The Rust Programming Language

2015-01-11

The Rust Programming Language

Welcome! This book will teach you about the Rust Programming Language. Rust is a modern systems programming language focusing on safety and speed. It accomplishes these goals by being memory safe without using garbage collection.

"The Rust Programming Language" is split into three sections, which you can navigate through the menu on the left.

0.0.0.1 Basics

This section is a linear introduction to the basic syntax and semantics of Rust. It has individual sections on each part of Rust's syntax, and culminates in a small project: a guessing game.

After reading "Basics," you will have a good foundation to learn more about Rust, and can write very simple programs.

0.0.0.2 Intermediate

This section contains individual chapters, which are self-contained. They focus on specific topics, and can be read in any order.

After reading "Intermediate," you will have a solid understanding of Rust, and will be able to understand most Rust code and write more complex programs.

0.0.0.3 Advanced

In a similar fashion to "Intermediate," this section is full of individual, deep-dive chapters, which stand alone and can be read in any order. These chapters focus on the most complex features, as well as some things that are only available in upcoming versions of Rust.

After reading “Advanced,” you’ll be a Rust expert!

1 The Basics

This section is a linear introduction to the basic syntax and semantics of Rust. It has individual sections on each part of Rust's syntax, and cumulates in a small project: a guessing game.

After reading "Basics," you will have a good foundation to learn more about Rust, and can write very simple programs.

1.1 Installing Rust

The first step to using Rust is to install it! There are a number of ways to install Rust, but the easiest is to use the `rustup` script. If you're on Linux or a Mac, all you need to do is this (note that you don't need to type in the `$`s, they just indicate the start of each command):

```
$ curl -L https://static.rust-lang.org/rustup.sh | sudo sh
```

If you're concerned about the potential insecurity of using `curl | sudo sh`, please keep reading and see our disclaimer below. And feel free to use a two-step version of the installation and examine our installation script:

```
$ curl -L https://static.rust-lang.org/rustup.sh -O
$ sudo sh rustup.sh
```

If you're on Windows, please download either the 32-bit installer or the 64-bit installer and run it.

If you decide you don't want Rust anymore, we'll be a bit sad, but that's okay. Not every programming language is great for everyone. Just pass an argument to the script:

```
$ curl -s https://static.rust-lang.org/rustup.sh | sudo sh -s -- --uninstall
```

If you used the Windows installer, just re-run the `.exe` and it will give you an uninstall option.

You can re-run this script any time you want to update Rust. Which, at this point, is often. Rust is still pre-1.0, and so people assume that you're using a very recent Rust.

This brings me to one other point: some people, and somewhat rightfully so, get very upset when we tell you to `curl | sudo sh`. And they should be! Basically, when you do this, you are trusting that the good people who maintain Rust aren't going to hack your computer and do bad things. That's a good instinct! If you're one of those people, please check out the documentation on building Rust from Source, or the official binary downloads. And we promise that this method will not be the way to install Rust forever: it's just the easiest way to keep people updated while Rust is in its alpha state.

Oh, we should also mention the officially supported platforms:

- Windows (7, 8, Server 2008 R2)
- Linux (2.6.18 or later, various distributions), x86 and x86-64
- OSX 10.7 (Lion) or greater, x86 and x86-64

We extensively test Rust on these platforms, and a few others, too, like Android. But these are the ones most likely to work, as they have the most testing.

Finally, a comment about Windows. Rust considers Windows to be a first-class platform upon release, but if we're honest, the Windows experience isn't as integrated as the Linux/OS X experience is. We're working on it! If anything does not work, it is a bug. Please let us know if that happens. Each and every commit is tested against Windows just like any other platform.

If you've got Rust installed, you can open up a shell, and type this:

```
$ rustc --version
```

You should see some output that looks something like this:

```
rustc 1.0.0-nightly (f11f3e7ba 2015-01-04 20:02:14 +0000)
```

If you did, Rust has been installed successfully! Congrats!

If not, there are a number of places where you can get help. The easiest is the #rust IRC channel on irc.mozilla.org, which you can access through Mibbit. Click that link, and you'll be chatting with other Rustaceans (a silly nickname we call ourselves), and we can help you out. Other great resources include our forum, the /r/rust subreddit, and Stack Overflow.

1.2 Hello, world!

Now that you have Rust installed, let's write your first Rust program. It's traditional to make your first program in any new language one that prints the text "Hello, world!" to the screen. The nice thing about starting with such a simple program is that you can verify that your compiler isn't just installed, but also working properly. And printing information to the screen is a pretty common thing to do.

The first thing that we need to do is make a file to put our code in. I like to make a `projects` directory in my home directory, and keep all my projects there. Rust does not care where your code lives.

This actually leads to one other concern we should address: this guide will assume that you have basic familiarity with the command line. Rust does not require that you know a whole ton about the command line, but until the language is in a more finished state, IDE support is spotty. Rust makes no specific demands on your editing tooling, or where your code lives.

With that said, let's make a directory in our projects directory.

```
$ mkdir ~/projects
$ cd ~/projects
$ mkdir hello_world
$ cd hello_world
```

If you're on Windows and not using PowerShell, the `~` may not work. Consult the documentation for your shell for more details.

Let's make a new source file next. I'm going to use the syntax `editor filename` to represent editing a file in these examples, but you should use whatever method you want. We'll call our file `main.rs`:

```
$ editor main.rs
```

Rust files always end in a `.rs` extension. If you're using more than one word in your filename, use an underscore. `hello_world.rs` rather than `helloworld.rs`.

Now that you've got your file open, type this in:

```
fn main() {
    println!("Hello, world!");
}
```

Save the file, and then type this into your terminal window:

```
$ rustc main.rs
$ ./main # or main.exe on Windows
Hello, world!
```

You can also run these examples on play.rust-lang.org by clicking on the arrow that appears in the upper right of the example when you mouse over the code.

Success! Let's go over what just happened in detail.

```
fn main() {

}
```

These lines define a *function* in Rust. The `main` function is special: it's the beginning of every Rust program. The first line says "I'm declaring a function named `main`, which takes no arguments and returns nothing." If there were arguments, they would go inside the parentheses (`(` and `)`), and because we aren't returning anything from this function, we've dropped that notation entirely. We'll get to it later.

You'll also note that the function is wrapped in curly braces (`{` and `}`). Rust requires these around all function bodies. It is also considered good style to put the opening curly brace on the same line as the function declaration, with one space in between.

Next up is this line:

```
    println!("Hello, world!");
```

This line does all of the work in our little program. There are a number of details that are important here. The first is that it's indented with four spaces, not tabs. Please configure your editor of choice to insert four spaces with the tab key. We provide some sample configurations for various editors.

The second point is the `println!()` part. This is calling a Rust *macro*, which is how metaprogramming is done in Rust. If it were a function instead, it would look like this: `println()`. For our purposes, we don't need to worry about this difference. Just know that sometimes, you'll see a `!`, and that means that you're calling a macro instead of a normal function. Rust implements `println!` as a macro rather than a function for good reasons, but that's a very advanced topic. You'll learn more when we talk about macros later. One last thing to mention: Rust's macros are significantly different from C macros, if you've used those. Don't be scared of using macros. We'll get to the details eventually, you'll just have to trust us for now.

Next, `"Hello, world!"` is a *string*. Strings are a surprisingly complicated topic in a systems programming language, and this is a *statically allocated* string. We will talk more about different kinds of allocation later. We pass this string as an argument to `println!`, which prints the string to the screen. Easy enough!

Finally, the line ends with a semicolon (`;`). Rust is an *expression oriented* language, which means that most things are expressions. The `;` is used to indicate that this expression is over, and the next one is ready to begin. Most lines of Rust code end with a `;`. We will cover this in-depth later in the guide.

Finally, actually *compiling* and *running* our program. We can compile with our compiler, `rustc`, by passing it the name of our source file:

```
$ rustc main.rs
```

This is similar to `gcc` or `clang`, if you come from a C or C++ background. Rust will output a binary executable. You can see it with `ls`:

```
$ ls
main  main.rs
```

Or on Windows:

```
$ dir
main.exe  main.rs
```

There are now two files: our source code, with the `.rs` extension, and the executable (`main.exe` on Windows, `main` everywhere else)

```
$ ./main  # or main.exe on Windows
```

This prints out our `Hello, world!` text to our terminal.

If you come from a dynamically typed language like Ruby, Python, or JavaScript, you may not be used to these two steps being separate. Rust is an *ahead-of-time compiled language*, which means that you can compile a program, give it to someone else, and they don't need to have Rust installed. If you give someone a `.rb` or `.py` or `.js` file, they need to have Ruby/Python/JavaScript installed, but you just need one command to both compile and run your program. Everything is a tradeoff in language design, and Rust has made its choice.

Congratulations! You have officially written a Rust program. That makes you a Rust programmer! Welcome.

Next, I'd like to introduce you to another tool, Cargo, which is used to write real-world Rust programs. Just using `rustc` is nice for simple things, but as your project grows, you'll want something to help you manage all of the options that it has, and to make it easy to share your code with other people and projects.

1.3 Hello, Cargo!

Cargo is a tool that Rustaceans use to help manage their Rust projects. Cargo is currently in an alpha state, just like Rust, and so it is still a work in progress. However, it is already good enough to use for many Rust projects, and so it is assumed that Rust projects will use Cargo from the beginning.

Cargo manages three things: building your code, downloading the dependencies your code needs, and building those dependencies. At first, your program doesn't have any dependencies, so we'll only be using the first part of its functionality. Eventually, we'll add more. Since we started off by using Cargo, it'll be easy to add later.

If you installed Rust via the official installers you will also have Cargo. If you installed Rust some other way, you may want to check the Cargo README for specific instructions about installing it.

Let's convert Hello World to Cargo.

To Cargo-ify our project, we need to do two things: Make a `Cargo.toml` configuration file, and put our source file in the right place. Let's do that part first:

```
$ mkdir src
$ mv main.rs src/main.rs
```

Cargo expects your source files to live inside a `src` directory. That leaves the top level for other things, like READMEs, license information, and anything not related to your code. Cargo helps us keep our projects nice and tidy. A place for everything, and everything in its place.

Next, our configuration file:

```
$ editor Cargo.toml
```

Make sure to get this name right: you need the capital `C`!

Put this inside:

```
[package]

name = "hello_world"
version = "0.0.1"
authors = [ "Your name <you@example.com>" ]

[[bin]]

name = "hello_world"
```

This file is in the TOML format. Let's let it explain itself to you:

> TOML aims to be a minimal configuration file format that's easy to read due to obvious semantics. TOML is designed to map unambiguously to a hash table. TOML should be easy to parse into data structures in a wide variety of languages.

TOML is very similar to INI, but with some extra goodies.

Anyway, there are two *tables* in this file: `package` and `bin`. The first tells Cargo metadata about your package. The second tells Cargo that we're interested in building a binary, not a library (though we could do both!), as well as what it is named.

Once you have this file in place, we should be ready to build! Try this:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/yourname/projects/hello_world)
$ ./target/hello_world
Hello, world!
```

Bam! We build our project with `cargo build`, and run it with `./target/hello_world`. This hasn't bought us a whole lot over our simple use of `rustc`, but think about the future: when our project has more than one file, we would need to call `rustc` more than once, and pass it a bunch of options to tell it to build everything together. With Cargo, as our project grows, we can just `cargo build` and it'll work the right way.

You'll also notice that Cargo has created a new file: `Cargo.lock`.

```
[root]
name = "hello_world"
version = "0.0.1"
```

This file is used by Cargo to keep track of dependencies in your application. Right now, we don't have any, so it's a bit sparse. You won't ever need to touch this file yourself, just let Cargo handle it.

That's it! We've successfully built `hello_world` with Cargo. Even though our program is simple, it's using much of the real tooling that you'll use for the rest of your Rust career.

Now that you've got the tools down, let's actually learn more about the Rust language itself. These are the basics that will serve you well through the rest of your time with Rust.

1.4 Variable Bindings

The first thing we'll learn about are *variable bindings*. They look like this:

```
fn main() {
    let x = 5;
}
```

Putting `fn main() {` in each example is a bit tedious, so we'll leave that out in the future. If you're following along, make sure to edit your `main()` function, rather than leaving it off. Otherwise, you'll get an error.

In many languages, this is called a *variable*. But Rust's variable bindings have a few tricks up their sleeves. Rust has a very powerful feature called *pattern matching* that we'll get into detail with later, but the left hand side of a `let` expression is a full pattern, not just a variable name. This means we can do things like:

```
let (x, y) = (1, 2);
```

After this expression is evaluated, `x` will be one, and `y` will be two. Patterns are really powerful, but this is about all we can do with them so far. So let's just keep this in the back of our minds as we go forward.

Rust is a statically typed language, which means that we specify our types up front. So why does our first example compile? Well, Rust has this thing called *type inference*. If it can figure out what the type of something is, Rust doesn't require you to actually type it out.

We can add the type if we want to, though. Types come after a colon (`:`):

```
let x: i32 = 5;
```

If I asked you to read this out loud to the rest of the class, you'd say "`x` is a binding with the type `i32` and the value `five`."

In future examples, we may annotate the type in a comment. The examples will look like this:

```
fn main() {
    let x = 5; // x: i32
}
```

Note the similarities between this annotation and the syntax you use with `let`. Including these kinds of comments is not idiomatic Rust, but we'll occasionally include them to help you understand what the types that Rust infers are.

By default, bindings are *immutable*. This code will not compile:

```
let x = 5;
x = 10;
```

It will give you this error:

```
error: re-assignment of immutable variable `x`
     x = 10;
     ^~~~~~
```

If you want a binding to be mutable, you can use `mut`:

```
let mut x = 5; // mut x: i32
x = 10;
```

There is no single reason that bindings are immutable by default, but we can think about it through one of Rust's primary focuses: safety. If you forget to say `mut`, the compiler will catch it, and let you know that you have mutated something you may not have intended to mutate. If bindings were mutable by default, the compiler would not be able to tell you this. If you *did* intend mutation, then the solution is quite easy: add `mut`.

There are other good reasons to avoid mutable state when possible, but they're out of the scope of this guide. In general, you can often avoid explicit mutation, and so it is preferable in Rust. That said, sometimes, mutation is what you need, so it's not verboten.

Let's get back to bindings. Rust variable bindings have one more aspect that differs from other languages: bindings are required to be initialized with a value before you're allowed to use them. If we try…

```
let x;
```

…we'll get an error:

```
src/main.rs:2:9: 2:10 error: cannot determine a type for this local variable: unconstrained type
src/main.rs:2     let x;
                      ^
```

Giving it a type will compile, though:

```
let x: i32;
```

Let's try it out. Change your `src/main.rs` file to look like this:

```
fn main() {
    let x: i32;
```

```
    println!("Hello world!");
}
```

You can use `cargo build` on the command line to build it. You'll get a warning, but it will still print "Hello, world!":

```
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:2:9: 2:10 warning: unused variable: `x`, #[warn(unused_variable)] on by default
src/main.rs:2     let x: i32;
                      ^
```

Rust warns us that we never use the variable binding, but since we never use it, no harm, no foul. Things change if we try to actually use this `x`, however. Let's do that. Change your program to look like this:

```
fn main() {
    let x: i32;

    println!("The value of x is: {}", x);
}
```

And try to build it. You'll get an error:

```
$ cargo build
   Compiling hello_world v0.0.1 (file:///home/you/projects/hello_world)
src/main.rs:4:39: 4:40 error: use of possibly uninitialized variable: `x`
src/main.rs:4     println!("The value of x is: {}", x);
                                                    ^
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
src/main.rs:4:5: 4:42 note: expansion site
error: aborting due to previous error
Could not compile `hello_world`.
```

Rust will not let us use a value that has not been initialized. Next, let's talk about this stuff we've added to `println!`.

If you include two curly braces (`{}`, some call them moustaches…) in your string to print, Rust will interpret this as a request to interpolate some sort of value. *String interpolation* is a computer science term that means "stick in the middle of a string." We add a comma, and then `x`, to indicate that we want `x` to be the value we're interpolating. The comma is used to separate arguments we pass to functions and macros, if you're passing more than one.

When you just use the curly braces, Rust will attempt to display the value in a meaningful way by checking out its type. If you want to specify the format in a more detailed manner, there are a wide number of options available. For now, we'll just stick to the default: integers aren't very complicated to print.

1.5 If

Rust's take on `if` is not particularly complex, but it's much more like the `if` you'll find in a dynamically typed language than in a more traditional systems language. So let's talk about it, to make sure you grasp the nuances.

`if` is a specific form of a more general concept, the *branch*. The name comes from a branch in a tree: a decision point, where depending on a choice, multiple paths can be taken.

In the case of `if`, there is one choice that leads down two paths:

```
let x = 5;

if x == 5 {
    println!("x is five!");
}
```

If we changed the value of `x` to something else, this line would not print. More specifically, if the expression after the `if` evaluates to `true`, then the block is executed. If it's `false`, then it is not.

If you want something to happen in the `false` case, use an `else`:

```
let x = 5;

if x == 5 {
    println!("x is five!");
} else {
    println!("x is not five :(");
}
```

This is all pretty standard. However, you can also do this:

```
let x = 5;

let y = if x == 5 {
    10
} else {
    15
}; // y: i32
```

Which we can (and probably should) write like this:

```
let x = 5;

let y = if x == 5 { 10 } else { 15 }; // y: i32
```

This reveals two interesting things about Rust: it is an expression-based language, and semicolons are different from semicolons in other 'curly brace and semicolon'-based languages. These two things are related.

1.5.0.1 Expressions vs. Statements

Rust is primarily an expression based language. There are only two kinds of statements, and everything else is an expression.

So what's the difference? Expressions return a value, and statements do not. In many languages, `if` is a statement, and therefore, `let x = if ...` would make no sense. But in Rust, `if` is an expression, which means that it returns a value. We can then use this value to initialize the binding.

Speaking of which, bindings are a kind of the first of Rust's two statements. The proper name is a *declaration statement*. So far, `let` is the only kind of declaration statement we've seen. Let's talk about that some more.

In some languages, variable bindings can be written as expressions, not just statements. Like Ruby:

```
x = y = 5
```

In Rust, however, using `let` to introduce a binding is *not* an expression. The following will produce a compile-time error:

```
let x = (let y = 5); // expected identifier, found keyword `let`
```

The compiler is telling us here that it was expecting to see the beginning of an expression, and a `let` can only begin a statement, not an expression.

Note that assigning to an already-bound variable (e.g. `y = 5`) is still an expression, although its value is not particularly useful. Unlike C, where an assignment evaluates to the assigned value (e.g. `5` in the previous example), in Rust the value of an assignment is the unit type `()` (which we'll cover later).

The second kind of statement in Rust is the *expression statement*. Its purpose is to turn any expression into a statement. In practical terms, Rust's grammar expects statements to follow other statements. This means that you

use semicolons to separate expressions from each other. This means that Rust looks a lot like most other languages that require you to use semicolons at the end of every line, and you will see semicolons at the end of almost every line of Rust code you see.

What is this exception that makes us say "almost"? You saw it already, in this code:

```
let x = 5;

let y: i32 = if x == 5 { 10 } else { 15 };
```

Note that I've added the type annotation to `y`, to specify explicitly that I want `y` to be an integer.

This is not the same as this, which won't compile:

```
let x = 5;

let y: i32 = if x == 5 { 10; } else { 15; };
```

Note the semicolons after the 10 and 15. Rust will give us the following error:

```
error: mismatched types: expected `i32` but found `()` (expected i32 but found ())
```

We expected an integer, but we got `()`. `()` is pronounced *unit*, and is a special type in Rust's type system. In Rust, `()` is *not* a valid value for a variable of type `i32`. It's only a valid value for variables of the type `()`, which aren't very useful. Remember how we said statements don't return a value? Well, that's the purpose of unit in this case. The semicolon turns any expression into a statement by throwing away its value and returning unit instead.

There's one more time in which you won't see a semicolon at the end of a line of Rust code. For that, we'll need our next concept: functions.

1.6 Functions

You've already seen one function so far, the `main` function:

```
fn main() {
}
```

This is the simplest possible function declaration. As we mentioned before, `fn` says "this is a function," followed by the name, some parentheses because this function takes no arguments, and then some curly braces to indicate the body. Here's a function named `foo`:

```
fn foo() {
}
```

So, what about taking arguments? Here's a function that prints a number:

```
fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

Here's a complete program that uses `print_number`:

```
fn main() {
    print_number(5);
}

fn print_number(x: i32) {
    println!("x is: {}", x);
}
```

As you can see, function arguments work very similar to `let` declarations: you add a type to the argument name, after a colon.

Here's a complete program that adds two numbers together and prints them:

```
fn main() {
    print_sum(5, 6);
}

fn print_sum(x: i32, y: i32) {
    println!("sum is: {}", x + y);
}
```

You separate arguments with a comma, both when you call the function, as well as when you declare it.

Unlike `let`, you *must* declare the types of function arguments. This does not work:

```
fn print_number(x, y) {
    println!("x is: {}", x + y);
}
```

You get this error:

```
hello.rs:5:18: 5:19 error: expected `:` but found `,`
hello.rs:5 fn print_number(x, y) {
```

This is a deliberate design decision. While full-program inference is possible, languages which have it, like Haskell, often suggest that documenting your types explicitly is a best-practice. We agree that forcing functions to declare types while allowing for inference inside of function bodies is a wonderful sweet spot between full inference and no inference.

What about returning a value? Here's a function that adds one to an integer:

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

Rust functions return exactly one value, and you declare the type after an "arrow," which is a dash (`-`) followed by a greater-than sign (`>`).

You'll note the lack of a semicolon here. If we added it in:

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

We would get an error:

```
error: not all control paths return a value
fn add_one(x: i32) -> i32 {
     x + 1;
}

help: consider removing this semicolon:
     x + 1;
          ^
```

Remember our earlier discussions about semicolons and `()`? Our function claims to return an `i32`, but with a semicolon, it would return `()` instead. Rust realizes this probably isn't what we want, and suggests removing the semicolon.

