a module with the mod keyword. The use keyword allows us to import items from another module into the current one. I also failed to mention that the ‘as’ keyword allows us to create an alias name for long module paths in conjunction with the use keyword. A crate can either be a binary crate or a library crate. We can only have one library crate per package, but multiple binary crates. The main binary crate and/or the library crate are called the crate root. The crate root for a binary crate is src/main.rs, and the crate root for a library crate is src/lib.rs. Additional binary crates are placed in src/bin/, and can be given any name, but cargo requires using the --bin option to specify which to execute. Another quick thing to note is that for structs, each individual member variable which should be accessible from a module needs to be marked pub, as well as the struct itself. For enums, however, only the enum itself needs to be marked pub.

Special Module Path Keywords:

To aid with module imports, there are a couple additional keywords and operators which can help make things simpler. The primary keywords we’ll be looking at are crate, self, and super. The crate keyword functions as an absolute path, whereas self and super act as relative paths. I’ll cover each of these individually, but first, let’s look at some operators which can help make our lives easier. When making multiple imports from the same module, it’s easier to use curly brace notation rather than redundantly repeat the same path for each item that you’re importing. For instance:

use std::io::{BufReader, BufWriter};

We can also nest this curly brace notation if necessary. Another operator which comes in handy is the kleane star, which lets you import all of the public items within a module:

use std::collections::\*;

With all that said, let’s look at those keywords I mentioned previously:

* **crate:** The crate keyword starts wherever the crate root is located, which is the src directory by default. This allows us to create absolute paths to modules e.g. use crate::path::to::crate;
* **super:** The super keyword will search for an item beginning in the parent module of the current module and then working its way up to the crate root.
* **self:** The self keyword is a bit unique. As you can guess, it expands to the current module, but you may be asking what this would be useful for. Well the answer is that it’s good in conjunction with the curly brace operator, since we can import a module, plus other specific items from that module like so:

use std::io::{self, Cursor};

This means import the std::io module, and std::io::Cursor. Note that the self keyword must be the first item in the curly braces list.

Tooling:

Understanding the Rust language will only get you about half way to becoming an efficient developer. It is crucial that you are also aware of the tooling that Rustaceans use to become an effective Rust developer. Let’s start with some of the basics. The Rust compiler is aptly named rustc. If you followed any tutorials to get Rust up and running, you likely used a tool called rustup. rustup is a toolchain installer. What this means is that you can easily switch between stable, beta, and nightly builds of the Rust toolchain using rustup. In current year (2023) many new features are being added to Rust in the nightly build and there are many Rustaceans who run off of that, so it may be of benefit to you to know how to switch from stable. Running rustup in your CLI will display all of the options for subcommands including things like listing all of the installed toolchains, checking for updates, opening documentation, and much more. The sub-command we’re most interested in is the toolchain command. Running rustup toolchain will again print a help dialog showing the various sub-sub-commands. rustup toolchain list should print out the following assuming that you have a fresh install of rustup:  
  
stable-x86\_64-unknown-linux-gnu (default)  
nightly-20XX-XX-XX-x86\_64-unknown-linux-gnu

We can switch to nightly by running rustup override set nightly.

Next, we’ll take a look at everyone’s favorite tool in Rust, cargo. Cargo is sort of a Swiss army knife in terms of what it can do. Primarily it is both a build tool and package manager. Cargo uses a single configuration file which is called Cargo.toml. toml (Tom’s Obvious Minimal Language) is a file format akin to JSON or yaml for configuring the project. We can, of course, create this file manually, but it is autogenerated for us if we build a project using cargo new <project\_name>. Alternatively, if you already have a directory which you’ve started development in that you now want to manage with cargo, you can use cargo init within the directory. Cargo build will compile your program but not run it, cargo run will both compile any modules/files that haven’t been compiled and run your program, and cargo check will simply check to see if the program will compile without actually doing a full build (this is more efficient in bigger projects where recompiling takes a long time). Some more cool stuff that we have includes cargo bench which will benchmark your code, cargo search which will query crates.io for crates given a regex pattern, cargo add which will add a crate from crates.io to your Cargo.toml file with the latest version so that the next time you build it gets installed, cargo test to run test cases, cargo publish to publish your package to the registry and cargo install/uninstall for installing/uninstalling Rust binaries. I will talk about a command I’ve intentionally left until last (cargo doc) in a moment. Cargo actually has a lot more hidden commands that are for some reason ommitted from the list when you just run the executable by itself. You can visit <https://doc.rust-lang.org/cargo/commands/> to get a full list of commands. Common ones are cargo fmt which is an auto formatter for your code which complies with the Rust convensions, as well as cargo clippy which is used to modify the linter so that it will either allow or deny lints of certain levels e.g., allowed, warnings, denied, or forbidden. One that I really like is cargo tree which lists out the dependency graph of your project so that you can see basically how bloated your project is. With that said, I now feel it appropriate to talk about cargo doc, which is, in my opinion, a really cool feature of cargo. Similar in some regards to auto-documentation tools such as Doxygen, cargo doc will generate docs for you on the fly. Unlike something like Doxygen, however, cargo doc can generate default documentation even if you don’t write any comments within your code. But wait, there’s more, because cargo doc will also generate the documentation for the dependencies within your project, so you can access them from within the docs generated for your project! This is really neat because it keeps you from having to open a browser tab for each library that you’re using in your project. In order to actually open the documentation upon it being generated, run cargo doc with the --open option. We can override the default documentation written for our functions by using comments that begin with triple slashes ‘///’ instead of the normal comment double slashes. Of course, as-is customary of auto-documentation tools, cargo doc support inline Markdown and HTML.   
  
 Of course, there are too many options and commands for me to do a full analysis of in this document, but I’ve tried to outline the main ones to give an idea of what the Rust tooling is capable of. You are tasked with learning more about the specifics of these tools and finding the ones that will enrich your developer experience.

Debugging:

My bias will shine through during this section quite a bit due to the fact that I typically use GDB to debug just about everything that I can. I’m certain that there are dedicated Rust debuggers out there, but I don’t care to discuss them. Instead, I’ll be talking about rust-gdb, which should be included as a binary when you install your distro’s gdb package.

Common Libraries:

Though Rust initially has a steep learning curve and can be quite overwhelming, you will eventually come to realize that the standard library is not as big as you might have thought. There are a couple of libraries in Rust that stand out amongst the crowd in terms of usability and the features that they offer. I will list a couple of notable ones to bring them to your attention, though you can go to crates.io for yourself, check out all crates and filter by all-time downloads and just scroll through to get an idea of what is out there.  
  
- **Serde:** Serde is a popular framework for serialization + deserealization

**- Regex:** You can probably guess what this ones for...

- **Itertools:** Provides additional iterator adaptors, methods, functions, and macros for iterating over collections that aren’t implemented in the standard library.

**- Tokio:** By far the most popular framework for performing asyncronous operations in Rust.

**- Nix:** Provides bindings to Unix/Linux APIs and system calls.

**- Winapi:** Provides binding to Windows APIs and system calls.

**- SQLX:** Provides a framework for maintaining and querying database connections.

**- Crossterm:** A cross-platform tool for manipulating terminals akin to something like ncurses.

**- Rayon:** A fantastic parallelization library which adds functions for doing stuff like parallel iterators.

**- Leptos:** A full-stack framework for creating web applications. It supports client-side rendering, server-side rendering, and server-side with hydration (meaning that individual components are re-rendered on update instead of reloading the entire DOM, similar to what React does). Uses routing for resource access.