This is very much like our `if` statement before: the result of the block (`{}`) is the value of the expression. Other expression-oriented languages, such as Ruby, work like this, but it's a bit unusual in the systems programming world. When people first learn about this, they usually assume that it introduces bugs. But because Rust's type system is so strong, and because unit is its own unique type, we have never seen an issue where adding or removing a semicolon in a return position would cause a bug.

But what about early returns? Rust does have a keyword for that, `return`:

```
fn foo(x: i32) -> i32 {
    if x < 5 { return x; }

    x + 1
}
```

Using a `return` as the last line of a function works, but is considered poor style:

```
fn foo(x: i32) -> i32 {
    if x < 5 { return x; }

    return x + 1;
}
```

There are some additional ways to define functions, but they involve features that we haven't learned about yet, so let's just leave it at that for now.

1.7 Comments

Now that we have some functions, it's a good idea to learn about comments. Comments are notes that you leave to other programmers to help explain things about your code. The compiler mostly ignores them.

Rust has two kinds of comments that you should care about: *line comments* and *doc comments*.

```
// Line comments are anything after '//' and extend to the end of the line.

let x = 5; // this is also a line comment.

// If you have a long explanation for something, you can put line comments next
// to each other. Put a space between the // and your comment so that it's
// more readable.
```

The other kind of comment is a doc comment. Doc comments use `///` instead of `//`, and support Markdown notation inside:

```
/// `hello` is a function that prints a greeting that is personalized based on
/// the name given.
///
/// # Arguments
///
/// * `name` - The name of the person you'd like to greet.
///
/// # Example
///
/// ```rust
/// let name = "Steve";
/// hello(name); // prints "Hello, Steve!"
/// ```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

When writing doc comments, adding sections for any arguments, return values, and providing some examples of usage is very, very helpful.

You can use the `rustdoc` tool to generate HTML documentation from these doc comments.

1.8 Compound Data Types

Rust, like many programming languages, has a number of different data types that are built-in. You've already done some simple work with integers and strings, but next, let's talk about some more complicated ways of storing

data.

1.8.0.1 Tuples

The first compound data type we're going to talk about are called *tuples*. Tuples are an ordered list of a fixed size. Like this:

```
let x = (1, "hello");
```

The parentheses and commas form this two-length tuple. Here's the same code, but with the type annotated:

```
let x: (i32, &str) = (1, "hello");
```

As you can see, the type of a tuple looks just like the tuple, but with each position having a type name rather than the value. Careful readers will also note that tuples are heterogeneous: we have an `i32` and a `&str` in this tuple. You haven't seen `&str` as a type before, and we'll discuss the details of strings later. In systems programming languages, strings are a bit more complex than in other languages. For now, just read `&str` as a *string slice*, and we'll learn more soon.

You can access the fields in a tuple through a *destructuring let*. Here's an example:

```
let (x, y, z) = (1, 2, 3);

println!("x is {}", x);
```

Remember before when I said the left-hand side of a `let` statement was more powerful than just assigning a binding? Here we are. We can put a pattern on the left-hand side of the `let`, and if it matches up to the right-hand side, we can assign multiple bindings at once. In this case, `let` "destructures," or "breaks up," the tuple, and assigns the bits to three bindings.

This pattern is very powerful, and we'll see it repeated more later.

There are also a few things you can do with a tuple as a whole, without destructuring. You can assign one tuple into another, if they have the same arity and contained types.

```
let mut x = (1, 2); // x: (i32, i32)
let y = (2, 3);     // y: (i32, i32)

x = y;
```

You can also check for equality with ==. Again, this will only compile if the tuples have the same type.

```
let x = (1, 2, 3);
let y = (2, 2, 4);

if x == y {
    println!("yes");
} else {
    println!("no");
}
```

This will print `no`, because some of the values aren't equal.

One other use of tuples is to return multiple values from a function:

```
fn next_two(x: i32) -> (i32, i32) { (x + 1, x + 2) }

fn main() {
    let (x, y) = next_two(5);
    println!("x, y = {}, {}", x, y);
}
```

Even though Rust functions can only return one value, a tuple *is* one value, that happens to be made up of more than one value. You can also see in this example how you can destructure a pattern returned by a function, as well.

Tuples are a very simple data structure, and so are not often what you want. Let's move on to their bigger sibling, structs.

1.8.0.2 Structs

A struct is another form of a *record type*, just like a tuple. There's a difference: structs give each element that they contain a name, called a *field* or a *member*. Check it out:

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

fn main() {
    let origin = Point { x: 0, y: 0 }; // origin: Point

    println!("The origin is at ({}, {})", origin.x, origin.y);
}
```

There's a lot going on here, so let's break it down. We declare a struct with the `struct` keyword, and then with a name. By convention, structs begin with a capital letter and are also camel cased: `PointInSpace`, not `Point_In_Space`.

We can create an instance of our struct via `let`, as usual, but we use a `key: value` style syntax to set each field. The order doesn't need to be the same as in the original declaration.

Finally, because fields have names, we can access the field through dot notation: `origin.x`.

The values in structs are immutable by default, like other bindings in Rust. Use `mut` to make them mutable:

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

fn main() {
    let mut point = Point { x: 0, y: 0 };

    point.x = 5;

    println!("The point is at ({}, {})", point.x, point.y);
}
```

This will print `The point is at (5, 0)`.

1.8.0.3 Tuple Structs and Newtypes

Rust has another data type that's like a hybrid between a tuple and a struct, called a *tuple struct*. Tuple structs do have a name, but their fields don't:

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

These two will not be equal, even if they have the same values:

```
# struct Color(i32, i32, i32);
# struct Point(i32, i32, i32);
let black  = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

It is almost always better to use a struct than a tuple struct. We would write `Color` and `Point` like this instead:

```
struct Color {
    red: i32,
    blue: i32,
    green: i32,
}

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

Now, we have actual names, rather than positions. Good names are important, and with a struct, we have actual names.

There *is* one case when a tuple struct is very useful, though, and that's a tuple struct with only one element. We call this a *newtype*, because it lets you create a new type that's a synonym for another one:

```
struct Inches(i32);

let length = Inches(10);

let Inches(integer_length) = length;
println!("length is {} inches", integer_length);
```

As you can see here, you can extract the inner integer type through a destructuring `let`.

1.8.0.4 Enums

Finally, Rust has a "sum type", an *enum*. Enums are an incredibly useful feature of Rust, and are used throughout the standard library. This is an enum that is provided by the Rust standard library:

```
enum Ordering {
    Less,
    Equal,
    Greater,
}
```

An `Ordering` can only be *one* of `Less`, `Equal`, or `Greater` at any given time.

Because `Ordering` is provided by the standard library, we can use the `use` keyword to use it in our code. We'll learn more about `use` later, but it's used to bring names into scope.

Here's an example of how to use `Ordering`:

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    let ordering = cmp(x, y); // ordering: Ordering

    if ordering == Ordering::Less {
        println!("less");
    } else if ordering == Ordering::Greater {
        println!("greater");
    } else if ordering == Ordering::Equal {
```

```
        println!("equal");
    }
}
```

There's a symbol here we haven't seen before: the double colon (`::`). This is used to indicate a namespace. In this case, `Ordering` lives in the `cmp` submodule of the `std` module. We'll talk more about modules later in the guide. For now, all you need to know is that you can `use` things from the standard library if you need them.

Okay, let's talk about the actual code in the example. `cmp` is a function that compares two things, and returns an `Ordering`. We return either `Ordering::Less`, `Ordering::Greater`, or `Ordering::Equal`, depending on if the two values are greater, less, or equal. Note that each variant of the `enum` is namespaced under the `enum` itself: it's `Ordering::Greater` not `Greater`.

The `ordering` variable has the type `Ordering`, and so contains one of the three values. We can then do a bunch of `if`/`else` comparisons to check which one it is. However, repeated `if`/`else` comparisons get quite tedious. Rust has a feature that not only makes them nicer to read, but also makes sure that you never miss a case. Before we get to that, though, let's talk about another kind of enum: one with values.

This enum has two variants, one of which has a value:

```
enum OptionalInt {
    Value(i32),
    Missing,
}
```

This enum represents an `i32` that we may or may not have. In the `Missing` case, we have no value, but in the `Value` case, we do. This enum is specific to `i32`s, though. We can make it usable by any type, but we haven't quite gotten there yet!

You can also have any number of values in an enum:

```
enum OptionalColor {
    Color(i32, i32, i32),
    Missing,
}
```

And you can also have something like this:

```
enum StringResult {
    StringOK(String),
    ErrorReason(String),
}
```

Where a `StringResult` is either a `StringResult::StringOK`, with the result of a computation, or an `StringResult::ErrorReason` with a `String` explaining what caused the computation to fail. These kinds of `enum`s are actually very useful and are even part of the standard library.

Here is an example of using our `StringResult`:

```
enum StringResult {
    StringOK(String),
    ErrorReason(String),
}

fn respond(greeting: &str) -> StringResult {
    if greeting == "Hello" {
        StringResult::StringOK("Good morning!".to_string())
    } else {
        StringResult::ErrorReason("I didn't understand you!".to_string())
    }
}
```

That's a lot of typing! We can use the `use` keyword to make it shorter:

```
use StringResult::StringOK;
use StringResult::ErrorReason;

enum StringResult {
    StringOK(String),
    ErrorReason(String),
}

# fn main() {}

fn respond(greeting: &str) -> StringResult {
    if greeting == "Hello" {
        StringOK("Good morning!".to_string())
    } else {
        ErrorReason("I didn't understand you!".to_string())
    }
}
```

`use` declarations must come before anything else, which looks a little strange in this example, since we `use` the variants before we define them. Anyway, in the body of `respond`, we can just say `StringOK` now, rather than the full `StringResult::StringOK`. Importing variants can be convenient, but can also cause name conflicts, so do this with caution. It's considered good style to rarely import variants for this reason.

As you can see, `enum`s with values are quite a powerful tool for data representation, and can be even more useful when they're generic across types. Before we get to generics, though, let's talk about how to use them with pattern matching, a tool that will let us deconstruct this sum type (the type theory term for enums) in a very elegant way and avoid all these messy `if`/`else`s.

1.9 Match

Often, a simple `if`/`else` isn't enough, because you have more than two possible options. Also, `else` conditions can get incredibly complicated, so what's the solution?

Rust has a keyword, `match`, that allows you to replace complicated `if`/`else` groupings with something more powerful. Check it out:

```
let x = 5;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    4 => println!("four"),
    5 => println!("five"),
    _ => println!("something else"),
}
```

`match` takes an expression and then branches based on its value. Each *arm* of the branch is of the form `val => expression`. When the value matches, that arm's expression will be evaluated. It's called `match` because of the term 'pattern matching', which `match` is an implementation of.

So what's the big advantage here? Well, there are a few. First of all, `match` enforces *exhaustiveness checking*. Do you see that last arm, the one with the underscore (_)? If we remove that arm, Rust will give us an error:

```
error: non-exhaustive patterns: `_` not covered
```

In other words, Rust is trying to tell us we forgot a value. Because `x` is an integer, Rust knows that it can have a number of different values – for example, `6`. Without the _, however, there is no arm that could match, and so Rust refuses to compile. _ acts like a *catch-all arm*. If none of the other arms match, the arm with _ will, and since we have this catch-all arm, we now have an arm for every possible value of `x`, and so our program will compile successfully.

`match` statements also destructure enums, as well. Remember this code from the section on enums?

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    let ordering = cmp(x, y);

    if ordering == Ordering::Less {
        println!("less");
    } else if ordering == Ordering::Greater {
        println!("greater");
    } else if ordering == Ordering::Equal {
        println!("equal");
    }
}
```

We can re-write this as a `match`:

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    match cmp(x, y) {
        Ordering::Less    => println!("less"),
        Ordering::Greater => println!("greater"),
        Ordering::Equal   => println!("equal"),
    }
}
```

This version has way less noise, and it also checks exhaustively to make sure that we have covered all possible variants of `Ordering`. With our `if`/`else` version, if we had forgotten the `Greater` case, for example, our program would have happily compiled. If we forget in the `match`, it will not. Rust helps us make sure to cover all of our bases.

`match` expressions also allow us to get the values contained in an `enum` (also known as destructuring) as follows:

```
enum OptionalInt {
    Value(i32),
    Missing,
}

fn main() {
    let x = OptionalInt::Value(5);
    let y = OptionalInt::Missing;

    match x {
        OptionalInt::Value(n) => println!("x is {}", n),
        OptionalInt::Missing  => println!("x is missing!"),
    }

    match y {
        OptionalInt::Value(n) => println!("y is {}", n),
        OptionalInt::Missing  => println!("y is missing!"),
```

```
    }
}
```

That is how you can get and use the values contained in `enum`s. It can also allow us to handle errors or unexpected computations; for example, a function that is not guaranteed to be able to compute a result (an `i32` here) could return an `OptionalInt`, and we would handle that value with a `match`. As you can see, `enum` and `match` used together are quite useful!

`match` is also an expression, which means we can use it on the right-hand side of a `let` binding or directly where an expression is used. We could also implement the previous example like this:

```
use std::cmp::Ordering;

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}

fn main() {
    let x = 5;
    let y = 10;

    println!("{}", match cmp(x, y) {
        Ordering::Less    => "less",
        Ordering::Greater => "greater",
        Ordering::Equal   => "equal",
    });
}
```

Sometimes, it's a nice pattern.

1.10 Looping

Looping is the last basic construct that we haven't learned yet in Rust. Rust has two main looping constructs: `for` and `while`.

1.10.0.1 `for`

The `for` loop is used to loop a particular number of times. Rust's `for` loops work a bit differently than in other systems languages, however. Rust's `for` loop doesn't look like this "C-style" `for` loop:

```
for (x = 0; x < 10; x++) {
    printf( "%d\n", x );
}
```

Instead, it looks like this:

```
for x in range(0, 10) {
    println!("{}", x); // x: i32
}
```

In slightly more abstract terms,

```
for var in expression {
    code
}
```

The expression is an iterator, which we will discuss in more depth later in the guide. The iterator gives back a series of elements. Each element is one iteration of the loop. That value is then bound to the name `var`, which is valid for the loop body. Once the body is over, the next value is fetched from the iterator, and we loop another time. When there are no more values, the `for` loop is over.

In our example, `range` is a function that takes a start and an end position, and gives an iterator over those values. The upper bound is exclusive, though, so our loop will print `0` through `9`, not `10`.

Rust does not have the "C-style" `for` loop on purpose. Manually controlling each element of the loop is complicated and error prone, even for experienced C developers.

We'll talk more about `for` when we cover *iterators*, later in the Guide.

1.10.0.2 `while`

The other kind of looping construct in Rust is the `while` loop. It looks like this:

```
let mut x = 5u;       // mut x: uint
let mut done = false; // mut done: bool

while !done {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { done = true; }
}
```

`while` loops are the correct choice when you're not sure how many times you need to loop.

If you need an infinite loop, you may be tempted to write this:

```
while true {
```

However, Rust has a dedicated keyword, `loop`, to handle this case:

```
loop {
```

Rust's control-flow analysis treats this construct differently than a `while true`, since we know that it will always loop. The details of what that *means* aren't super important to understand at this stage, but in general, the more information we can give to the compiler, the better it can do with safety and code generation, so you should always prefer `loop` when you plan to loop infinitely.

1.10.0.3 Ending iteration early

Let's take a look at that `while` loop we had earlier:

```
let mut x = 5u;
let mut done = false;

while !done {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { done = true; }
}
```

We had to keep a dedicated `mut` boolean variable binding, `done`, to know when we should exit out of the loop. Rust has two keywords to help us with modifying iteration: `break` and `continue`.

In this case, we can write the loop in a better way with `break`:

```
let mut x = 5u;

loop {
    x += x - 3;
    println!("{}", x);
    if x % 5 == 0 { break; }
}
```

We now loop forever with `loop` and use `break` to break out early.

`continue` is similar, but instead of ending the loop, goes to the next iteration. This will only print the odd numbers:

```
for x in range(0, 10) {
    if x % 2 == 0 { continue; }

    println!("{}", x);
}
```

Both `continue` and `break` are valid in both kinds of loops.

1.11 Strings

Strings are an important concept for any programmer to master. Rust's string handling system is a bit different from other languages, due to its systems focus. Any time you have a data structure of variable size, things can get tricky, and strings are a re-sizable data structure. That being said, Rust's strings also work differently than in some other systems languages, such as C.

Let's dig into the details. A *string* is a sequence of Unicode scalar values encoded as a stream of UTF-8 bytes. All strings are guaranteed to be validly encoded UTF-8 sequences. Additionally, strings are not null-terminated and can contain null bytes.

Rust has two main types of strings: `&str` and `String`.

The first kind is a `&str`. These are called *string slices*. String literals are of the type `&str`:

```
let string = "Hello there."; // string: &str
```

This string is statically allocated, meaning that it's saved inside our compiled program, and exists for the entire duration it runs. The `string` binding is a reference to this statically allocated string. String slices have a fixed size, and cannot be mutated.

A `String`, on the other hand, is an in-memory string. This string is growable, and is also guaranteed to be UTF-8.

```
let mut s = "Hello".to_string(); // mut s: String
println!("{}", s);

s.push_str(", world.");
println!("{}", s);
```

You can get a `&str` view into a `String` with the `as_slice()` method:

```
fn takes_slice(slice: &str) {
    println!("Got: {}", slice);
}

fn main() {
    let s = "Hello".to_string();
    takes_slice(s.as_slice());
}
```

To compare a String to a constant string, prefer `as_slice()`...

```
fn compare(string: String) {
    if string.as_slice() == "Hello" {
        println!("yes");
    }
}
```

... over `to_string()`:

```
fn compare(string: String) {
    if string == "Hello".to_string() {
        println!("yes");
    }
}
```

Viewing a `String` as a `&str` is cheap, but converting the `&str` to a `String` involves allocating memory. No reason to do that unless you have to!

That's the basics of strings in Rust! They're probably a bit more complicated than you are used to, if you come from a scripting language, but when the low-level details matter, they really matter. Just remember that `String`s allocate memory and control their data, while `&str`s are a reference to another string, and you'll be all set.

1.12 Arrays, Vectors, and Slices

Like many programming languages, Rust has list types to represent a sequence of things. The most basic is the *array*, a fixed-size list of elements of the same type. By default, arrays are immutable.

```
let a = [1, 2, 3];      // a: [i32; 3]
let mut m = [1, 2, 3]; // mut m: [i32; 3]
```

There's a shorthand for initializing each element of an array to the same value. In this example, each element of `a` will be initialized to `0`:

```
let a = [0; 20]; // a: [i32; 20]
```

Arrays have type `[T; N]`. We'll talk about this `T` notation later, when we cover generics.

You can get the number of elements in an array `a` with `a.len()`, and use `a.iter()` to iterate over them with a for loop. This code will print each number in order:

```
let a = [1, 2, 3];

println!("a has {} elements", a.len());
for e in a.iter() {
    println!("{}", e);
}
```

You can access a particular element of an array with *subscript notation*:

```
let names = ["Graydon", "Brian", "Niko"]; // names: [&str; 3]

println!("The second name is: {}", names[1]);
```

Subscripts start at zero, like in most programming languages, so the first name is `names[0]` and the second name is `names[1]`. The above example prints `The second name is: Brian`. If you try to use a subscript that is not in the array, you will get an error: array access is bounds-checked at run-time. Such errant access is the source of many bugs in other systems programming languages.

A *vector* is a dynamic or "growable" array, implemented as the standard library type `Vec<T>` (we'll talk about what the `<T>` means later). Vectors are to arrays what `String` is to `&str`. You can create them with the `vec!` macro:

```
let v = vec![1, 2, 3]; // v: Vec<i32>
```

(Notice that unlike the `println!` macro we've used in the past, we use square brackets `[]` with `vec!`. Rust allows you to use either in either situation, this is just convention.)

You can get the length of, iterate over, and subscript vectors just like arrays. In addition, (mutable) vectors can grow automatically:

```
let mut nums = vec![1, 2, 3]; // mut nums: Vec<i32>
```

```
nums.push(4);

println!("The length of nums is now {}", nums.len());   // Prints 4
```

Vectors have many more useful methods.

A *slice* is a reference to (or "view" into) an array. They are useful for allowing safe, efficient access to a portion of an array without copying. For example, you might want to reference just one line of a file read into memory. By nature, a slice is not created directly, but from an existing variable. Slices have a length, can be mutable or not, and in many ways behave like arrays:

```
let a = [0, 1, 2, 3, 4];
let middle = a.slice(1, 4);     // A slice of a: just the elements [1,2,3]

for e in middle.iter() {
    println!("{}", e);          // Prints 1, 2, 3
}
```

You can also take a slice of a vector, `String`, or `&str`, because they are backed by arrays. Slices have type `&[T]`, which we'll talk about when we cover generics.

We have now learned all of the most basic Rust concepts. We're ready to start building our guessing game, we just need to know one last thing: how to get input from the keyboard. You can't have a guessing game without the ability to guess!

1.13 Standard Input

Getting input from the keyboard is pretty easy, but uses some things we haven't seen before. Here's a simple program that reads some input, and then prints it back out:

```
fn main() {
    println!("Type something!");

    let input = std::io::stdin().read_line().ok().expect("Failed to read line");

    println!("{}", input);
}
```

Let's go over these chunks, one by one:

```
std::io::stdin();
```

This calls a function, `stdin()`, that lives inside the `std::io` module. As you can imagine, everything in `std` is provided by Rust, the 'standard library.' We'll talk more about the module system later.

Since writing the fully qualified name all the time is annoying, we can use the `use` statement to import it in:

```
use std::io::stdin;

stdin();
```

However, it's considered better practice to not import individual functions, but to import the module, and only use one level of qualification:

```
use std::io;

io::stdin();
```

Let's update our example to use this style:

```
use std::io;

fn main() {
```

```
    println!("Type something!");

    let input = io::stdin().read_line().ok().expect("Failed to read line");

    println!("{}", input);
}
```

Next up:

```
.read_line()
```

The `read_line()` method can be called on the result of `stdin()` to return a full line of input. Nice and easy.

```
.ok().expect("Failed to read line");
```

Do you remember this code?

```
enum OptionalInt {
    Value(i32),
    Missing,
}

fn main() {
    let x = OptionalInt::Value(5);
    let y = OptionalInt::Missing;

    match x {
        OptionalInt::Value(n) => println!("x is {}", n),
        OptionalInt::Missing  => println!("x is missing!"),
    }

    match y {
        OptionalInt::Value(n) => println!("y is {}", n),
        OptionalInt::Missing  => println!("y is missing!"),
    }
}
```

We had to match each time to see if we had a value or not. In this case, though, we *know* that `x` has a `Value`, but `match` forces us to handle the `missing` case. This is what we want 99% of the time, but sometimes, we know better than the compiler.

Likewise, `read_line()` does not return a line of input. It *might* return a line of input, though it might also fail to do so. This could happen if our program isn't running in a terminal, but as part of a cron job, or some other context where there's no standard input. Because of this, `read_line` returns a type very similar to our `OptionalInt`: an `IoResult<T>`. We haven't talked about `IoResult<T>` yet because it is the *generic* form of our `OptionalInt`. Until then, you can think of it as being the same thing, just for any type – not just `i32`s.

Rust provides a method on these `IoResult<T>`s called `ok()`, which does the same thing as our `match` statement but assumes that we have a valid value. We then call `expect()` on the result, which will terminate our program if we don't have a valid value. In this case, if we can't get input, our program doesn't work, so we're okay with that. In most cases, we would want to handle the error case explicitly. `expect()` allows us to give an error message if this crash happens.

We will cover the exact details of how all of this works later in the Guide. For now, this gives you enough of a basic understanding to work with.

Back to the code we were working on! Here's a refresher:

```
use std::io;

fn main() {
    println!("Type something!");

    let input = io::stdin().read_line().ok().expect("Failed to read line");
```

```
    println!("{}", input);
}
```

With long lines like this, Rust gives you some flexibility with the whitespace. We *could* write the example like this:

```
use std::io;

fn main() {
    println!("Type something!");

                                                // here, we'll show the types at each step

    let input = io::stdin()                     // std::io::stdio::StdinReader
                  .read_line()                  // IoResult<String>
                  .ok()                         // Option<String>
                  .expect("Failed to read line"); // String

    println!("{}", input);
}
```

Sometimes, this makes things more readable – sometimes, less. Use your judgement here.

That's all you need to get basic input from the standard input! It's not too complicated, but there are a number of small parts.

1.14 Guessing Game

Okay! We've got the basics of Rust down. Let's write a bigger program.

For our first project, we'll implement a classic beginner programming problem: the guessing game. Here's how it works: Our program will generate a random integer between one and a hundred. It will then prompt us to enter a guess. Upon entering our guess, it will tell us if we're too low or too high. Once we guess correctly, it will congratulate us. Sound good?

1.14.0.1 Set up

Let's set up a new project. Go to your projects directory. Remember how we had to create our directory structure and a `Cargo.toml` for `hello_world`? Cargo has a command that does that for us. Let's give it a shot:

```
$ cd ~/projects
$ cargo new guessing_game --bin
$ cd guessing_game
```

We pass the name of our project to `cargo new`, and then the `--bin` flag, since we're making a binary, rather than a library.

Check out the generated `Cargo.toml`:

```
[package]

name = "guessing_game"
version = "0.0.1"
authors = ["Your Name <you@example.com>"]
```

Cargo gets this information from your environment. If it's not correct, go ahead and fix that.

Finally, Cargo generated a "Hello, world!" for us. Check out `src/main.rs`:

```
fn main() {
    println!("Hello, world!")
}
```

Let's try compiling what Cargo gave us:

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
```

Excellent! Open up your `src/main.rs` again. We'll be writing all of our code in this file. We'll talk about multiple-file projects later on in the guide.

Before we move on, let me show you one more Cargo command: `run`. `cargo run` is kind of like `cargo build`, but it also then runs the produced executable. Try it out:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Hello, world!
```

Great! The `run` command comes in handy when you need to rapidly iterate on a project. Our game is just such a project, we need to quickly test each iteration before moving on to the next one.

1.14.0.2 Processing a Guess

Let's get to it! The first thing we need to do for our guessing game is allow our player to input a guess. Put this in your `src/main.rs`:

```
use std::io;

fn main() {
    println!("Guess the number!");

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");

    println!("You guessed: {}", input);
}
```

You've seen this code before, when we talked about standard input. We import the `std::io` module with `use`, and then our `main` function contains our program's logic. We print a little message announcing the game, ask the user to input a guess, get their input, and then print it out.

Because we talked about this in the section on standard I/O, I won't go into more details here. If you need a refresher, go re-read that section.

1.14.0.3 Generating a secret number

Next, we need to generate a secret number. To do that, we need to use Rust's random number generation, which we haven't talked about yet. Rust includes a bunch of interesting functions in its standard library. If you need a bit of code, it's possible that it's already been written for you! In this case, we do know that Rust has random number generation, but we don't know how to use it.

Enter the docs. Rust has a page specifically to document the standard library. You can find that page here. There's a lot of information on that page, but the best part is the search bar. Right up at the top, there's a box that you can enter in a search term. The search is pretty primitive right now, but is getting better all the time. If you type "random" in that box, the page will update to this one. The very first result is a link to `std::rand::random`. If we click on that result, we'll be taken to its documentation page.

This page shows us a few things: the type signature of the function, some explanatory text, and then an example. Let's try to modify our code to add in the `random` function and see what happens:

```
use std::io;
use std::rand;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random() % 100) + 1; // secret_number: i32

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");


    println!("You guessed: {}", input);
}
```

The first thing we changed was to `use std::rand`, as the docs explained. We then added in a `let` expression to create a variable binding named `secret_number`, and we printed out its result.

Also, you may wonder why we are using `%` on the result of `rand::random()`. This operator is called *modulo*, and it returns the remainder of a division. By taking the modulo of the result of `rand::random()`, we're limiting the values to be between 0 and 99. Then, we add one to the result, making it from 1 to 100. Using modulo can give you a very, very small bias in the result, but for this example, it is not important.

Let's try to compile this using `cargo build`:

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:7:26: 7:34 error: the type of this value must be known in this context
src/main.rs:7     let secret_number = (rand::random() % 100) + 1;
                                       ^~~~~~~~
error: aborting due to previous error
```

It didn't work! Rust says "the type of this value must be known in this context." What's up with that? Well, as it turns out, `rand::random()` can generate many kinds of random values, not just integers. And in this case, Rust isn't sure what kind of value `random()` should generate. So we have to help it. With number literals, we can just add an `i32` onto the end to tell Rust they're integers, but that does not work with functions. There's a different syntax, and it looks like this:

```
rand::random::<i32>();
```

This says "please give me a random `i32` value." We can change our code to use this hint:

```
use std::io;
use std::rand;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<i32>() % 100) + 1;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");


    println!("You guessed: {}", input);
}
```

Try running our new program a few times:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 7
Please input your guess.
4
You guessed: 4
$ ./target/guessing_game
Guess the number!
The secret number is: 83
Please input your guess.
5
You guessed: 5
$ ./target/guessing_game
Guess the number!
The secret number is: -29
Please input your guess.
42
You guessed: 42
```

Wait. Negative 29? We wanted a number between one and a hundred! We have two options here: we can either ask `random()` to generate an unsigned integer, which can only be positive, or we can use the `abs()` function. Let's go with the unsigned integer approach. If we want a random positive number, we should ask for a random positive number. Our code looks like this now:

```
use std::io;
use std::rand;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");


    println!("You guessed: {}", input);
}
```

And trying it out:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 57
Please input your guess.
3
You guessed: 3
```

Great! Next up: let's compare our guess to the secret guess.

1.14.0.4 Comparing guesses

If you remember, earlier in the guide, we made a `cmp` function that compared two numbers. Let's add that in, along with a `match` statement to compare our guess to the secret number:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");


    println!("You guessed: {}", input);

    match cmp(input, secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}

fn cmp(a: i32, b: i32) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

If we try to compile, we'll get some errors:

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:20:15: 20:20 error: mismatched types: expected `i32` but found `collections::string::
src/main.rs:20     match cmp(input, secret_number) {
                         ^~~~~
src/main.rs:20:22: 20:35 error: mismatched types: expected `i32` but found `uint` (expected i32 b
src/main.rs:20     match cmp(input, secret_number) {
                                    ^~~~~~~~~~~~~
error: aborting due to 2 previous errors
```

This often happens when writing Rust programs, and is one of Rust's greatest strengths. You try out some code, see if it compiles, and Rust tells you that you've done something wrong. In this case, our `cmp` function works on integers, but we've given it unsigned integers. In this case, the fix is easy, because we wrote the `cmp` function! Let's change it to take `uint`s:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");


    println!("You guessed: {}", input);
```

```
    match cmp(input, secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

And try compiling again:

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:20:15: 20:20 error: mismatched types: expected `uint` but found `collections::string:
src/main.rs:20     match cmp(input, secret_number) {
                         ^~~~~
error: aborting due to previous error
```

This error is similar to the last one: we expected to get a `uint`, but we got a `String` instead! That's because our `input` variable is coming from the standard input, and you can guess anything. Try it:

```
$ ./target/guessing_game
Guess the number!
The secret number is: 73
Please input your guess.
hello
You guessed: hello
```

Oops! Also, you'll note that we just ran our program even though it didn't compile. This works because the older version we did successfully compile was still lying around. Gotta be careful!

Anyway, we have a `String`, but we need a `uint`. What to do? Well, there's a function for that:

```
let input = io::stdin().read_line()
                       .ok()
                       .expect("Failed to read line");
let input_num: Option<uint> = input.parse();
```

The `parse` function takes in a `&str` value and converts it into something. We tell it what kind of something with a type hint. Remember our type hint with `random()`? It looked like this:

```
rand::random::<uint>();
```

There's an alternate way of providing a hint too, and that's declaring the type in a `let`:

```
let x: uint = rand::random();
```

In this case, we say `x` is a `uint` explicitly, so Rust is able to properly tell `random()` what to generate. In a similar fashion, both of these work:

```
let input_num = "5".parse::<uint>();         // input_num: Option<uint>
let input_num: Option<uint> = "5".parse();   // input_num: Option<uint>
```

Anyway, with us now converting our input to a number, our code looks like this:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;
```

```
    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");
    let input_num: Option<uint> = input.parse();

    println!("You guessed: {}", input_num);

    match cmp(input_num, secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

Let's try it out!

```
$ cargo build
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
src/main.rs:22:15: 22:24 error: mismatched types: expected `uint` but found `core::option::Option
src/main.rs:22     match cmp(input_num, secret_number) {
                             ^~~~~~~~~
error: aborting due to previous error
```

Oh yeah! Our `input_num` has the type `Option<uint>`, rather than `uint`. We need to unwrap the Option. If you remember from before, `match` is a great way to do that. Try this code:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");
    let input_num: Option<uint> = input.parse();

    let num = match input_num {
        Some(num) => num,
        None      => {
            println!("Please input a number!");
            return;
        }
    };


    println!("You guessed: {}", num);

    match cmp(num, secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
```

```
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

We use a `match` to either give us the `uint` inside of the `Option`, or else print an error message and return. Let's give this a shot:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 17
Please input your guess.
5
Please input a number!
```

Uh, what? But we did!

… actually, we didn't. See, when you get a line of input from `stdin()`, you get all the input. Including the `\n` character from you pressing Enter. Therefore, `parse()` sees the string `"5\n"` and says "nope, that's not a number; there's non-number stuff in there!" Luckily for us, `&str`s have an easy method we can use defined on them: `trim()`. One small modification, and our code looks like this:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    println!("Please input your guess.");

    let input = io::stdin().read_line()
                           .ok()
                           .expect("Failed to read line");
    let input_num: Option<uint> = input.trim().parse();

    let num = match input_num {
        Some(num) => num,
        None      => {
            println!("Please input a number!");
            return;
        }
    };


    println!("You guessed: {}", num);

    match cmp(num, secret_number) {
        Ordering::Less    => println!("Too small!"),
        Ordering::Greater => println!("Too big!"),
        Ordering::Equal   => println!("You win!"),
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

Let's try it!

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 58
Please input your guess.
  76
You guessed: 76
Too big!
```

Nice! You can see I even added spaces before my guess, and it still figured out that I guessed 76. Run the program a few times, and verify that guessing the number works, as well as guessing a number too small.

The Rust compiler helped us out quite a bit there! This technique is called "leaning on the compiler", and it's often useful when working on some code. Let the error messages help guide you towards the correct types.

Now we've got most of the game working, but we can only make one guess. Let's change that by adding loops!

1.14.0.5 Looping

As we already discussed, the `loop` keyword gives us an infinite loop. Let's add that in:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    loop {

        println!("Please input your guess.");

        let input = io::stdin().read_line()
                               .ok()
                               .expect("Failed to read line");
        let input_num: Option<uint> = input.trim().parse();

        let num = match input_num {
            Some(num) => num,
            None      => {
                println!("Please input a number!");
                return;
            }
        };


        println!("You guessed: {}", num);

        match cmp(num, secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => println!("You win!"),
        }
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

And try it out. But wait, didn't we just add an infinite loop? Yup. Remember that `return`? If we give a non-number answer, we'll `return` and quit. Observe:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 59
Please input your guess.
45
You guessed: 45
Too small!
Please input your guess.
60
You guessed: 60
Too big!
Please input your guess.
59
You guessed: 59
You win!
Please input your guess.
quit
Please input a number!
```

Ha! `quit` actually quits. As does any other non-number input. Well, this is suboptimal to say the least. First, let's actually quit when you win the game:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    loop {

        println!("Please input your guess.");

        let input = io::stdin().read_line()
                               .ok()
                               .expect("Failed to read line");
        let input_num: Option<uint> = input.trim().parse();

        let num = match input_num {
            Some(num) => num,
            None      => {
                println!("Please input a number!");
                return;
            }
        };


        println!("You guessed: {}", num);

        match cmp(num, secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                return;
            },
        }
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
```

```
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

By adding the `return` line after the `You win!`, we'll exit the program when we win. We have just one more tweak to make: when someone inputs a non-number, we don't want to quit, we just want to ignore it. Change that `return` to `continue`:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    println!("The secret number is: {}", secret_number);

    loop {

        println!("Please input your guess.");

        let input = io::stdin().read_line()
                               .ok()
                               .expect("Failed to read line");
        let input_num: Option<uint> = input.trim().parse();

        let num = match input_num {
            Some(num) => num,
            None      => {
                println!("Please input a number!");
                continue;
            }
        };


        println!("You guessed: {}", num);

        match cmp(num, secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                return;
            },
        }
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

Now we should be good! Let's try:

```
$ cargo run
   Compiling guessing_game v0.0.1 (file:///home/you/projects/guessing_game)
     Running `target/guessing_game`
Guess the number!
The secret number is: 61
Please input your guess.
10
You guessed: 10
Too small!
Please input your guess.
99
```

```
You guessed: 99
Too big!
Please input your guess.
foo
Please input a number!
Please input your guess.
61
You guessed: 61
You win!
```

Awesome! With one tiny last tweak, we have finished the guessing game. Can you think of what it is? That's right, we don't want to print out the secret number. It was good for testing, but it kind of ruins the game. Here's our final source:

```
use std::io;
use std::rand;
use std::cmp::Ordering;

fn main() {
    println!("Guess the number!");

    let secret_number = (rand::random::<uint>() % 100u) + 1u;

    loop {

        println!("Please input your guess.");

        let input = io::stdin().read_line()
                               .ok()
                               .expect("Failed to read line");
        let input_num: Option<uint> = input.trim().parse();

        let num = match input_num {
            Some(num) => num,
            None      => {
                println!("Please input a number!");
                continue;
            }
        };


        println!("You guessed: {}", num);

        match cmp(num, secret_number) {
            Ordering::Less    => println!("Too small!"),
            Ordering::Greater => println!("Too big!"),
            Ordering::Equal   => {
                println!("You win!");
                return;
            },
        }
    }
}

fn cmp(a: uint, b: uint) -> Ordering {
    if a < b { Ordering::Less }
    else if a > b { Ordering::Greater }
    else { Ordering::Equal }
}
```

1.14.0.6 Complete!

At this point, you have successfully built the Guessing Game! Congratulations!

You've now learned the basic syntax of Rust. All of this is relatively close to various other programming languages you have used in the past. These fundamental syntactical and semantic elements will form the foundation for the rest of your Rust education.

Now that you’re an expert at the basics, it’s time to learn about some of Rust’s more unique features.

2 Intermediate Rust

This section contains individual chapters, which are self-contained. They focus on specific topics, and can be read in any order.

After reading "Intermediate," you will have a solid understanding of Rust, and will be able to understand most Rust code and write more complex programs.

2.1 Crates and Modules

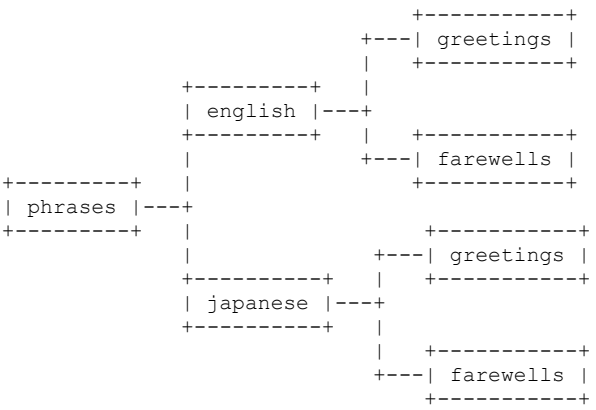
When a project starts getting large, it's considered a good software engineering practice to split it up into a bunch of smaller pieces, and then fit them together. It's also important to have a well-defined interface, so that some of your functionality is private, and some is public. To facilitate these kinds of things, Rust has a module system.

2.1.1 Basic terminology: Crates and Modules

Rust has two distinct terms that relate to the module system: *crate* and *module*. A crate is synonymous with a *library* or *package* in other languages. Hence "Cargo" as the name of Rust's package management tool: you ship your crates to others with Cargo. Crates can produce an executable or a shared library, depending on the project.

Each crate has an implicit *root module* that contains the code for that crate. You can then define a tree of sub-modules under that root module. Modules allow you to partition your code within the crate itself.

As an example, let's make a *phrases* crate, which will give us various phrases in different languages. To keep things simple, we'll stick to "greetings" and "farewells" as two kinds of phrases, and use English and Japanese (日本語） as two languages for those phrases to be in. We'll use this module layout:



In this example, `phrases` is the name of our crate. All of the rest are modules. You can see that they form a tree, branching out from the crate *root*, which is the root of the tree: `phrases` itself.

Now that we have a plan, let's define these modules in code. To start, generate a new crate with Cargo:

```
$ cargo new phrases
$ cd phrases
```

If you remember, this generates a simple project for us:

```
$ tree .
.
├── Cargo.toml
└── src
```

```
    └── lib.rs

1 directory, 2 files
```

`src/lib.rs` is our crate root, corresponding to the `phrases` in our diagram above.

2.1.2 Defining Modules

To define each of our modules, we use the `mod` keyword. Let's make our `src/lib.rs` look like this:

```
// in src/lib.rs

mod english {
    mod greetings {

    }

    mod farewells {

    }
}

mod japanese {
    mod greetings {

    }

    mod farewells {

    }
}
```

After the `mod` keyword, you give the name of the module. Module names follow the conventions for other Rust identifiers: `lower_snake_case`. The contents of each module are within curly braces (`{}`).

Within a given `mod`, you can declare sub-`mod`s. We can refer to sub-modules with double-colon (`::`) notation: our four nested modules are `english::greetings`, `english::farewells`, `japanese::greetings`, and `japanese::farewells`. Because these sub-modules are namespaced under their parent module, the names don't conflict: `english::greetings` and `japanese::greetings` are distinct, even though their names are both `greetings`.

Because this crate does not have a `main()` function, and is called `lib.rs`, Cargo will build this crate as a library:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
$ ls target
deps  libphrases-a7448e02a0468eaa.rlib  native
```

`libphrase-hash.rlib` is the compiled crate. Before we see how to use this crate from another crate, let's break it up into multiple files.

2.1.3 Multiple file crates

If each crate were just one file, these files would get very large. It's often easier to split up crates into multiple files, and Rust supports this in two ways.

Instead of declaring a module like this:

```
mod english {
    // contents of our module go here
}
```

We can instead declare our module like this:

```
mod english;
```

If we do that, Rust will expect to find either a `english.rs` file, or a `english/mod.rs` file with the contents of our module:

```
// contents of our module go here
```

Note that in these files, you don't need to re-declare the module: that's already been done with the initial `mod` declaration.

Using these two techniques, we can break up our crate into two directories and seven files:

```
$ tree .
.
├── Cargo.lock
├── Cargo.toml
├── src
│   ├── english
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   ├── japanese
│   │   ├── farewells.rs
│   │   ├── greetings.rs
│   │   └── mod.rs
│   └── lib.rs
└── target
    ├── deps
    ├── libphrases-a7448e02a0468eaa.rlib
    └── native
```

`src/lib.rs` is our crate root, and looks like this:

```
// in src/lib.rs

mod english;

mod japanese;
```

These two declarations tell Rust to look for either `src/english.rs` and `src/japanese.rs`, or `src/english/mod.rs` and `src/japanese/mod.rs`, depending on our preference. In this case, because our modules have sub-modules, we've chosen the second. Both `src/english/mod.rs` and `src/japanese/mod.rs` look like this:

```
// both src/english/mod.rs and src/japanese/mod.rs

mod greetings;

mod farewells;
```

Again, these declarations tell Rust to look for either `src/english/greetings.rs` and `src/japanese/greetings.rs` or `src/english/farewells/mod.rs` and `src/japanese/farewells/mod.rs`. Because these sub-modules don't have their own sub-modules, we've chosen to make them `src/english/greetings.rs` and `src/japanese/farewells.rs`. Whew!

Right now, the contents of `src/english/greetings.rs` and `src/japanese/farewells.rs` are both empty at the moment. Let's add some functions.

Put this in `src/english/greetings.rs`:

```
// in src/english/greetings.rs

fn hello() -> String {
    "Hello!".to_string()
}
```

Put this in `src/english/farewells.rs`:

```
// in src/english/farewells.rs

fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Put this in `src/japanese/greetings.rs`:

```
// in src/japanese/greetings.rs

fn hello() -> String {
    "こんにちは".to_string()
}
```

Of course, you can copy and paste this from this web page, or just type something else. It's not important that you actually put "konnichiwa" to learn about the module system.

Put this in `src/japanese/farewells.rs`:

```
// in src/japanese/farewells.rs

fn goodbye() -> String {
    "さようなら".to_string()
}
```

(This is "Sayoonara", if you're curious.)

Now that we have our some functionality in our crate, let's try to use it from another crate.

2.1.4 Importing External Crates

We have a library crate. Let's make an executable crate that imports and uses our library.

Make a `src/main.rs` and put this in it: (it won't quite compile yet)

```
// in src/main.rs

extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());

    println!("Hello in Japanese: {}", phrases::japanese::greetings::hello());
    println!("Goodbye in Japanese: {}", phrases::japanese::farewells::goodbye());
}
```

The `extern crate` declaration tells Rust that we need to compile and link to the `phrases` crate. We can then use `phrases`' modules in this one. As we mentioned earlier, you can use double colons to refer to sub-modules and the functions inside of them.

Also, Cargo assumes that `src/main.rs` is the crate root of a binary crate, rather than a library crate. Once we compile `src/main.rs`, we'll get an executable that we can run. Our package now has two crates: `src/lib.rs` and `src/main.rs`. This pattern is quite common for executable crates: most functionality is in a library crate, and the executable crate uses that library. This way, other programs can also use the library crate, and it's also a nice separation of concerns.

This doesn't quite work yet, though. We get four errors that look similar to this:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
```

```
/home/you/projects/phrases/src/main.rs:4:38: 4:72 error: function `hello` is private
/home/you/projects/phrases/src/main.rs:4     println!("Hello in English: {}", phrases::english::g
                                                                                ^~~~~~~~~~~~~~~~~~~~
note: in expansion of format_args!
<std macros>:2:23: 2:77 note: expansion site
<std macros>:1:1: 3:2 note: in expansion of println!
/home/you/projects/phrases/src/main.rs:4:5: 4:76 note: expansion site
```

By default, everything is private in Rust. Let's talk about this in some more depth.

2.1.5 Exporting a Public Interface

Rust allows you to precisely control which aspects of your interface are public, and so private is the default. To make things public, you use the `pub` keyword. Let's focus on the `english` module first, so let's reduce our `src/main.rs` to just this:

```
// in src/main.rs

extern crate phrases;

fn main() {
    println!("Hello in English: {}", phrases::english::greetings::hello());
    println!("Goodbye in English: {}", phrases::english::farewells::goodbye());
}
```

In our `src/lib.rs`, let's add `pub` to the `english` module declaration:

```
// in src/lib.rs

pub mod english;

mod japanese;
```

And in our `src/english/mod.rs`, let's make both `pub`:

```
// in src/english/mod.rs

pub mod greetings;

pub mod farewells;
```

In our `src/english/greetings.rs`, let's add `pub` to our `fn` declaration:

```
// in src/english/greetings.rs

pub fn hello() -> String {
    "Hello!".to_string()
}
```

And also in `src/english/farewells.rs`:

```
// in src/english/farewells.rs

pub fn goodbye() -> String {
    "Goodbye.".to_string()
}
```

Now, our crate compiles, albeit with warnings about not using the `japanese` functions:

```
$ cargo run
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
/home/you/projects/phrases/src/japanese/greetings.rs:1:1: 3:2 warning: code is never used: `hello
/home/you/projects/phrases/src/japanese/greetings.rs:1 fn hello() -> String {
/home/you/projects/phrases/src/japanese/greetings.rs:2     "こんにちは".to_string()
/home/you/projects/phrases/src/japanese/greetings.rs:3 }
/home/you/projects/phrases/src/japanese/farewells.rs:1:1: 3:2 warning: code is never used: `goodb
```

```
/home/you/projects/phrases/src/japanese/farewells.rs:1 fn goodbye() -> String {
/home/you/projects/phrases/src/japanese/farewells.rs:2     "さようなら".to_string()
/home/you/projects/phrases/src/japanese/farewells.rs:3 }
     Running `target/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
```

Now that our functions are public, we can use them. Great! However, typing out `phrases::english::greetings::hello()` is very long and repetitive. Rust has another keyword for importing names into the current scope, so that you can refer to them with shorter names. Let's talk about `use`.

2.1.6 Importing Modules with `use`

Rust has a `use` keyword, which allows us to import names into our local scope. Let's change our `src/main.rs` to look like this:

```
// in src/main.rs

extern crate phrases;

use phrases::english::greetings;
use phrases::english::farewells;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());
}
```

The two `use` lines import each module into the local scope, so we can refer to the functions by a much shorter name. By convention, when importing functions, it's considered best practice to import the module, rather than the function directly. In other words, you *can* do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::english::farewells::goodbye;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Goodbye in English: {}", goodbye());
}
```

But it is not idiomatic. This is significantly more likely to introducing a naming conflict. In our short program, it's not a big deal, but as it grows, it becomes a problem. If we have conflicting names, Rust will give a compilation error. For example, if we made the `japanese` functions public, and tried to do this:

```
extern crate phrases;

use phrases::english::greetings::hello;
use phrases::japanese::greetings::hello;

fn main() {
    println!("Hello in English: {}", hello());
    println!("Hello in Japanese: {}", hello());
}
```

Rust will give us a compile-time error:

```
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
/home/you/projects/phrases/src/main.rs:4:5: 4:40 error: a value named `hello` has already been im
/home/you/projects/phrases/src/main.rs:4 use phrases::japanese::greetings::hello;
                                             ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
error: aborting due to previous error
Could not compile `phrases`.
```

If we're importing multiple names from the same module, we don't have to type it out twice. Rust has a shortcut syntax for writing this:

```
use phrases::english::greetings;
use phrases::english::farewells;
```

You use curly braces:

```
use phrases::english::{greetings, farewells};
```

These two declarations are equivalent, but the second is a lot less typing.

2.1.6.1 Re-exporting with `pub use`

You don't just use `use` to shorten identifiers. You can also use it inside of your crate to re-export a function inside another module. This allows you to present an external interface that may not directly map to your internal code organization.

Let's look at an example. Modify your `src/main.rs` to read like this:

```
// in src/main.rs

extern crate phrases;

use phrases::english::{greetings,farewells};
use phrases::japanese;

fn main() {
    println!("Hello in English: {}", greetings::hello());
    println!("Goodbye in English: {}", farewells::goodbye());

    println!("Hello in Japanese: {}", japanese::hello());
    println!("Goodbye in Japanese: {}", japanese::goodbye());
}
```

Then, modify your `src/lib.rs` to make the `japanese` mod public:

```
// in src/lib.rs

pub mod english;

pub mod japanese;
```

Next, make the two functions public, first in `src/japanese/greetings.rs`:

```
// in src/japanese/greetings.rs

pub fn hello() -> String {
    "こんにちは".to_string()
}
```

And then in `src/japanese/farewells.rs`:

```
// in src/japanese/farewells.rs

pub fn goodbye() -> String {
    "さようなら".to_string()
}
```

Finally, modify your `src/japanese/mod.rs` to read like this:

```
// in src/japanese/mod.rs

pub use self::greetings::hello;
pub use self::farewells::goodbye;
```

```
mod greetings;

mod farewells;
```

The `pub use` declaration brings the function into scope at this part of our module hierarchy. Because we've `pub use`d this inside of our `japanese` module, we now have a `phrases::japanese::hello()` function and a `phrases::japanese::goodbye()` function, even though the code for them lives in `phrases::japanese::greetings::hello()` and `phrases::japanese::farewells::goodbye()`. Our internal organization doesn't define our external interface.

Also, note that we `pub use`d before we declared our `mod`s. Rust requires that `use` declarations go first.

This will build and run:

```
$ cargo build
   Compiling phrases v0.0.1 (file:///home/you/projects/phrases)
     Running `target/phrases`
Hello in English: Hello!
Goodbye in English: Goodbye.
Hello in Japanese: こんにちは
Goodbye in Japanese: さようなら
```

2.2 Testing

> Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence.
>
> Edsger W. Dijkstra, "The Humble Programmer" (1972)

Let's talk about how to test Rust code. What we will not be talking about is the right way to test Rust code. There are many schools of thought regarding the right and wrong way to write tests. All of these approaches use the same basic tools, and so we'll show you the syntax for using them.

2.2.1 The `test` attribute

At its simplest, a test in Rust is a function that's annotated with the `test` attribute. Let's make a new project with Cargo called `adder`:

```
$ cargo new adder
$ cd adder
```

Cargo will automatically generate a simple test when you make a new project. Here's the contents of `src/lib.rs`:

```
#[test]
fn it_works() {
}
```

Note the `#[test]`. This attribute indicates that this is a test function. It currently has no body. That's good enough to pass! We can run the tests with `cargo test`:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder
```

```
running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Cargo compiled and ran our tests. There are two sets of output here: one for the test we wrote, and another for documentation tests. We'll talk about those later. For now, see this line:

```
test it_works ... ok
```

Note the `it_works`. This comes from the name of our function:

```
fn it_works() {
# }
```

We also get a summary line:

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured
```

So why does our do-nothing test pass? Any test which doesn't `panic!` passes, and any test that does `panic!` fails. Let's make our test fail:

```
#[test]
fn it_works() {
    assert!(false);
}
```

`assert!` is a macro provided by Rust which takes one argument: if the argument is `true`, nothing happens. If the argument is false, it `panic!`s. Let's run our tests again:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... FAILED

failures:

---- it_works stdout ----
        task 'it_works' panicked at 'assertion failed: false', /home/steve/tmp/adder/src/lib.rs:3



failures:
    it_works

test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured

task '<main>' panicked at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:247
```

Rust indicates that our test failed:

```
test it_works ... FAILED
```

And that's reflected in the summary line:

```
test result: FAILED. 0 passed; 1 failed; 0 ignored; 0 measured
```

We also get a non-zero status code:

```
$ echo $?
101
```

This is useful if you want to integrate `cargo test` into other tooling.

We can invert our test's failure with another attribute: `should_fail`:

```
#[test]
#[should_fail]
fn it_works() {
    assert!(false);
}
```

This test will now succeed if we `panic!` and fail if we complete. Let's try it:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Rust provides another macro, `assert_eq!`, that compares two arguments for equality:

```
#[test]
#[should_fail]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

Does this test pass or fail? Because of the `should_fail` attribute, it passes:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

`should_fail` tests can be fragile, as it's hard to guarantee that the test didn't fail for an unexpected reason. To help with this, an optional `expected` parameter can be added to the `should_fail` attribute. The test harness will make sure that the failure message contains the provided text. A safer version of the example above would be:

```
#[test]
#[should_fail(expected = "assertion failed")]
fn it_works() {
    assert_eq!("Hello", "world");
}
```

That's all there is to the basics! Let's write one 'real' test:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[test]
fn it_works() {
    assert_eq!(4, add_two(2));
}
```

This is a very common use of `assert_eq!`: call some function with some known arguments and compare it to the expected output.

2.2.2 The `test` module

There is one way in which our existing example is not idiomatic: it's missing the test module. The idiomatic way of writing our example looks like this:

```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::add_two;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

There's a few changes here. The first is the introduction of a `mod tests` with a `cfg` attribute. The module allows us to group all of our tests together, and to also define helper functions if needed, that don't become a part of the rest of our crate. The `cfg` attribute only compiles our test code if we're currently trying to run the tests. This can save compile time, and also ensures that our tests are entirely left out of a normal build.

The second change is the `use` declaration. Because we're in an inner module, we need to bring our test function into scope. This can be annoying if you have a large module, and so this is a common use of the `glob` feature. Let's change our `src/lib.rs` to make use of it:

```
#![feature(globs)]

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Note the `feature` attribute, as well as the different `use` line. Now we run our tests:

```
$ cargo test
    Updating registry `https://github.com/rust-lang/crates.io-index`
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test test::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

It works!

The current convention is to use the `test` module to hold your "unit-style" tests. Anything that just tests one small bit of functionality makes sense to go here. But what about "integration-style" tests instead? For that, we have the `tests` directory

2.2.3 The `tests` directory

To write an integration test, let's make a `tests` directory, and put a `tests/lib.rs` file inside, with this as its contents:

```
extern crate adder;

#[test]
fn it_works() {
    assert_eq!(4, adder::add_two(2));
}
```

This looks similar to our previous tests, but slightly different. We now have an `extern crate adder` at the top. This is because the tests in the `tests` directory are an entirely separate crate, and so we need to import our library. This is also why `tests` is a suitable place to write integration-style tests: they use the library like any other consumer of it would.

Let's run them:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/you/projects/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test test::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 0 tests

test result: ok. 0 passed; 0 failed; 0 ignored; 0 measured
```

Now we have three sections: our previous test is also run, as well as our new one.

That's all there is to the `tests` directory. The `test` module isn't needed here, since the whole thing is focused on tests.

Let's finally check out that third section: documentation tests.

2.2.4 Documentation tests

Nothing is better than documentation with examples. Nothing is worse than examples that don't actually work, because the code has changed since the documentation has been written. To this end, Rust supports automatically running examples in your documentation. Here's a fleshed-out `src/lib.rs` with examples:

```
//! The `adder` crate provides functions that add numbers to other numbers.
//!
//! # Examples
//!
//! ```
//! assert_eq!(4, adder::add_two(2));
```

```
//! ```

#![feature(globs)]

/// This function adds two to its argument.
///
/// # Examples
///
/// ```
/// use adder::add_two;
///
/// assert_eq!(4, add_two(2));
/// ```
pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }
}
```

Note the module-level documentation with `//!` and the function-level documentation with `///`. Rust's documentation supports Markdown in comments, and so triple graves mark code blocks. It is conventional to include the `# Examples` section, exactly like that, with examples following.

Let's run the tests again:

```
$ cargo test
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/adder-91b3e234d4ed382a

running 1 test
test test::it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

     Running target/lib-c18e7d3494509e74

running 1 test
test it_works ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured

   Doc-tests adder

running 2 tests
test add_two_0 ... ok
test _0 ... ok

test result: ok. 2 passed; 0 failed; 0 ignored; 0 measured
```

Now we have all three kinds of tests running! Note the names of the documentation tests: the `_0` is generated for the module test, and `add_two_0` for the function test. These will auto increment with names like `add_two_1` as you add more examples.

2.2.5 Benchmark tests

Rust also supports benchmark tests, which can test the performance of your code. Let's make our `src/lib.rs` look like this (comments elided):

```
#![feature(globs)]

extern crate test;

pub fn add_two(a: i32) -> i32 {
    a + 2
}

#[cfg(test)]
mod tests {
    use super::*;
    use test::Bencher;

    #[test]
    fn it_works() {
        assert_eq!(4, add_two(2));
    }

    #[bench]
    fn bench_add_two(b: &mut Bencher) {
        b.iter(|| add_two(2));
    }
}
```

We've imported the `test` crate, which contains our benchmarking support. We have a new function as well, with the `bench` attribute. Unlike regular tests, which take no arguments, benchmark tests take a `&mut Bencher`. This `Bencher` provides an `iter` method, which takes a closure. This closure contains the code we'd like to benchmark.

We can run benchmark tests with `cargo bench`:

```
$ cargo bench
   Compiling adder v0.0.1 (file:///home/steve/tmp/adder)
     Running target/release/adder-91b3e234d4ed382a

running 2 tests
test tests::it_works ... ignored
test tests::bench_add_two ... bench:         1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 1 ignored; 1 measured
```

Our non-benchmark test was ignored. You may have noticed that `cargo bench` takes a bit longer than `cargo test`. This is because Rust runs our benchmark a number of times, and then takes the average. Because we're doing so little work in this example, we have a `1 ns/iter (+/- 0)`, but this would show the variance if there was one.

Advice on writing benchmarks:

- Move setup code outside the `iter` loop; only put the part you want to measure inside
- Make the code do "the same thing" on each iteration; do not accumulate or change state
- Make the outer function idempotent too; the benchmark runner is likely to run it many times
- Make the inner `iter` loop short and fast so benchmark runs are fast and the calibrator can adjust the run-length at fine resolution
- Make the code in the `iter` loop do something simple, to assist in pinpointing performance improvements (or regressions)

2.2.5.1 Gotcha: optimizations

There's another tricky part to writing benchmarks: benchmarks compiled with optimizations activated can be dramatically changed by the optimizer so that the benchmark is no longer benchmarking what one expects. For example, the compiler might recognize that some calculation has no external effects and remove it entirely.

```
extern crate test;
use test::Bencher;

#[bench]
```

```
fn bench_xor_1000_ints(b: &mut Bencher) {
    b.iter(|| {
        range(0u, 1000).fold(0, |old, new| old ^ new);
    });
}
```

gives the following results

```
running 1 test
test bench_xor_1000_ints ... bench:         0 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

The benchmarking runner offers two ways to avoid this. Either, the closure that the `iter` method receives can return an arbitrary value which forces the optimizer to consider the result used and ensures it cannot remove the computation entirely. This could be done for the example above by adjusting the `b.iter` call to

```
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} } let b = X;
b.iter(|| {
    // note lack of `;` (could also use an explicit `return`).
    range(0u, 1000).fold(0, |old, new| old ^ new)
});
```

Or, the other option is to call the generic `test::black_box` function, which is an opaque "black box" to the optimizer and so forces it to consider any argument as used.

```
extern crate test;

# fn main() {
# struct X;
# impl X { fn iter<T, F>(&self, _: F) where F: FnMut() -> T {} } let b = X;
b.iter(|| {
    let mut n = 1000_u32;

    test::black_box(&mut n); // pretend to modify `n`

    range(0, n).fold(0, |a, b| a ^ b)
})
# }
```

Neither of these read or modify the value, and are very cheap for small values. Larger values can be passed indirectly to reduce overhead (e.g. `black_box(&huge_struct)`).

Performing either of the above changes gives the following benchmarking results

```
running 1 test
test bench_xor_1000_ints ... bench:       1 ns/iter (+/- 0)

test result: ok. 0 passed; 0 failed; 0 ignored; 1 measured
```

However, the optimizer can still modify a testcase in an undesirable manner even when using either of the above.

2.3 Pointers

Rust's pointers are one of its more unique and compelling features. Pointers are also one of the more confusing topics for newcomers to Rust. They can also be confusing for people coming from other languages that support pointers, such as C++. This guide will help you understand this important topic.

Be sceptical of non-reference pointers in Rust: use them for a deliberate purpose, not just to make the compiler happy. Each pointer type comes with an explanation about when they are appropriate to use. Default to references unless you're in one of those specific situations.

You may be interested in the cheat sheet, which gives a quick overview of the types, names, and purpose of the various pointers.

2.3.1 An introduction

If you aren't familiar with the concept of pointers, here's a short introduction. Pointers are a very fundamental concept in systems programming languages, so it's important to understand them.

2.3.1.1 Pointer Basics

When you create a new variable binding, you're giving a name to a value that's stored at a particular location on the stack. (If you're not familiar with the *heap* vs. *stack*, please check out this Stack Overflow question, as the rest of this guide assumes you know the difference.) Like this:

```
let x = 5i;
let y = 8i;
```

location	value
0xd3e030	5
0xd3e028	8

We're making up memory locations here, they're just sample values. Anyway, the point is that `x`, the name we're using for our variable, corresponds to the memory location `0xd3e030`, and the value at that location is `5`. When we refer to `x`, we get the corresponding value. Hence, `x` is `5`.

Let's introduce a pointer. In some languages, there is just one type of 'pointer,' but in Rust, we have many types. In this case, we'll use a Rust *reference*, which is the simplest kind of pointer.

```
let x = 5i;
let y = 8i;
let z = &y;
```

location	value
0xd3e030	5
0xd3e028	8
0xd3e020	0xd3e028

See the difference? Rather than contain a value, the value of a pointer is a location in memory. In this case, the location of `y`. `x` and `y` have the type `int`, but `z` has the type `&int`. We can print this location using the `{:p}` format string:

```
let x = 5i;
let y = 8i;
let z = &y;

println!("{:p}", z);
```

This would print `0xd3e028`, with our fictional memory addresses.

Because `int` and `&int` are different types, we can't, for example, add them together:

```
let x = 5i;
let y = 8i;
let z = &y;

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

This gives us an error:

```
hello.rs:6:24: 6:25 error: mismatched types: expected `int` but found `&int` (expected int but fo
hello.rs:6     println!("{}", x + z);
                              ^
```

We can *dereference* the pointer by using the `*` operator. Dereferencing a pointer means accessing the value at the location stored in the pointer. This will work:

```
let x = 5i;
let y = 8i;
let z = &y;

println!("{}", x + *z);
```

It prints `13`.

That's it! That's all pointers are: they point to some memory location. Not much else to them. Now that we've discussed the *what* of pointers, let's talk about the *why*.

2.3.1.2 Pointer uses

Rust's pointers are quite useful, but in different ways than in other systems languages. We'll talk about best practices for Rust pointers later in the guide, but here are some ways that pointers are useful in other languages:

In C, strings are a pointer to a list of `char`s, ending with a null byte. The only way to use strings is to get quite familiar with pointers.

Pointers are useful to point to memory locations that are not on the stack. For example, our example used two stack variables, so we were able to give them names. But if we allocated some heap memory, we wouldn't have that name available. In C, `malloc` is used to allocate heap memory, and it returns a pointer.

As a more general variant of the previous two points, any time you have a structure that can change in size, you need a pointer. You can't tell at compile time how much memory to allocate, so you've gotta use a pointer to point at the memory where it will be allocated, and deal with it at run time.

Pointers are useful in languages that are pass-by-value, rather than pass-by-reference. Basically, languages can make two choices (this is made up syntax, it's not Rust):

```
func foo(x) {
    x = 5
}

func main() {
    i = 1
    foo(i)
    // what is the value of i here?
}
```

In languages that are pass-by-value, `foo` will get a copy of `i`, and so the original version of `i` is not modified. At the comment, `i` will still be `1`. In a language that is pass-by-reference, `foo` will get a reference to `i`, and therefore, can change its value. At the comment, `i` will be `5`.

So what do pointers have to do with this? Well, since pointers point to a location in memory…

```
func foo(&int x) {
    *x = 5
}

func main() {
    i = 1
    foo(&i)
    // what is the value of i here?
}
```

Even in a language which is pass by value, `i` will be `5` at the comment. You see, because the argument `x` is a pointer, we do send a copy over to `foo`, but because it points at a memory location, which we then assign to, the original value is still changed. This pattern is called *pass-reference-by-value*. Tricky!

2.3.1.3 Common pointer problems

We've talked about pointers, and we've sung their praises. So what's the downside? Well, Rust attempts to mitigate each of these kinds of problems, but here are problems with pointers in other languages:

Uninitialized pointers can cause a problem. For example, what does this program do?

```
&int x;
*x = 5; // whoops!
```

Who knows? We just declare a pointer, but don't point it at anything, and then set the memory location that it points at to be `5`. But which location? Nobody knows. This might be harmless, and it might be catastrophic.

When you combine pointers and functions, it's easy to accidentally invalidate the memory the pointer is pointing to. For example:

```
func make_pointer(): &int {
    x = 5;

    return &x;
}

func main() {
    &int i = make_pointer();
    *i = 5; // uh oh!
}
```

`x` is local to the `make_pointer` function, and therefore, is invalid as soon as `make_pointer` returns. But we return a pointer to its memory location, and so back in `main`, we try to use that pointer, and it's a very similar situation to our first one. Setting invalid memory locations is bad.

As one last example of a big problem with pointers, *aliasing* can be an issue. Two pointers are said to alias when they point at the same location in memory. Like this:

```
func mutate(&int i, int j) {
    *i = j;
}

func main() {
  x = 5;
  y = &x;
  z = &x; //y and z are aliased


  run_in_new_thread(mutate, y, 1);
  run_in_new_thread(mutate, z, 100);

  // what is the value of x here?
}
```

In this made-up example, `run_in_new_thread` spins up a new thread, and calls the given function name with its arguments. Since we have two threads, and they're both operating on aliases to `x`, we can't tell which one finishes first, and therefore, the value of `x` is actually non-deterministic. Worse, what if one of them had invalidated the memory location they pointed to? We'd have the same problem as before, where we'd be setting an invalid location.

2.3.1.4 Conclusion

That's a basic overview of pointers as a general concept. As we alluded to before, Rust has different kinds of pointers, rather than just one, and mitigates all of the problems that we talked about, too. This does mean that Rust pointers are slightly more complicated than in other languages, but it's worth it to not have the problems that simple pointers have.

2.3.2 References

The most basic type of pointer that Rust has is called a *reference*. Rust references look like this:

```
let x = 5i;
let y = &x;

println!("{}", *y);
println!("{:p}", y);
println!("{}", y);
```

We'd say "`y` is a reference to `x`." The first `println!` prints out the value of `y`'s referent by using the dereference operator, `*`. The second one prints out the memory location that `y` points to, by using the pointer format string. The third `println!` *also* prints out the value of `y`'s referent, because `println!` will automatically dereference it for us.

Here's a function that takes a reference:

```
fn succ(x: &int) -> int { *x + 1 }
```

You can also use `&` as an operator to create a reference, so we can call this function in two different ways:

```
fn succ(x: &int) -> int { *x + 1 }

fn main() {

    let x = 5i;
    let y = &x;

    println!("{}", succ(y));
    println!("{}", succ(&x));
}
```

Both of these `println!`s will print out `6`.

Of course, if this were real code, we wouldn't bother with the reference, and just write:

```
fn succ(x: int) -> int { x + 1 }
```

References are immutable by default:

```
let x = 5i;
let y = &x;

*y = 5; // error: cannot assign to immutable dereference of `&`-pointer `*y`
```

They can be made mutable with `mut`, but only if its referent is also mutable. This works:

```
let mut x = 5i;
let y = &mut x;
```

This does not:

```
let x = 5i;
let y = &mut x; // error: cannot borrow immutable local variable `x` as mutable
```

Immutable pointers are allowed to alias:

```
let x = 5i;
let y = &x;
```

```
let z = &x;
```

Mutable ones, however, are not:

```
let mut x = 5i;
let y = &mut x;
let z = &mut x; // error: cannot borrow `x` as mutable more than once at a time
```

Despite their complete safety, a reference's representation at runtime is the same as that of an ordinary pointer in a C program. They introduce zero overhead. The compiler does all safety checks at compile time. The theory that allows for this was originally called *region pointers*. Region pointers evolved into what we know today as *lifetimes*.

Here's the simple explanation: would you expect this code to compile?

```
fn main() {
    println!("{}", x);
    let x = 5;
}
```

Probably not. That's because you know that the name `x` is valid from where it's declared to when it goes out of scope. In this case, that's the end of the `main` function. So you know this code will cause an error. We call this duration a *lifetime*. Let's try a more complex example:

```
fn main() {
    let x = &mut 5i;

    if *x < 10 {
        let y = &x;

        println!("Oh no: {}", y);
        return;
    }

    *x -= 1;

    println!("Oh no: {}", x);
}
```

Here, we're borrowing a pointer to `x` inside of the `if`. The compiler, however, is able to determine that that pointer will go out of scope without `x` being mutated, and therefore, lets us pass. This wouldn't work:

```
fn main() {
    let x = &mut 5i;

    if *x < 10 {
        let y = &x;
        *x -= 1;

        println!("Oh no: {}", y);
        return;
    }

    *x -= 1;

    println!("Oh no: {}", x);
}
```

It gives this error:

```
test.rs:5:8: 5:10 error: cannot assign to `*x` because it is borrowed
test.rs:5         *x -= 1;
                  ^~
test.rs:4:16: 4:18 note: borrow of `*x` occurs here
test.rs:4         let y = &x;
                          ^~
```

As you might guess, this kind of analysis is complex for a human, and therefore hard for a computer, too! There is an entire guide devoted to references, ownership, and lifetimes that goes into this topic in great detail, so if you want the full details, check that out.

2.3.2.1 Best practices

In general, prefer stack allocation over heap allocation. Using references to stack allocated information is preferred whenever possible. Therefore, references are the default pointer type you should use, unless you have a specific reason to use a different type. The other types of pointers cover when they're appropriate to use in their own best practices sections.

Use references when you want to use a pointer, but do not want to take ownership. References just borrow ownership, which is more polite if you don't need the ownership. In other words, prefer:

```
fn succ(x: &int) -> int { *x + 1 }
```

to

```
fn succ(x: Box<int>) -> int { *x + 1 }
```

As a corollary to that rule, references allow you to accept a wide variety of other pointers, and so are useful so that you don't have to write a number of variants per pointer. In other words, prefer:

```
fn succ(x: &int) -> int { *x + 1 }
```

to

```
use std::rc::Rc;

fn box_succ(x: Box<int>) -> int { *x + 1 }

fn rc_succ(x: Rc<int>) -> int { *x + 1 }
```

Note that the caller of your function will have to modify their calls slightly:

```
# use std::boxed::Box;
use std::rc::Rc;

fn succ(x: &int) -> int { *x + 1 }

let ref_x = &5i;
let box_x = Box::new(5i);
let rc_x  = Rc::new(5i);

succ(ref_x);
succ(&*box_x);
succ(&*rc_x);
```

The initial `*` dereferences the pointer, and then `&` takes a reference to those contents.

2.3.3 Boxes

`Box<T>` is Rust's *boxed pointer* type. Boxes provide the simplest form of heap allocation in Rust. Creating a box looks like this:

```
# use std::boxed::Box;
let x = Box::new(5i);
```

Boxes are heap allocated and they are deallocated automatically by Rust when they go out of scope:

```
# use std::boxed::Box;
{
    let x = Box::new(5i);
```

```
    // stuff happens

} // x is destructed and its memory is free'd here
```

However, boxes do *not* use reference counting or garbage collection. Boxes are what's called an *affine type*. This means that the Rust compiler, at compile time, determines when the box comes into and goes out of scope, and inserts the appropriate calls there. Furthermore, boxes are a specific kind of affine type, known as a *region*. You can read more about regions in this paper on the Cyclone programming language.

You don't need to fully grok the theory of affine types or regions to grok boxes, though. As a rough approximation, you can treat this Rust code:

```
# use std::boxed::Box;
{
    let x = Box::new(5i);

    // stuff happens
}
```

As being similar to this C code:

```
{
    int *x;
    x = (int *)malloc(sizeof(int));
    *x = 5;

    // stuff happens

    free(x);
}
```

Of course, this is a 10,000 foot view. It leaves out destructors, for example. But the general idea is correct: you get the semantics of `malloc`/`free`, but with some improvements:

1. It's impossible to allocate the incorrect amount of memory, because Rust figures it out from the types.
2. You cannot forget to `free` memory you've allocated, because Rust does it for you.
3. Rust ensures that this `free` happens at the right time, when it is truly not used. Use-after-free is not possible.
4. Rust enforces that no other writeable pointers alias to this heap memory, which means writing to an invalid pointer is not possible.

See the section on references or the ownership guide for more detail on how lifetimes work.

Using boxes and references together is very common. For example:

```
# use std::boxed::Box;
fn add_one(x: &int) -> int {
    *x + 1
}

fn main() {
    let x = Box::new(5i);

    println!("{}", add_one(&*x));
}
```

In this case, Rust knows that `x` is being *borrowed* by the `add_one()` function, and since it's only reading the value, allows it.

We can borrow `x` multiple times, as long as it's not simultaneous:

```
# use std::boxed::Box;
fn add_one(x: &int) -> int {
    *x + 1
}
```

```
fn main() {
    let x = Box::new(5i);

    println!("{}", add_one(&*x));
    println!("{}", add_one(&*x));
    println!("{}", add_one(&*x));
}
```

Or as long as it's not a mutable borrow. This will error:

```
# use std::boxed::Box;
fn add_one(x: &mut int) -> int {
    *x + 1
}

fn main() {
    let x = Box::new(5i);

    println!("{}", add_one(&*x)); // error: cannot borrow immutable dereference
                                  // of `&`-pointer as mutable
}
```

Notice we changed the signature of `add_one()` to request a mutable reference.

2.3.3.1 Best practices

Boxes are appropriate to use in two situations: Recursive data structures, and occasionally, when returning data.

2.3.3.1.1 Recursive data structures

Sometimes, you need a recursive data structure. The simplest is known as a *cons list*:

```
# use std::boxed::Box;
#[derive(Show)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<int> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Cons(3, Box::new(Li
    println!("{:?}", list);
}
```

This prints:

```
Cons(1, Box(Cons(2, Box(Cons(3, Box(Nil))))))
```

The reference to another `List` inside of the `Cons` enum variant must be a box, because we don't know the length of the list. Because we don't know the length, we don't know the size, and therefore, we need to heap allocate our list.

Working with recursive or other unknown-sized data structures is the primary use-case for boxes.

2.3.3.1.2 Returning data

This is important enough to have its own section entirely. The TL;DR is this: you don't generally want to return pointers, even when you might in a language like C or C++.

See Returning Pointers below for more.

2.3.4 Rc and Arc

This part is coming soon.

2.3.4.1 Best practices

This part is coming soon.

2.3.5 Raw Pointers

This part is coming soon.

2.3.5.1 Best practices

This part is coming soon.

2.3.6 Returning Pointers

In many languages with pointers, you'd return a pointer from a function so as to avoid copying a large data structure. For example:

```
# use std::boxed::Box;
struct BigStruct {
    one: int,
    two: int,
    // etc
    one_hundred: int,
}

fn foo(x: Box<BigStruct>) -> Box<BigStruct> {
    return Box::new(*x);
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });

    let y = foo(x);
}
```

The idea is that by passing around a box, you're only copying a pointer, rather than the hundred `int`s that make up the `BigStruct`.

This is an antipattern in Rust. Instead, write this:

```
# use std::boxed::Box;
struct BigStruct {
    one: int,
    two: int,
    // etc
    one_hundred: int,
}

fn foo(x: Box<BigStruct>) -> BigStruct {
    return *x;
}

fn main() {
    let x = Box::new(BigStruct {
        one: 1,
        two: 2,
        one_hundred: 100,
    });
```

```
    let y = Box::new(foo(x));
}
```

This gives you flexibility without sacrificing performance.

You may think that this gives us terrible performance: return a value and then immediately box it up ?! Isn't that the worst of both worlds? Rust is smarter than that. There is no copy in this code. `main` allocates enough room for the `box`, passes a pointer to that memory into `foo` as `x`, and then `foo` writes the value straight into that pointer. This writes the return value directly into the allocated box.

This is important enough that it bears repeating: pointers are not for optimizing returning values from your code. Allow the caller to choose how they want to use your output.

2.3.7 Creating your own Pointers

This part is coming soon.

2.3.7.1 Best practices

This part is coming soon.

2.3.8 Patterns and `ref`

When you're trying to match something that's stored in a pointer, there may be a situation where matching directly isn't the best option available. Let's see how to properly handle this:

```
fn possibly_print(x: &Option<String>) {
    match *x {
        // BAD: cannot move out of a `&`
        Some(s) => println!("{}", s)

        // GOOD: instead take a reference into the memory of the `Option`
        Some(ref s) => println!("{}", *s),
        None => {}
    }
}
```

The `ref s` here means that `s` will be of type `&String`, rather than type `String`.

This is important when the type you're trying to get access to has a destructor and you don't want to move it, you just want a reference to it.

2.3.9 Cheat Sheet

Here's a quick rundown of Rust's pointer types:

Type	Name	Summary
`&T`	Reference	Allows one or more references to read `T`
`&mut T`	Mutable Reference	Allows a single reference to read and write `T`
`Box<T>`	Box	Heap allocated `T` with a single owner that may read and write `T`.
`Rc<T>`	"arr cee" pointer	Heap allocated `T` with many readers
`Arc<T>`	Arc pointer	Same as above, but safe sharing across threads
`*const T`	Raw pointer	Unsafe read access to `T`
`*mut T`	Mutable raw pointer	Unsafe read and write access to `T`

2.3.10 Related resources

- API documentation for Box
- Ownership guide
- Cyclone paper on regions, which inspired Rust's lifetime system

2.4 Ownership

This guide presents Rust's ownership system. This is one of Rust's most unique and compelling features, with which Rust developers should become quite acquainted. Ownership is how Rust achieves its largest goal, memory safety. The ownership system has a few distinct concepts: *ownership*, *borrowing*, and *lifetimes*. We'll talk about each one in turn.

2.4.1 Meta

Before we get to the details, two important notes about the ownership system.

Rust has a focus on safety and speed. It accomplishes these goals through many *zero-cost abstractions*, which means that in Rust, abstractions cost as little as possible in order to make them work. The ownership system is a prime example of a zero cost abstraction. All of the analysis we'll talk about in this guide is *done at compile time*. You do not pay any run-time cost for any of these features.

However, this system does have a certain cost: learning curve. Many new users to Rust experience something we like to call "fighting with the borrow checker," where the Rust compiler refuses to compile a program that the author thinks is valid. This often happens because the programmer's mental model of how ownership should work doesn't match the actual rules that Rust implements. You probably will experience similar things at first. There is good news, however: more experienced Rust developers report that once they work with the rules of the ownership system for a period of time, they fight the borrow checker less and less.

With that in mind, let's learn about ownership.

2.4.2 Ownership

At its core, ownership is about *resources*. For the purposes of the vast majority of this guide, we will talk about a specific resource: memory. The concept generalizes to any kind of resource, like a file handle, but to make it more concrete, we'll focus on memory.

When your program allocates some memory, it needs some way to deallocate that memory. Imagine a function `foo` that allocates four bytes of memory, and then never deallocates that memory. We call this problem *leaking* memory, because each time we call `foo`, we're allocating another four bytes. Eventually, with enough calls to `foo`, we will run our system out of memory. That's no good. So we need some way for `foo` to deallocate those four bytes. It's also important that we don't deallocate too many times, either. Without getting into the details, attempting to deallocate memory multiple times can lead to problems. In other words, any time some memory is allocated, we need to make sure that we deallocate that memory once and only once. Too many times is bad, not enough times is bad. The counts must match.

There's one other important detail with regards to allocating memory. Whenever we request some amount of memory, what we are given is a handle to that memory. This handle (often called a *pointer*, when we're referring to memory) is how we interact with the allocated memory. As long as we have that handle, we can do something with the memory. Once we're done with the handle, we're also done with the memory, as we can't do anything useful without a handle to it.

Historically, systems programming languages require you to track these allocations, deallocations, and handles yourself. For example, if we want some memory from the heap in a language like C, we do this:

```
{
    int *x = malloc(sizeof(int));

    // we can now do stuff with our handle x
```

```
    *x = 5;

    free(x);
}
```

The call to `malloc` allocates some memory. The call to `free` deallocates the memory. There's also bookkeeping about allocating the correct amount of memory.

Rust combines these two aspects of allocating memory (and other resources) into a concept called *ownership*. Whenever we request some memory, that handle we receive is called the *owning handle*. Whenever that handle goes out of scope, Rust knows that you cannot do anything with the memory anymore, and so therefore deallocates the memory for you. Here's the equivalent example in Rust:

```
# use std::boxed::Box;
{
    let x = Box::new(5i);
}
```

The `Box::new` function creates a `Box<T>` (specifically `Box<int>` in this case) by allocating a small segment of memory on the heap with enough space to fit an `int`. But where in the code is the box deallocated? We said before that we must have a deallocation for each allocation. Rust handles this for you. It knows that our handle, `x`, is the owning reference to our box. Rust knows that `x` will go out of scope at the end of the block, and so it inserts a call to deallocate the memory at the end of the scope. Because the compiler does this for us, it's impossible to forget. We always have exactly one deallocation paired with each of our allocations.

This is pretty straightforward, but what happens when we want to pass our box to a function? Let's look at some code:

```
# use std::boxed::Box;
fn main() {
    let x = Box::new(5i);

    add_one(x);
}

fn add_one(mut num: Box<int>) {
    *num += 1;
}
```

This code works, but it's not ideal. For example, let's add one more line of code, where we print out the value of `x`:

```
# use std::boxed::Box;
fn main() {
    let x = Box::new(5i);

    add_one(x);

    println!("{}", x);
}

fn add_one(mut num: Box<int>) {
    *num += 1;
}
```

This does not compile, and gives us an error:

```
error: use of moved value: `x`
   println!("{}", x);
                  ^
```

Remember, we need one deallocation for every allocation. When we try to pass our box to `add_one`, we would have two handles to the memory: `x` in `main`, and `num` in `add_one`. If we deallocated the memory when each handle went out of scope, we would have two deallocations and one allocation, and that's wrong. So when we call

`add_one`, Rust defines `num` as the owner of the handle. And so, now that we've given ownership to `num`, `x` is invalid. `x`'s value has "moved" from `x` to `num`. Hence the error: use of moved value `x`.

To fix this, we can have `add_one` give ownership back when it's done with the box:

```
# use std::boxed::Box;
fn main() {
    let x = Box::new(5i);

    let y = add_one(x);

    println!("{}", y);
}

fn add_one(mut num: Box<int>) -> Box<int> {
    *num += 1;

    num
}
```

This code will compile and run just fine. Now, we return a `box`, and so the ownership is transferred back to `y` in `main`. We only have ownership for the duration of our function before giving it back. This pattern is very common, and so Rust introduces a concept to describe a handle which temporarily refers to something another handle owns. It's called *borrowing*, and it's done with *references*, designated by the `&` symbol.

2.4.3 Borrowing

Here's the current state of our `add_one` function:

```
fn add_one(mut num: Box<int>) -> Box<int> {
    *num += 1;

    num
}
```

This function takes ownership, because it takes a `Box`, which owns its contents. But then we give ownership right back.

In the physical world, you can give one of your possessions to someone for a short period of time. You still own your possession, you're just letting someone else use it for a while. We call that *lending* something to someone, and that person is said to be *borrowing* that something from you.

Rust's ownership system also allows an owner to lend out a handle for a limited period. This is also called *borrowing*. Here's a version of `add_one` which borrows its argument rather than taking ownership:

```
fn add_one(num: &mut int) {
    *num += 1;
}
```

This function borrows an `int` from its caller, and then increments it. When the function is over, and `num` goes out of scope, the borrow is over.

2.4.4 Lifetimes

Lending out a reference to a resource that someone else owns can be complicated, however. For example, imagine this set of operations:

1. I acquire a handle to some kind of resource.
2. I lend you a reference to the resource.
3. I decide I'm done with the resource, and deallocate it, while you still have your reference.
4. You decide to use the resource.

Uh oh! Your reference is pointing to an invalid resource. This is called a *dangling pointer* or "use after free," when the resource is memory.

To fix this, we have to make sure that step four never happens after step three. The ownership system in Rust does this through a concept called *lifetimes*, which describe the scope that a reference is valid for.

Let's look at that function which borrows an `int` again:

```
fn add_one(num: &int) -> int {
    *num + 1
}
```

Rust has a feature called *lifetime elision*, which allows you to not write lifetime annotations in certain circumstances. This is one of them. We will cover the others later. Without eliding the lifetimes, `add_one` looks like this:

```
fn add_one<'a>(num: &'a int) -> int {
    *num + 1
}
```

The `'a` is called a *lifetime*. Most lifetimes are used in places where short names like `'a`, `'b` and `'c` are clearest, but it's often useful to have more descriptive names. Let's dig into the syntax in a bit more detail:

```
fn add_one<'a>(...)
```

This part *declares* our lifetimes. This says that `add_one` has one lifetime, `'a`. If we had two, it would look like this:

```
fn add_two<'a, 'b>(...)
```

Then in our parameter list, we use the lifetimes we've named:

```
...(num: &'a int) -> ...
```

If you compare `&int` to `&'a int`, they're the same, it's just that the lifetime `'a` has snuck in between the `&` and the `int`. We read `&int` as "a reference to an int" and `&'a int` as "a reference to an int with the lifetime 'a.'"

Why do lifetimes matter? Well, for example, here's some code:

```
struct Foo<'a> {
    x: &'a int,
}

fn main() {
    let y = &5i; // this is the same as `let _y = 5; let y = &_y;
    let f = Foo { x: y };

    println!("{}", f.x);
}
```

As you can see, `struct`s can also have lifetimes. In a similar way to functions,

```
struct Foo<'a> {
# x: &'a int,
# }
```

declares a lifetime, and

```
# struct Foo<'a> {
x: &'a int,
# }
```

uses it. So why do we need a lifetime here? We need to ensure that any reference to a `Foo` cannot outlive the reference to an `int` it contains.

2.4.4.1 Thinking in scopes

A way to think about lifetimes is to visualize the scope that a reference is valid for. For example:

```
fn main() {
    let y = &5i;     // -+ y goes into scope
                     //  |
    // stuff         //  |
                     //  |
}                    // -+ y goes out of scope
```

Adding in our `Foo`:

```
struct Foo<'a> {
    x: &'a int,
}

fn main() {
    let y = &5i;           // -+ y goes into scope
    let f = Foo { x: y }; // -+ f goes into scope
    // stuff               //  |
                           //  |
}                          // -+ f and y go out of scope
```

Our `f` lives within the scope of `y`, so everything works. What if it didn't? This code won't work:

```
struct Foo<'a> {
    x: &'a int,
}

fn main() {
    let x;                     // -+ x goes into scope
                               //  |
    {                          //  |
        let y = &5i;           // ---+ y goes into scope
        let f = Foo { x: y }; // ---+ f goes into scope
        x = &f.x;              //  | | error here
    }                          // ---+ f and y go out of scope
                               //  |
    println!("{}", x);         //  |
}                              // -+ x goes out of scope
```

Whew! As you can see here, the scopes of `f` and `y` are smaller than the scope of `x`. But when we do `x = &f.x`, we make `x` a reference to something that's about to go out of scope.

Named lifetimes are a way of giving these scopes a name. Giving something a name is the first step towards being able to talk about it.

2.4.4.2 'static

The lifetime named *static* is a special lifetime. It signals that something has the lifetime of the entire program. Most Rust programmers first come across `'static` when dealing with strings:

```
let x: &'static str = "Hello, world.";
```

String literals have the type `&'static str` because the reference is always alive: they are baked into the data segment of the final binary. Another example are globals:

```
static FOO: int = 5i;
let x: &'static int = &FOO;
```

This adds an `int` to the data segment of the binary, and FOO is a reference to it.

2.4.5 Shared Ownership

In all the examples we've considered so far, we've assumed that each handle has a singular owner. But sometimes, this doesn't work. Consider a car. Cars have four wheels. We would want a wheel to know which car it was attached to. But this won't work:

```
struct Car {
    name: String,
}

struct Wheel {
    size: int,
    owner: Car,
}

fn main() {
    let car = Car { name: "DeLorean".to_string() };

    for _ in range(0u, 4) {
        Wheel { size: 360, owner: car };
    }
}
```

We try to make four `Wheel`s, each with a `Car` that it's attached to. But the compiler knows that on the second iteration of the loop, there's a problem:

```
error: use of moved value: `car`
    Wheel { size: 360, owner: car };
                              ^~~
note: `car` moved here because it has type `Car`, which is non-copyable
    Wheel { size: 360, owner: car };
                              ^~~
```

We need our `Car` to be pointed to by multiple `Wheel`s. We can't do that with `Box<T>`, because it has a single owner. We can do it with `Rc<T>` instead:

```
use std::rc::Rc;

struct Car {
    name: String,
}

struct Wheel {
    size: int,
    owner: Rc<Car>,
}

fn main() {
    let car = Car { name: "DeLorean".to_string() };

    let car_owner = Rc::new(car);

    for _ in range(0u, 4) {
        Wheel { size: 360, owner: car_owner.clone() };
    }
}
```

We wrap our `Car` in an `Rc<T>`, getting an `Rc<Car>`, and then use the `clone()` method to make new references. We've also changed our `Wheel` to have an `Rc<Car>` rather than just a `Car`.

This is the simplest kind of multiple ownership possible. For example, there's also `Arc<T>`, which uses more expensive atomic instructions to be the thread-safe counterpart of `Rc<T>`.

2.4.5.1 Lifetime Elision

Earlier, we mentioned *lifetime elision*, a feature of Rust which allows you to not write lifetime annotations in certain circumstances. All references have a lifetime, and so if you elide a lifetime (like `&T` instead of `&'a T`), Rust will do three things to determine what those lifetimes should be.

When talking about lifetime elision, we use the term *input lifetime* and *output lifetime*. An *input lifetime* is a lifetime associated with a parameter of a function, and an *output lifetime* is a lifetime associated with the return value of a function. For example, this function has an input lifetime:

```
fn foo<'a>(bar: &'a str)
```

This one has an output lifetime:

```
fn foo<'a>() -> &'a str
```

This one has a lifetime in both positions:

```
fn foo<'a>(bar: &'a str) -> &'a str
```

Here are the three rules:

- Each elided lifetime in a function's arguments becomes a distinct lifetime parameter.
- If there is exactly one input lifetime, elided or not, that lifetime is assigned to all elided lifetimes in the return values of that function.
- If there are multiple input lifetimes, but one of them is `&self` or `&mut self`, the lifetime of `self` is assigned to all elided output lifetimes.

Otherwise, it is an error to elide an output lifetime.

2.4.5.1.1 Examples

Here are some examples of functions with elided lifetimes, and the version of what the elided lifetimes are expand to:

```
fn print(s: &str);                                    // elided
fn print<'a>(s: &'a str);                             // expanded

fn debug(lvl: uint, s: &str);                         // elided
fn debug<'a>(lvl: uint, s: &'a str);                  // expanded

// In the preceeding example, `lvl` doesn't need a lifetime because it's not a
// reference (`&`). Only things relating to references (such as a `struct`
// which contains a reference) need lifetimes.

fn substr(s: &str, until: uint) -> &str;              // elided
fn substr<'a>(s: &'a str, until: uint) -> &'a str;    // expanded

fn get_str() -> &str;                                 // ILLEGAL, no inputs

fn frob(s: &str, t: &str) -> &str;                    // ILLEGAL, two inputs

fn get_mut(&mut self) -> &mut T;                      // elided
fn get_mut<'a>(&'a mut self) -> &'a mut T;            // expanded

fn args<T:ToCStr>(&mut self, args: &[T]) -> &mut Command                  // elided
fn args<'a, 'b, T:ToCStr>(&'a mut self, args: &'b [T]) -> &'a mut Command // expanded

fn new(buf: &mut [u8]) -> BufWriter;                  // elided
fn new<'a>(buf: &'a mut [u8]) -> BufWriter<'a>        // expanded
```

2.4.6 Related Resources

Coming Soon.

2.5 Patterns

We've made use of patterns a few times in the guide: first with `let` bindings, then with `match` statements. Let's go on a whirlwind tour of all of the things patterns can do!

A quick refresher: you can match against literals directly, and _ acts as an *any* case:

```
let x = 1i;

match x {
    1 => println!("one"),
    2 => println!("two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

You can match multiple patterns with `|`:

```
let x = 1i;

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

You can match a range of values with `...`:

```
let x = 1i;

match x {
    1 ... 5 => println!("one through five"),
    _ => println!("anything"),
}
```

Ranges are mostly used with integers and single characters.

If you're matching multiple things, via a `|` or a `...`, you can bind the value to a name with `@`:

```
let x = 1i;

match x {
    e @ 1 ... 5 => println!("got a range element {}", e),
    _ => println!("anything"),
}
```

If you're matching on an enum which has variants, you can use `..` to ignore the value and type in the variant:

```
enum OptionalInt {
    Value(int),
    Missing,
}

let x = OptionalInt::Value(5i);

match x {
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing   => println!("No such luck."),
}
```

You can introduce *match guards* with `if`:

```
enum OptionalInt {
    Value(int),
    Missing,
}

let x = OptionalInt::Value(5i);

match x {
```

```
    OptionalInt::Value(i) if i > 5 => println!("Got an int bigger than five!"),
    OptionalInt::Value(..) => println!("Got an int!"),
    OptionalInt::Missing   => println!("No such luck."),
}
```

If you're matching on a pointer, you can use the same syntax as you declared it with. First, `&`:

```
let x = &5i;

match x {
    &val => println!("Got a value: {}", val),
}
```

Here, the `val` inside the `match` has type `int`. In other words, the left-hand side of the pattern destructures the value. If we have `&5i`, then in `&val`, `val` would be `5i`.

If you want to get a reference, use the `ref` keyword:

```
let x = 5i;

match x {
    ref r => println!("Got a reference to {}", r),
}
```

Here, the `r` inside the `match` has the type `&int`. In other words, the `ref` keyword *creates* a reference, for use in the pattern. If you need a mutable reference, `ref mut` will work in the same way:

```
let mut x = 5i;

match x {
    ref mut mr => println!("Got a mutable reference to {}", mr),
}
```

If you have a struct, you can destructure it inside of a pattern:

```
# #![allow(non_shorthand_field_patterns)]
struct Point {
    x: int,
    y: int,
}

let origin = Point { x: 0i, y: 0i };

match origin {
    Point { x: x, y: y } => println!("({},{})", x, y),
}
```

If we only care about some of the values, we don't have to give them all names:

```
# #![allow(non_shorthand_field_patterns)]
struct Point {
    x: int,
    y: int,
}

let origin = Point { x: 0i, y: 0i };

match origin {
    Point { x: x, .. } => println!("x is {}", x),
}
```

You can do this kind of match on any member, not just the first:

```
# #![allow(non_shorthand_field_patterns)]
struct Point {
    x: int,
    y: int,
}
```

```
let origin = Point { x: 0i, y: 0i };

match origin {
    Point { y: y, .. } => println!("y is {}", y),
}
```

If you want to match against a slice or array, you can use `[]`:

```
fn main() {
    let v = vec!["match_this", "1"];

    match v.as_slice() {
        ["match_this", second] => println!("The second element is {}", second),
        _ => {},
    }
}
```

Whew! That's a lot of different ways to match things, and they can all be mixed and matched, depending on what you're doing:

```
match x {
    Foo { x: Some(ref name), y: None } => ...
}
```

Patterns are very powerful. Make good use of them.

2.6 Method Syntax

Functions are great, but if you want to call a bunch of them on some data, it can be awkward. Consider this code:

```
baz(bar(foo(x)));
```

We would read this left-to right, and so we see "baz bar foo." But this isn't the order that the functions would get called in, that's inside-out: "foo bar baz." Wouldn't it be nice if we could do this instead?

```
x.foo().bar().baz();
```

Luckily, as you may have guessed with the leading question, you can! Rust provides the ability to use this *method call syntax* via the `impl` keyword.

Here's how it works:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

fn main() {
    let c = Circle { x: 0.0, y: 0.0, radius: 2.0 };
    println!("{}", c.area());
}
```

This will print `12.566371`.

We've made a struct that represents a circle. We then write an `impl` block, and inside it, define a method, `area`. Methods take a special first parameter, `&self`. There are three variants: `self`, `&self`, and `&mut self`. You can think of this first parameter as being the `x` in `x.foo()`. The three variants correspond to the three kinds of thing `x` could

be: `self` if it's just a value on the stack, `&self` if it's a reference, and `&mut self` if it's a mutable reference. We should default to using `&self`, as it's the most common.

Finally, as you may remember, the value of the area of a circle is $\pi*r^2$. Because we took the `&self` parameter to `area`, we can use it just like any other parameter. Because we know it's a `Circle`, we can access the `radius` just like we would with any other struct. An import of π and some multiplications later, and we have our area.

You can also define methods that do not take a `self` parameter. Here's a pattern that's very common in Rust code:

```
# #![allow(non_shorthand_field_patterns)]
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn new(x: f64, y: f64, radius: f64) -> Circle {
        Circle {
            x: x,
            y: y,
            radius: radius,
        }
    }
}

fn main() {
    let c = Circle::new(0.0, 0.0, 2.0);
}
```

This *static method* builds a new `Circle` for us. Note that static methods are called with the `Struct::method()` syntax, rather than the `ref.method()` syntax.

2.7 Closures

So far, we've made lots of functions in Rust, but we've given them all names. Rust also allows us to create anonymous functions. Rust's anonymous functions are called *closures*. By themselves, closures aren't all that interesting, but when you combine them with functions that take closures as arguments, really powerful things are possible.

Let's make a closure:

```
let add_one = |&: x| { 1 + x };

println!("The sum of 5 plus 1 is {}.", add_one(5));
```

We create a closure using the `|...| { ... }` syntax, and then we create a binding so we can use it later. Note that we call the function using the binding name and two parentheses, just like we would for a named function.

Let's compare syntax. The two are pretty close:

```
let add_one = |&: x: i32| -> i32 { 1 + x };
fn  add_one      (x: i32) -> i32 { 1 + x }
```

As you may have noticed, closures infer their argument and return types, so you don't need to declare one. This is different from named functions, which default to returning unit (`()`).

There's one big difference between a closure and named functions, and it's in the name: a closure "closes over its environment." What does that mean? It means this:

```
fn main() {
    let x: i32 = 5;
```

```
    let printer = |&:| { println!("x is: {}", x); };

    printer(); // prints "x is: 5"
}
```

The `||` syntax means this is an anonymous closure that takes no arguments. Without it, we'd just have a block of code in `{}`s.

In other words, a closure has access to variables in the scope where it's defined. The closure borrows any variables it uses, so this will error:

```
fn main() {
    let mut x = 5;

    let printer = |&:| { println!("x is: {}", x); };

    x = 6; // error: cannot assign to `x` because it is borrowed
}
```

2.7.0.1 Moving closures

Rust has a second type of closure, called a *moving closure*. Moving closures are indicated using the `move` keyword (e.g., `move || x * x`). The difference between a moving closure and an ordinary closure is that a moving closure always takes ownership of all variables that it uses. Ordinary closures, in contrast, just create a reference into the enclosing stack frame. Moving closures are most useful with Rust's concurrency features, and so we'll just leave it at this for now. We'll talk about them more in the "Threads" section of the guide.

2.7.0.2 Accepting closures as arguments

Closures are most useful as an argument to another function. Here's an example:

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
    f(x) + f(x)
}

fn main() {
    let square = |&: x: i32| { x * x };

    twice(5, square); // evaluates to 50
}
```

Let's break the example down, starting with `main`:

```
let square = |&: x: i32| { x * x };
```

We've seen this before. We make a closure that takes an integer, and returns its square.

```
# fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 { f(x) + f(x) }
# let square = |&: x: i32| { x * x };
twice(5, square); // evaluates to 50
```

This line is more interesting. Here, we call our function, `twice`, and we pass it two arguments: an integer, `5`, and our closure, `square`. This is just like passing any other two variable bindings to a function, but if you've never worked with closures before, it can seem a little complex. Just think: "I'm passing two variables: one is an i32, and one is a function."

Next, let's look at how `twice` is defined:

```
fn twice(x: i32, f: |i32| -> i32) -> i32 {
```

`twice` takes two arguments, `x` and `f`. That's why we called it with two arguments. `x` is an `i32`, we've done that a ton of times. `f` is a function, though, and that function takes an `i32` and returns an `i32`. Notice how the `|i32| ->`

`i32` syntax looks a lot like our definition of `square` above, if we added the return type in:

```
let square = |&: x: i32| -> i32 { x * x };
//           |i32|       -> i32
```

This function takes an `i32` and returns an `i32`.

This is the most complicated function signature we've seen yet! Give it a read a few times until you can see how it works. It takes a teeny bit of practice, and then it's easy.

Finally, `twice` returns an `i32` as well.

Okay, let's look at the body of `twice`:

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
  f(x) + f(x)
}
```

Since our closure is named `f`, we can call it just like we called our closures before, and we pass in our `x` argument to each one, hence the name `twice`.

If you do the math, `(5 * 5) + (5 * 5) == 50`, so that's the output we get.

Play around with this concept until you're comfortable with it. Rust's standard library uses lots of closures where appropriate, so you'll be using this technique a lot.

If we didn't want to give `square` a name, we could just define it inline. This example is the same as the previous one:

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
    f(x) + f(x)
}

fn main() {
    twice(5, |x: i32| { x * x }); // evaluates to 50
}
```

A named function's name can be used wherever you'd use a closure. Another way of writing the previous example:

```
fn twice<F: Fn(i32) -> i32>(x: i32, f: F) -> i32 {
    f(x) + f(x)
}

fn square(x: i32) -> i32 { x * x }

fn main() {
    twice(5, square); // evaluates to 50
}
```

Doing this is not particularly common, but it's useful every once in a while.

That's all you need to get the hang of closures! Closures are a little bit strange at first, but once you're used to them, you'll miss them in other languages. Passing functions to other functions is incredibly powerful, as you will see in the following chapter about iterators.

2.8 Iterators

Let's talk about loops.

Remember Rust's `for` loop? Here's an example:

```
for x in range(0i, 10i) {
    println!("{}", x);
}
```

Now that you know more Rust, we can talk in detail about how this works. The `range` function returns an *iterator*. An iterator is something that we can call the `.next()` method on repeatedly, and it gives us a sequence of things.

Like this:

```
let mut range = range(0i, 10i);

loop {
    match range.next() {
        Some(x) => {
            println!("{}", x);
        },
        None => { break }
    }
}
```

We make a mutable binding to the return value of `range`, which is our iterator. We then `loop`, with an inner `match`. This `match` is used on the result of `range.next()`, which gives us a reference to the next value of the iterator. `next` returns an `Option<int>`, in this case, which will be `Some(int)` when we have a value and `None` once we run out. If we get `Some(int)`, we print it out, and if we get `None`, we `break` out of the loop.

This code sample is basically the same as our `for` loop version. The `for` loop is just a handy way to write this `loop`/`match`/`break` construct.

`for` loops aren't the only thing that uses iterators, however. Writing your own iterator involves implementing the `Iterator` trait. While doing that is outside of the scope of this guide, Rust provides a number of useful iterators to accomplish various tasks. Before we talk about those, we should talk about a Rust anti-pattern. And that's `range`.

Yes, we just talked about how `range` is cool. But `range` is also very primitive. For example, if you needed to iterate over the contents of a vector, you may be tempted to write this:

```
let nums = vec![1i, 2i, 3i];

for i in range(0u, nums.len()) {
    println!("{}", nums[i]);
}
```

This is strictly worse than using an actual iterator. The `.iter()` method on vectors returns an iterator which iterates through a reference to each element of the vector in turn. So write this:

```
let nums = vec![1i, 2i, 3i];

for num in nums.iter() {
    println!("{}", num);
}
```

There are two reasons for this. First, this more directly expresses what we mean. We iterate through the entire vector, rather than iterating through indexes, and then indexing the vector. Second, this version is more efficient: the first version will have extra bounds checking because it used indexing, `nums[i]`. But since we yield a reference to each element of the vector in turn with the iterator, there's no bounds checking in the second example. This is very common with iterators: we can ignore unnecessary bounds checks, but still know that we're safe.

There's another detail here that's not 100% clear because of how `println!` works. `num` is actually of type `&int`. That is, it's a reference to an `int`, not an `int` itself. `println!` handles the dereferencing for us, so we don't see it. This code works fine too:

```
let nums = vec![1i, 2i, 3i];

for num in nums.iter() {
```

```
    println!("{}", *num);
}
```

Now we're explicitly dereferencing `num`. Why does `iter()` give us references? Well, if it gave us the data itself, we would have to be its owner, which would involve making a copy of the data and giving us the copy. With references, we're just borrowing a reference to the data, and so it's just passing a reference, without needing to do the copy.

So, now that we've established that `range` is often not what you want, let's talk about what you do want instead.

There are three broad classes of things that are relevant here: iterators, *iterator adapters*, and *consumers*. Here's some definitions:

- *iterators* give you a sequence of values.
- *iterator adapters* operate on an iterator, producing a new iterator with a different output sequence.
- *consumers* operate on an iterator, producing some final set of values.

Let's talk about consumers first, since you've already seen an iterator, `range`.

2.8.0.1 Consumers

A *consumer* operates on an iterator, returning some kind of value or values. The most common consumer is `collect()`. This code doesn't quite compile, but it shows the intention:

```
let one_to_one_hundred = range(1i, 101i).collect();
```

As you can see, we call `collect()` on our iterator. `collect()` takes as many values as the iterator will give it, and returns a collection of the results. So why won't this compile? Rust can't determine what type of things you want to collect, and so you need to let it know. Here's the version that does compile:

```
let one_to_one_hundred = range(1i, 101i).collect::<Vec<int>>();
```

If you remember, the `::<>` syntax allows us to give a type hint, and so we tell it that we want a vector of integers.

`collect()` is the most common consumer, but there are others too. `find()` is one:

```
let greater_than_forty_two = range(0i, 100i)
                             .find(|x| *x > 42);

match greater_than_forty_two {
    Some(_) => println!("We got some numbers!"),
    None    => println!("No numbers found :("),
}
```

`find` takes a closure, and works on a reference to each element of an iterator. This closure returns `true` if the element is the element we're looking for, and `false` otherwise. Because we might not find a matching element, `find` returns an `Option` rather than the element itself.

Another important consumer is `fold`. Here's what it looks like:

```
let sum = range(1i, 4i)
              .fold(0i, |sum, x| sum + x);
```

`fold()` is a consumer that looks like this: `fold(base, |accumulator, element| ...)`. It takes two arguments: the first is an element called the *base*. The second is a closure that itself takes two arguments: the first is called the *accumulator*, and the second is an *element*. Upon each iteration, the closure is called, and the result is the value of the accumulator on the next iteration. On the first iteration, the base is the value of the accumulator.

Okay, that's a bit confusing. Let's examine the values of all of these things in this iterator:

base	accumulator	element	closure result
0i	0i	1i	1i
0i	1i	2i	3i
0i	3i	3i	6i

We called `fold()` with these arguments:

```
# range(1i, 4i)
.fold(0i, |sum, x| sum + x);
```

So, `0i` is our base, `sum` is our accumulator, and `x` is our element. On the first iteration, we set `sum` to `0i`, and `x` is the first element of `nums`, `1i`. We then add `sum` and `x`, which gives us `0i + 1i = 1i`. On the second iteration, that value becomes our accumulator, `sum`, and the element is the second element of the array, `2i`. `1i + 2i = 3i`, and so that becomes the value of the accumulator for the last iteration. On that iteration, `x` is the last element, `3i`, and `3i + 3i = 6i`, which is our final result for our sum. `1 + 2 + 3 = 6`, and that's the result we got.

Whew. `fold` can be a bit strange the first few times you see it, but once it clicks, you can use it all over the place. Any time you have a list of things, and you want a single result, `fold` is appropriate.

Consumers are important due to one additional property of iterators we haven't talked about yet: laziness. Let's talk some more about iterators, and you'll see why consumers matter.

2.8.0.2 Iterators

As we've said before, an iterator is something that we can call the `.next()` method on repeatedly, and it gives us a sequence of things. Because you need to call the method, this means that iterators are *lazy* and don't need to generate all of the values upfront. This code, for example, does not actually generate the numbers `1-100`, and just creates a value that represents the sequence:

```
let nums = range(1i, 100i);
```

Since we didn't do anything with the range, it didn't generate the sequence. Let's add the consumer:

```
let nums = range(1i, 100i).collect::<Vec<int>>();
```

Now, `collect()` will require that `range()` give it some numbers, and so it will do the work of generating the sequence.

`range` is one of two basic iterators that you'll see. The other is `iter()`, which you've used before. `iter()` can turn a vector into a simple iterator that gives you each element in turn:

```
let nums = [1i, 2i, 3i];

for num in nums.iter() {
   println!("{}", num);
}
```

These two basic iterators should serve you well. There are some more advanced iterators, including ones that are infinite. Like `count`:

```
std::iter::count(1i, 5i);
```

This iterator counts up from one, adding five each time. It will give you a new integer every time, forever (well, technically, until it reaches the maximum number representable by an `int`). But since iterators are lazy, that's okay! You probably don't want to use `collect()` on it, though…

That's enough about iterators. Iterator adapters are the last concept we need to talk about with regards to iterators. Let's get to it!

2.8.0.3 Iterator adapters

Iterator adapters take an iterator and modify it somehow, producing a new iterator. The simplest one is called `map`:

```
range(1i, 100i).map(|x| x + 1i);
```

`map` is called upon another iterator, and produces a new iterator where each element reference has the closure it's been given as an argument called on it. So this would give us the numbers from `2-100`. Well, almost! If you compile the example, you'll get a warning:

```
warning: unused result which must be used: iterator adaptors are lazy and
         do nothing unless consumed, #[warn(unused_must_use)] on by default
 range(1i, 100i).map(|x| x + 1i);
 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Laziness strikes again! That closure will never execute. This example doesn't print any numbers:

```
range(1i, 100i).map(|x| println!("{}", x));
```

If you are trying to execute a closure on an iterator for its side effects, just use `for` instead.

There are tons of interesting iterator adapters. `take(n)` will return an iterator over the next `n` elements of the original iterator, note that this has no side effect on the original iterator. Let's try it out with our infinite iterator from before, `count()`:

```
for i in std::iter::count(1i, 5i).take(5) {
    println!("{}", i);
}
```

This will print

```
1
6
11
16
21
```

`filter()` is an adapter that takes a closure as an argument. This closure returns `true` or `false`. The new iterator `filter()` produces only the elements that that closure returns `true` for:

```
for i in range(1i, 100i).filter(|&x| x % 2 == 0) {
    println!("{}", i);
}
```

This will print all of the even numbers between one and a hundred. (Note that because `filter` doesn't consume the elements that are being iterated over, it is passed a reference to each element, and thus the filter predicate uses the `&x` pattern to extract the integer itself.)

You can chain all three things together: start with an iterator, adapt it a few times, and then consume the result. Check it out:

```
range(1i, 1000i)
    .filter(|&x| x % 2 == 0)
    .filter(|&x| x % 3 == 0)
    .take(5)
    .collect::<Vec<int>>();
```

This will give you a vector containing `6`, `12`, `18`, `24`, and `30`.

This is just a small taste of what iterators, iterator adapters, and consumers can help you with. There are a number of really useful iterators, and you can write your own as well. Iterators provide a safe, efficient way to manipulate all kinds of lists. They're a little unusual at first, but if you play with them, you'll get hooked. For a full list of the different iterators and consumers, check out the iterator module documentation.

2.9 Generics

Sometimes, when writing a function or data type, we may want it to work for multiple types of arguments. For example, remember our `OptionalInt` type?

```
enum OptionalInt {
    Value(int),
    Missing,
}
```

If we wanted to also have an `OptionalFloat64`, we would need a new enum:

```
enum OptionalFloat64 {
    Valuef64(f64),
    Missingf64,
}
```

This is really unfortunate. Luckily, Rust has a feature that gives us a better way: generics. Generics are called *parametric polymorphism* in type theory, which means that they are types or functions that have multiple forms (*poly* is multiple, *morph* is form) over a given parameter (*parametric*).

Anyway, enough with type theory declarations, let's check out the generic form of `OptionalInt`. It is actually provided by Rust itself, and looks like this:

```
enum Option<T> {
    Some(T),
    None,
}
```

The `<T>` part, which you've seen a few times before, indicates that this is a generic data type. Inside the declaration of our enum, wherever we see a `T`, we substitute that type for the same type used in the generic. Here's an example of using `Option<T>`, with some extra type annotations:

```
let x: Option<int> = Some(5i);
```

In the type declaration, we say `Option<int>`. Note how similar this looks to `Option<T>`. So, in this particular `Option`, `T` has the value of `int`. On the right-hand side of the binding, we do make a `Some(T)`, where `T` is `5i`. Since that's an `int`, the two sides match, and Rust is happy. If they didn't match, we'd get an error:

```
let x: Option<f64> = Some(5i);
// error: mismatched types: expected `core::option::Option<f64>`
// but found `core::option::Option<int>` (expected f64 but found int)
```

That doesn't mean we can't make `Option<T>`s that hold an `f64`! They just have to match up:

```
let x: Option<int> = Some(5i);
let y: Option<f64> = Some(5.0f64);
```

This is just fine. One definition, multiple uses.

Generics don't have to only be generic over one type. Consider Rust's built-in `Result<T, E>` type:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

This type is generic over *two* types: `T` and `E`. By the way, the capital letters can be any letter you'd like. We could define `Result<T, E>` as:

```
enum Result<H, N> {
    Ok(H),
```

```
    Err(N),
}
```

if we wanted to. Convention says that the first generic parameter should be `T`, for 'type,' and that we use `E` for 'error.' Rust doesn't care, however.

The `Result<T, E>` type is intended to be used to return the result of a computation, and to have the ability to return an error if it didn't work out. Here's an example:

```
let x: Result<f64, String> = Ok(2.3f64);
let y: Result<f64, String> = Err("There was an error.".to_string());
```

This particular Result will return an `f64` if there's a success, and a `String` if there's a failure. Let's write a function that uses `Result<T, E>`:

```
fn inverse(x: f64) -> Result<f64, String> {
    if x == 0.0f64 { return Err("x cannot be zero!".to_string()); }

    Ok(1.0f64 / x)
}
```

We don't want to take the inverse of zero, so we check to make sure that we weren't passed zero. If we were, then we return an `Err`, with a message. If it's okay, we return an `Ok`, with the answer.

Why does this matter? Well, remember how `match` does exhaustive matches? Here's how this function gets used:

```
# fn inverse(x: f64) -> Result<f64, String> {
#     if x == 0.0f64 { return Err("x cannot be zero!".to_string()); }
#     Ok(1.0f64 / x)
# }
let x = inverse(25.0f64);

match x {
    Ok(x) => println!("The inverse of 25 is {}", x),
    Err(msg) => println!("Error: {}", msg),
}
```

The `match` enforces that we handle the `Err` case. In addition, because the answer is wrapped up in an `Ok`, we can't just use the result without doing the match:

```
let x = inverse(25.0f64);
println!("{}", x + 2.0f64); // error: binary operation `+` cannot be applied
           // to type `core::result::Result<f64,collections::string::String>`
```

This function is great, but there's one other problem: it only works for 64 bit floating point values. What if we wanted to handle 32 bit floating point as well? We'd have to write this:

```
fn inverse32(x: f32) -> Result<f32, String> {
    if x == 0.0f32 { return Err("x cannot be zero!".to_string()); }

    Ok(1.0f32 / x)
}
```

Bummer. What we need is a *generic function*. Luckily, we can write one! However, it won't *quite* work yet. Before we get into that, let's talk syntax. A generic version of `inverse` would look something like this:

```
fn inverse<T>(x: T) -> Result<T, String> {
    if x == 0.0 { return Err("x cannot be zero!".to_string()); }

    Ok(1.0 / x)
}
```

Just like how we had `Option<T>`, we use a similar syntax for `inverse<T>`. We can then use `T` inside the rest of the signature: `x` has type `T`, and half of the `Result` has type `T`. However, if we try to compile that example, we'll get an error:

```
error: binary operation `==` cannot be applied to type `T`
```

Because `T` can be *any* type, it may be a type that doesn't implement `==`, and therefore, the first line would be wrong. What do we do?

To fix this example, we need to learn about another Rust feature: traits.

2.10 Traits

Do you remember the `impl` keyword, used to call a function with method syntax?

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

Traits are similar, except that we define a trait with just the method signature, then implement the trait for that struct. Like this:

```
struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}
```

As you can see, the `trait` block looks very similar to the `impl` block, but we don't define a body, just a type signature. When we `impl` a trait, we use `impl Trait for Item`, rather than just `impl Item`.

So what's the big deal? Remember the error we were getting with our generic `inverse` function?

```
error: binary operation `==` cannot be applied to type `T`
```

We can use traits to constrain our generics. Consider this function, which does not compile, and gives us a similar error:

```
fn print_area<T>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

Rust complains:

```
error: type `T` does not implement any method in scope named `area`
```

Because `T` can be any type, we can't be sure that it implements the `area` method. But we can add a *trait constraint* to our generic `T`, ensuring that it does:

```
# trait HasArea {
#     fn area(&self) -> f64;
```

```
# }
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}
```

The syntax `<T: HasArea>` means `any type that implements the HasArea trait`. Because traits define function type signatures, we can be sure that any type which implements `HasArea` will have an `.area()` method.

Here's an extended example of how this works:

```
trait HasArea {
    fn area(&self) -> f64;
}

struct Circle {
    x: f64,
    y: f64,
    radius: f64,
}

impl HasArea for Circle {
    fn area(&self) -> f64 {
        std::f64::consts::PI * (self.radius * self.radius)
    }
}

struct Square {
    x: f64,
    y: f64,
    side: f64,
}

impl HasArea for Square {
    fn area(&self) -> f64 {
        self.side * self.side
    }
}

fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    let s = Square {
        x: 0.0f64,
        y: 0.0f64,
        side: 1.0f64,
    };

    print_area(c);
    print_area(s);
}
```

This program outputs:

```
This shape has an area of 3.141593
This shape has an area of 1
```

As you can see, `print_area` is now generic, but also ensures that we have passed in the correct types. If we pass in an incorrect type:

```
print_area(5i);
```

We get a compile-time error:

```
error: failed to find an implementation of trait main::HasArea for int
```

So far, we've only added trait implementations to structs, but you can implement a trait for any type. So technically, we *could* implement `HasArea` for `int`:

```
trait HasArea {
    fn area(&self) -> f64;
}

impl HasArea for int {
    fn area(&self) -> f64 {
        println!("this is silly");

        *self as f64
    }
}

5i.area();
```

It is considered poor style to implement methods on such primitive types, even though it is possible.

This may seem like the Wild West, but there are two other restrictions around implementing traits that prevent this from getting out of hand. First, traits must be `use`d in any scope where you wish to use the trait's method. So for example, this does not work:

```
mod shapes {
    use std::f64::consts;

    trait HasArea {
        fn area(&self) -> f64;
    }

    struct Circle {
        x: f64,
        y: f64,
        radius: f64,
    }

    impl HasArea for Circle {
        fn area(&self) -> f64 {
            consts::PI * (self.radius * self.radius)
        }
    }
}

fn main() {
    let c = shapes::Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    println!("{}", c.area());
}
```

Now that we've moved the structs and traits into their own module, we get an error:

```
error: type `shapes::Circle` does not implement any method in scope named `area`
```

If we add a `use` line right above `main` and make the right things public, everything is fine:

```
use shapes::HasArea;

mod shapes {
    use std::f64::consts;
```

```
    pub trait HasArea {
        fn area(&self) -> f64;
    }

    pub struct Circle {
        pub x: f64,
        pub y: f64,
        pub radius: f64,
    }

    impl HasArea for Circle {
        fn area(&self) -> f64 {
            consts::PI * (self.radius * self.radius)
        }
    }
}


fn main() {
    let c = shapes::Circle {
        x: 0.0f64,
        y: 0.0f64,
        radius: 1.0f64,
    };

    println!("{}", c.area());
}
```

This means that even if someone does something bad like add methods to `int`, it won't affect you, unless you `use` that trait.

There's one more restriction on implementing traits. Either the trait or the type you're writing the `impl` for must be inside your crate. So, we could implement the `HasArea` type for `int`, because `HasArea` is in our crate. But if we tried to implement `Float`, a trait provided by Rust, for `int`, we could not, because both the trait and the type aren't in our crate.

One last thing about traits: generic functions with a trait bound use *monomorphization* (*mono*: one, *morph*: form), so they are statically dispatched. What's that mean? Well, let's take a look at `print_area` again:

```
fn print_area<T: HasArea>(shape: T) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle { ... };

    let s = Square { ... };

    print_area(c);
    print_area(s);
}
```

When we use this trait with `Circle` and `Square`, Rust ends up generating two different functions with the concrete type, and replacing the call sites with calls to the concrete implementations. In other words, you get something like this:

```
fn __print_area_circle(shape: Circle) {
    println!("This shape has an area of {}", shape.area());
}

fn __print_area_square(shape: Square) {
    println!("This shape has an area of {}", shape.area());
}

fn main() {
    let c = Circle { ... };

    let s = Square { ... };
```

```
    __print_area_circle(c);
    __print_area_square(s);
}
```

The names don't actually change to this, it's just for illustration. But as you can see, there's no overhead of deciding which version to call here, hence *statically dispatched*. The downside is that we have two copies of the same function, so our binary is a little bit larger.

2.11 Tasks

NOTE This guide is badly out of date and needs to be rewritten.

2.11.1 Introduction

Rust provides safe concurrent abstractions through a number of core library primitives. This guide will describe the concurrency model in Rust, how it relates to the Rust type system, and introduce the fundamental library abstractions for constructing concurrent programs.

Threads provide failure isolation and recovery. When a fatal error occurs in Rust code as a result of an explicit call to `panic!()`, an assertion failure, or another invalid operation, the runtime system destroys the entire thread. Unlike in languages such as Java and C++, there is no way to `catch` an exception. Instead, threads may monitor each other to see if they panic.

Threads use Rust's type system to provide strong memory safety guarantees. In particular, the type system guarantees that threads cannot induce a data race from shared mutable state.

2.11.2 Basics

At its simplest, creating a thread is a matter of calling the `spawn` function with a closure argument. `spawn` executes the closure in the new thread.

```
# use std::thread::spawn;

// Print something profound in a different thread using a named function
fn print_message() { println!("I am running in a different thread!"); }
spawn(print_message);

// Alternatively, use a `move ||` expression instead of a named function.
// `||` expressions evaluate to an unnamed closure. The `move` keyword
// indicates that the closure should take ownership of any variables it
// touches.
spawn(move || println!("I am also running in a different thread!"));
```

In Rust, a thread is not a concept that appears in the language semantics. Instead, Rust's type system provides all the tools necessary to implement safe concurrency: particularly, ownership. The language leaves the implementation details to the standard library.

The `spawn` function has the type signature: `fn spawn<F:FnOnce()+Send>(f: F)`. This indicates that it takes as argument a closure (of type `F`) that it will run exactly once. This closure is limited to capturing `Send`-able data from its environment (that is, data which is deeply owned). Limiting the closure to `Send` ensures that `spawn` can safely move the entire closure and all its associated state into an entirely different thread for execution.

```
# use std::thread::spawn;
# fn generate_thread_number() -> int { 0 }
// Generate some state locally
let child_thread_number = generate_thread_number();

spawn(move || {
    // Capture it in the remote thread. The `move` keyword indicates
    // that this closure should move `child_thread_number` into its
```

```
    // environment, rather than capturing a reference into the
    // enclosing stack frame.
    println!("I am child number {}", child_thread_number);
});
```

2.11.2.1 Communication

Now that we have spawned a new thread, it would be nice if we could communicate with it. For this, we use *channels*. A channel is simply a pair of endpoints: one for sending messages and another for receiving messages.

The simplest way to create a channel is to use the `channel` function to create a `(Sender, Receiver)` pair. In Rust parlance, a *sender* is a sending endpoint of a channel, and a *receiver* is the receiving endpoint. Consider the following example of calculating two results concurrently:

```
# use std::thread::spawn;

let (tx, rx): (Sender<int>, Receiver<int>) = channel();

spawn(move || {
    let result = some_expensive_computation();
    tx.send(result);
});

some_other_expensive_computation();
let result = rx.recv();
# fn some_expensive_computation() -> int { 42 }
# fn some_other_expensive_computation() {}
```

Let's examine this example in detail. First, the `let` statement creates a stream for sending and receiving integers (the left-hand side of the `let`, `(tx, rx)`, is an example of a destructuring let: the pattern separates a tuple into its component parts).

```
let (tx, rx): (Sender<int>, Receiver<int>) = channel();
```

The child thread will use the sender to send data to the parent thread, which will wait to receive the data on the receiver. The next statement spawns the child thread.

```
# use std::thread::spawn;
# fn some_expensive_computation() -> int { 42 }
# let (tx, rx) = channel();
spawn(move || {
    let result = some_expensive_computation();
    tx.send(result);
});
```

Notice that the creation of the thread closure transfers `tx` to the child thread implicitly: the closure captures `tx` in its environment. Both `Sender` and `Receiver` are sendable types and may be captured into threads or otherwise transferred between them. In the example, the child thread runs an expensive computation, then sends the result over the captured channel.

Finally, the parent continues with some other expensive computation, then waits for the child's result to arrive on the receiver:

```
# fn some_other_expensive_computation() {}
# let (tx, rx) = channel::<int>();
# tx.send(0);
some_other_expensive_computation();
let result = rx.recv();
```

The `Sender` and `Receiver` pair created by `channel` enables efficient communication between a single sender and a single receiver, but multiple senders cannot use a single `Sender` value, and multiple receivers cannot use a single `Receiver` value. What if our example needed to compute multiple results across a number of threads? The following program is ill-typed:

```
# fn some_expensive_computation() -> int { 42 }
let (tx, rx) = channel();

spawn(move || {
    tx.send(some_expensive_computation());
});

// ERROR! The previous spawn statement already owns the sender,
// so the compiler will not allow it to be captured again
spawn(move || {
    tx.send(some_expensive_computation());
});
```

Instead we can clone the `tx`, which allows for multiple senders.

```
let (tx, rx) = channel();

for init_val in range(0u, 3) {
    // Create a new channel handle to distribute to the child thread
    let child_tx = tx.clone();
    spawn(move || {
        child_tx.send(some_expensive_computation(init_val));
    });
}

let result = rx.recv() + rx.recv() + rx.recv();
# fn some_expensive_computation(_i: uint) -> int { 42 }
```

Cloning a `Sender` produces a new handle to the same channel, allowing multiple threads to send data to a single receiver. It upgrades the channel internally in order to allow this functionality, which means that channels that are not cloned can avoid the overhead required to handle multiple senders. But this fact has no bearing on the channel's usage: the upgrade is transparent.

Note that the above cloning example is somewhat contrived since you could also simply use three `Sender` pairs, but it serves to illustrate the point. For reference, written with multiple streams, it might look like the example below.

```
# use std::thread::spawn;

// Create a vector of ports, one for each child thread
let rxs = Vec::from_fn(3, |init_val| {
    let (tx, rx) = channel();
    spawn(move || {
        tx.send(some_expensive_computation(init_val));
    });
    rx
});

// Wait on each port, accumulating the results
let result = rxs.iter().fold(0, |accum, rx| accum + rx.recv() );
# fn some_expensive_computation(_i: uint) -> int { 42 }
```

2.11.2.2 Backgrounding computations: Futures

With `sync::Future`, rust has a mechanism for requesting a computation and getting the result later.

The basic example below illustrates this.

```
# #![allow(deprecated)]
use std::sync::Future;

# fn main() {
# fn make_a_sandwich() {};
fn fib(n: u64) -> u64 {
    // lengthy computation returning an uint
    12586269025
}
```

```
let mut delayed_fib = Future::spawn(move || fib(50));
make_a_sandwich();
println!("fib(50) = {}", delayed_fib.get())
# }
```

The call to `future::spawn` immediately returns a `future` object regardless of how long it takes to run `fib(50)`. You can then make yourself a sandwich while the computation of `fib` is running. The result of the execution of the method is obtained by calling `get` on the future. This call will block until the value is available (*i.e.* the computation is complete). Note that the future needs to be mutable so that it can save the result for next time `get` is called.

Here is another example showing how futures allow you to background computations. The workload will be distributed on the available cores.

```
# #![allow(deprecated)]
# use std::num::Float;
# use std::sync::Future;
fn partial_sum(start: uint) -> f64 {
    let mut local_sum = 0f64;
    for num in range(start*100000, (start+1)*100000) {
        local_sum += (num as f64 + 1.0).powf(-2.0);
    }
    local_sum
}

fn main() {
    let mut futures = Vec::from_fn(200, |ind| Future::spawn(move || partial_sum(ind)));

    let mut final_res = 0f64;
    for ft in futures.iter_mut()  {
        final_res += ft.get();
    }
    println!("π^2/6 is not far from : {}", final_res);
}
```

2.11.2.3 Sharing without copying: Arc

To share data between threads, a first approach would be to only use channel as we have seen previously. A copy of the data to share would then be made for each thread. In some cases, this would add up to a significant amount of wasted memory and would require copying the same data more than necessary.

To tackle this issue, one can use an Atomically Reference Counted wrapper (`Arc`) as implemented in the `sync` library of Rust. With an Arc, the data will no longer be copied for each thread. The Arc acts as a reference to the shared data and only this reference is shared and cloned.

Here is a small example showing how to use Arcs. We wish to run concurrently several computations on a single large vector of floats. Each thread needs the full vector to perform its duty.

```
use std::num::Float;
use std::rand;
use std::sync::Arc;

fn pnorm(nums: &[f64], p: uint) -> f64 {
    nums.iter().fold(0.0, |a, b| a + b.powf(p as f64)).powf(1.0 / (p as f64))
}

fn main() {
    let numbers = Vec::from_fn(1000000, |_| rand::random::<f64>());
    let numbers_arc = Arc::new(numbers);

    for num in range(1u, 10) {
        let thread_numbers = numbers_arc.clone();

        spawn(move || {
```

```
            println!("{}-norm = {}", num, pnorm(thread_numbers.as_slice(), num));
        });
    }
}
```

The function `pnorm` performs a simple computation on the vector (it computes the sum of its items at the power given as argument and takes the inverse power of this value). The Arc on the vector is created by the line:

```
# use std::rand;
# use std::sync::Arc;
# fn main() {
# let numbers = Vec::from_fn(1000000, |_| rand::random::<f64>());
let numbers_arc = Arc::new(numbers);
# }
```

and a clone is captured for each thread via a procedure. This only copies the wrapper and not its contents. Within the thread's procedure, the captured Arc reference can be used as a shared reference to the underlying vector as if it were local.

```
# use std::rand;
# use std::sync::Arc;
# fn pnorm(nums: &[f64], p: uint) -> f64 { 4.0 }
# fn main() {
# let numbers=Vec::from_fn(1000000, |_| rand::random::<f64>());
# let numbers_arc = Arc::new(numbers);
# let num = 4;
let thread_numbers = numbers_arc.clone();
spawn(move || {
    // Capture thread_numbers and use it as if it was the underlying vector
    println!("{}-norm = {}", num, pnorm(thread_numbers.as_slice(), num));
});
# }
```

2.11.3 Handling thread panics

Rust has a built-in mechanism for raising exceptions. The `panic!()` macro (which can also be written with an error string as an argument: `panic!( ~reason)`) and the `assert!` construct (which effectively calls `panic!()` if a boolean expression is false) are both ways to raise exceptions. When a thread raises an exception, the thread unwinds its stack—running destructors and freeing memory along the way—and then exits. Unlike exceptions in C++, exceptions in Rust are unrecoverable within a single thread: once a thread panics, there is no way to "catch" the exception.

While it isn't possible for a thread to recover from panicking, threads may notify each other if they panic. The simplest way of handling a panic is with the `try` function, which is similar to `spawn`, but immediately blocks and waits for the child thread to finish. `try` returns a value of type `Result<T, Box<Any + Send>>`. `Result` is an `enum` type with two variants: `Ok` and `Err`. In this case, because the type arguments to `Result` are `int` and `()`, callers can pattern-match on a result to check whether it's an `Ok` result with an `int` field (representing a successful result) or an `Err` result (representing termination with an error).

```
# use std::thread::Thread;
# fn some_condition() -> bool { false }
# fn calculate_result() -> int { 0 }
let result: Result<int, Box<std::any::Any + Send>> = Thread::spawn(move || {
    if some_condition() {
        calculate_result()
    } else {
        panic!("oops!");
    }
}).join();
assert!(result.is_err());
```

Unlike `spawn`, the function spawned using `try` may return a value, which `try` will dutifully propagate back to the caller in a `Result` enum. If the child thread terminates successfully, `try` will return an `Ok` result; if the child thread panics, `try` will return an `Error` result.

Note: A panicked thread does not currently produce a useful error value (`try` always returns `Err(())`). In the future, it may be possible for threads to intercept the value passed to `panic!()`.

But not all panics are created equal. In some cases you might need to abort the entire program (perhaps you're writing an assert which, if it trips, indicates an unrecoverable logic error); in other cases you might want to contain the panic at a certain boundary (perhaps a small piece of input from the outside world, which you happen to be processing in parallel, is malformed such that the processing thread cannot proceed).

2.12 Error Handling

> The best-laid plans of mice and men Often go awry
>
> "Tae a Moose", Robert Burns

Sometimes, things just go wrong. It's important to have a plan for when the inevitable happens. Rust has rich support for handling errors that may (let's be honest: will) occur in your programs.

There are two main kinds of errors that can occur in your programs: failures, and panics. Let's talk about the difference between the two, and then discuss how to handle each. Then, we'll discuss upgrading failures to panics.

2.12.1 Failure vs. Panic

Rust uses two terms to differentiate between two forms of error: failure, and panic. A *failure* is an error that can be recovered from in some way. A *panic* is an error that cannot be recovered from.

What do we mean by "recover"? Well, in most cases, the possibility of an error is expected. For example, consider the `from_str` function:

```
from_str("5");
```

This function takes a string argument and converts it into another type. But because it's a string, you can't be sure that the conversion actually works. For example, what should this convert to?

```
from_str("hello5world");
```

This won't work. So we know that this function will only work properly for some inputs. It's expected behavior. We call this kind of error a *failure*.

On the other hand, sometimes, there are errors that are unexpected, or which we cannot recover from. A classic example is an `assert!`:

```
assert!(x == 5);
```

We use `assert!` to declare that something is true. If it's not true, something is very wrong. Wrong enough that we can't continue with things in the current state. Another example is using the `unreachable!()` macro:

```
enum Event {
    NewRelease,
}

fn probability(_: &Event) -> f64 {
    // real implementation would be more complex, of course
    0.95
}

fn descriptive_probability(event: Event) -> &'static str {
    match probability(&event) {
        1.00          => "certain",
        0.00          => "impossible",
        0.00 ... 0.25 => "very unlikely",
        0.25 ... 0.50 => "unlikely",
```

```
        0.50 ... 0.75 => "likely",
        0.75 ... 1.00  => "very likely",
    }
}

fn main() {
    std::io::println(descriptive_probability(NewRelease));
}
```

This will give us an error:

```
error: non-exhaustive patterns: `_` not covered [E0004]
```

While we know that we've covered all possible cases, Rust can't tell. It doesn't know that probability is between 0.0 and 1.0. So we add another case:

```
use Event::NewRelease;

enum Event {
    NewRelease,
}

fn probability(_: &Event) -> f64 {
    // real implementation would be more complex, of course
    0.95
}

fn descriptive_probability(event: Event) -> &'static str {
    match probability(&event) {
        1.00          => "certain",
        0.00          => "impossible",
        0.00 ... 0.25 => "very unlikely",
        0.25 ... 0.50 => "unlikely",
        0.50 ... 0.75 => "likely",
        0.75 ... 1.00  => "very likely",
        _ => unreachable!()
    }
}

fn main() {
    println!("{}", descriptive_probability(NewRelease));
}
```

We shouldn't ever hit the _ case, so we use the `unreachable!()` macro to indicate this. `unreachable!()` gives a different kind of error than `Result`. Rust calls these sorts of errors *panics*.

2.12.2 Handling errors with `Option` and `Result`

The simplest way to indicate that a function may fail is to use the `Option<T>` type. Remember our `from_str()` example? Here's its type signature:

```
pub fn from_str<A: FromStr>(s: &str) -> Option<A>
```

`from_str()` returns an `Option<A>`. If the conversion succeeds, it will return `Some(value)`, and if it fails, it will return `None`.

This is appropriate for the simplest of cases, but doesn't give us a lot of information in the failure case. What if we wanted to know *why* the conversion failed? For this, we can use the `Result<T, E>` type. It looks like this:

```
enum Result<T, E> {
   Ok(T),
   Err(E)
}
```

This enum is provided by Rust itself, so you don't need to define it to use it in your code. The `Ok(T)` variant represents a success, and the `Err(E)` variant represents a failure. Returning a `Result` instead of an `Option` is

recommended for all but the most trivial of situations.

Here's an example of using `Result`:

```
#[derive(Show)]
enum Version { Version1, Version2 }

#[derive(Show)]
enum ParseError { InvalidHeaderLength, InvalidVersion }

fn parse_version(header: &[u8]) -> Result<Version, ParseError> {
    if header.len() < 1 {
        return Err(ParseError::InvalidHeaderLength);
    }
    match header[0] {
        1 => Ok(Version::Version1),
        2 => Ok(Version::Version2),
        _ => Err(ParseError::InvalidVersion)
    }
}

let version = parse_version(&[1, 2, 3, 4]);
match version {
    Ok(v) => {
        println!("working with version: {:?}", v);
    }
    Err(e) => {
        println!("error parsing header: {:?}", e);
    }
}
```

This function makes use of an enum, `ParseError`, to enumerate the various errors that can occur.

2.12.3 Non-recoverable errors with `panic!`

In the case of an error that is unexpected and not recoverable, the `panic!` macro will induce a panic. This will crash the current task, and give an error:

```
panic!("boom");
```

gives

```
task '<main>' panicked at 'boom', hello.rs:2
```

when you run it.

Because these kinds of situations are relatively rare, use panics sparingly.

2.12.4 Upgrading failures to panics

In certain circumstances, even though a function may fail, we may want to treat it as a panic instead. For example, `io::stdin().read_line()` returns an `IoResult<String>`, a form of `Result`, when there is an error reading the line. This allows us to handle and possibly recover from this sort of error.

If we don't want to handle this error, and would rather just abort the program, we can use the `unwrap()` method:

```
io::stdin().read_line().unwrap();
```

`unwrap()` will `panic!` if the `Option` is `None`. This basically says "Give me the value, and if something goes wrong, just crash." This is less reliable than matching the error and attempting to recover, but is also significantly shorter. Sometimes, just crashing is appropriate.

There's another way of doing this that's a bit nicer than `unwrap()`:

```
let input = io::stdin().read_line()
                       .ok()
                       .expect("Failed to read line");
```

`ok()` converts the `IoResult` into an `Option`, and `expect()` does the same thing as `unwrap()`, but takes a message. This message is passed along to the underlying `panic!`, providing a better error message if the code errors.

3 Advanced Topics

In a similar fashion to “Intermediate,” this section is full of individual, deep-dive chapters, which stand alone and can be read in any order. These chapters focus on the most complex features, as well as some things that are only available in upcoming versions of Rust.

After reading “Advanced,” you’ll be a Rust expert!

3.1 FFI

3.1.1 Introduction

This guide will use the snappy compression/decompression library as an introduction to writing bindings for foreign code. Rust is currently unable to call directly into a C++ library, but snappy includes a C interface (documented in `snappy-c.h`).

The following is a minimal example of calling a foreign function which will compile if snappy is installed:

```
extern crate libc;
use libc::size_t;

#[link(name = "snappy")]
extern {
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
}

fn main() {
    let x = unsafe { snappy_max_compressed_length(100) };
    println!("max compressed length of a 100 byte buffer: {}", x);
}
```

The `extern` block is a list of function signatures in a foreign library, in this case with the platform’s C ABI. The `#[link(...)]` attribute is used to instruct the linker to link against the snappy library so the symbols are resolved.

Foreign functions are assumed to be unsafe so calls to them need to be wrapped with `unsafe {}` as a promise to the compiler that everything contained within truly is safe. C libraries often expose interfaces that aren’t thread-safe, and almost any function that takes a pointer argument isn’t valid for all possible inputs since the pointer could be dangling, and raw pointers fall outside of Rust’s safe memory model.

When declaring the argument types to a foreign function, the Rust compiler can not check if the declaration is correct, so specifying it correctly is part of keeping the binding correct at runtime.

The `extern` block can be extended to cover the entire snappy API:

```
extern crate libc;
use libc::{c_int, size_t};
```

```
#[link(name = "snappy")]
extern {
    fn snappy_compress(input: *const u8,
                       input_length: size_t,
                       compressed: *mut u8,
                       compressed_length: *mut size_t) -> c_int;
    fn snappy_uncompress(compressed: *const u8,
                         compressed_length: size_t,
                         uncompressed: *mut u8,
                         uncompressed_length: *mut size_t) -> c_int;
    fn snappy_max_compressed_length(source_length: size_t) -> size_t;
    fn snappy_uncompressed_length(compressed: *const u8,
                                  compressed_length: size_t,
                                  result: *mut size_t) -> c_int;
    fn snappy_validate_compressed_buffer(compressed: *const u8,
                                         compressed_length: size_t) -> c_int;
}
# fn main() {}
```

3.1.2 Creating a safe interface

The raw C API needs to be wrapped to provide memory safety and make use of higher-level concepts like vectors. A library can choose to expose only the safe, high-level interface and hide the unsafe internal details.

Wrapping the functions which expect buffers involves using the `slice::raw` module to manipulate Rust vectors as pointers to memory. Rust's vectors are guaranteed to be a contiguous block of memory. The length is number of elements currently contained, and the capacity is the total size in elements of the allocated memory. The length is less than or equal to the capacity.

```
# extern crate libc;
# use libc::{c_int, size_t};
# unsafe fn snappy_validate_compressed_buffer(_: *const u8, _: size_t) -> c_int { 0 }
# fn main() {}
pub fn validate_compressed_buffer(src: &[u8]) -> bool {
    unsafe {
        snappy_validate_compressed_buffer(src.as_ptr(), src.len() as size_t) == 0
    }
}
```

The `validate_compressed_buffer` wrapper above makes use of an `unsafe` block, but it makes the guarantee that calling it is safe for all inputs by leaving off `unsafe` from the function signature.

The `snappy_compress` and `snappy_uncompress` functions are more complex, since a buffer has to be allocated to hold the output too.

The `snappy_max_compressed_length` function can be used to allocate a vector with the maximum required capacity to hold the compressed output. The vector can then be passed to the `snappy_compress` function as an output parameter. An output parameter is also passed to retrieve the true length after compression for setting the length.

```
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_compress(a: *const u8, b: size_t, c: *mut u8,
#                           d: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_max_compressed_length(a: size_t) -> size_t { a }
# fn main() {}
```

```
pub fn compress(src: &[u8]) -> Vec<u8> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen = snappy_max_compressed_length(srclen);
        let mut dst = Vec::with_capacity(dstlen as uint);
        let pdst = dst.as_mut_ptr();

        snappy_compress(psrc, srclen, pdst, &mut dstlen);
        dst.set_len(dstlen as uint);
        dst
    }
}
```

Decompression is similar, because snappy stores the uncompressed size as part of the compression format and `snappy_uncompressed_length` will retrieve the exact buffer size required.

```
# extern crate libc;
# use libc::{size_t, c_int};
# unsafe fn snappy_uncompress(compressed: *const u8,
#                             compressed_length: size_t,
#                             uncompressed: *mut u8,
#                             uncompressed_length: *mut size_t) -> c_int { 0 }
# unsafe fn snappy_uncompressed_length(compressed: *const u8,
#                                      compressed_length: size_t,
#                                      result: *mut size_t) -> c_int { 0 }
# fn main() {}
pub fn uncompress(src: &[u8]) -> Option<Vec<u8>> {
    unsafe {
        let srclen = src.len() as size_t;
        let psrc = src.as_ptr();

        let mut dstlen: size_t = 0;
        snappy_uncompressed_length(psrc, srclen, &mut dstlen);

        let mut dst = Vec::with_capacity(dstlen as uint);
        let pdst = dst.as_mut_ptr();

        if snappy_uncompress(psrc, srclen, pdst, &mut dstlen) == 0 {
            dst.set_len(dstlen as uint);
            Some(dst)
        } else {
            None // SNAPPY_INVALID_INPUT
        }
    }
}
```

For reference, the examples used here are also available as an library on GitHub.

3.1.3 Stack management

Rust tasks by default run on a *large stack*. This is actually implemented as a reserving a large segment of the address space and then lazily mapping in pages as they are needed. When calling an external C function, the code is invoked on the same stack as the rust stack. This means that there is no extra stack-switching mechanism in place because it is assumed that the large stack for the rust task is plenty for the C function to have.

A planned future improvement (not yet implemented at the time of this writing) is to have a guard page at the end of every rust stack. No rust function will hit this guard page (due to Rust's usage of LLVM's `__morestack`). The intention for this unmapped page is to prevent infinite recursion in C from overflowing onto other rust stacks. If the guard page is hit, then the process will be terminated with a message saying that the guard page was hit.

For normal external function usage, this all means that there shouldn't be any need for any extra effort on a user's perspective. The C stack naturally interleaves with the rust stack, and it's "large enough" for both to interoperate. If, however, it is determined that a larger stack is necessary, there are appropriate functions in the task spawning API to control the size of the stack of the task which is spawned.

3.1.4 Destructors

Foreign libraries often hand off ownership of resources to the calling code. When this occurs, we must use Rust's destructors to provide safety and guarantee the release of these resources (especially in the case of panic).

3.1.5 Callbacks from C code to Rust functions

Some external libraries require the usage of callbacks to report back their current state or intermediate data to the caller. It is possible to pass functions defined in Rust to an external library. The requirement for this is that the callback function is marked as `extern` with the correct calling convention to make it callable from C code.

The callback function can then be sent through a registration call to the C library and afterwards be invoked from there.

A basic example is:

Rust code:

```
extern fn callback(a: i32) {
    println!("I'm called from C with value {0}", a);
}

#[link(name = "extlib")]
extern {
   fn register_callback(cb: extern fn(i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    unsafe {
        register_callback(callback);
        trigger_callback(); // Triggers the callback
    }
}
```

C code:

```
typedef void (*rust_callback)(int32_t);
rust_callback cb;
```

```
int32_t register_callback(rust_callback callback) {
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(7); // Will call callback(7) in Rust
}
```

In this example Rust's `main()` will call `trigger_callback()` in C, which would, in turn, call back to `callback()` in Rust.

3.1.5.1 Targeting callbacks to Rust objects

The former example showed how a global function can be called from C code. However it is often desired that the callback is targeted to a special Rust object. This could be the object that represents the wrapper for the respective C object.

This can be achieved by passing an unsafe pointer to the object down to the C library. The C library can then include the pointer to the Rust object in the notification. This will allow the callback to unsafely access the referenced Rust object.

Rust code:

```
# use std::boxed::Box;

#[repr(C)]
struct RustObject {
    a: i32,
    // other members
}

extern "C" fn callback(target: *mut RustObject, a: i32) {
    println!("I'm called from C with value {0}", a);
    unsafe {
        // Update the value in RustObject with the value received from the callback
        (*target).a = a;
    }
}

#[link(name = "extlib")]
extern {
   fn register_callback(target: *mut RustObject,
                        cb: extern fn(*mut RustObject, i32)) -> i32;
   fn trigger_callback();
}

fn main() {
    // Create the object that will be referenced in the callback
    let mut rust_object = Box::new(RustObject { a: 5 });

    unsafe {
        register_callback(&mut *rust_object, callback);
        trigger_callback();
    }
}
```

C code:

```
typedef void (*rust_callback)(void*, int32_t);
void* cb_target;
rust_callback cb;

int32_t register_callback(void* callback_target, rust_callback callback) {
    cb_target = callback_target;
    cb = callback;
    return 1;
}

void trigger_callback() {
  cb(cb_target, 7); // Will call callback(&rustObject, 7) in Rust
}
```

3.1.5.2 Asynchronous callbacks

In the previously given examples the callbacks are invoked as a direct reaction to a function call to the external C library. The control over the current thread is switched from Rust to C to Rust for the execution of the callback, but in the end the callback is executed on the same thread (and Rust task) that lead called the function which triggered the callback.

Things get more complicated when the external library spawns its own threads and invokes callbacks from there. In these cases access to Rust data structures inside the callbacks is especially unsafe and proper synchronization mechanisms must be used. Besides classical synchronization mechanisms like mutexes, one possibility in Rust is to use channels (in `std::comm`) to forward data from the C thread that invoked the callback into a Rust task.

If an asynchronous callback targets a special object in the Rust address space it is also absolutely necessary that no more callbacks are performed by the C library after the respective Rust object gets destroyed. This can be achieved by unregistering the callback in the object's destructor and designing the library in a way that guarantees that no callback will be performed after deregistration.

3.1.6 Linking

The `link` attribute on `extern` blocks provides the basic building block for instructing rustc how it will link to native libraries. There are two accepted forms of the link attribute today:

- `#[link(name = "foo")]`
- `#[link(name = "foo", kind = "bar")]`

In both of these cases, `foo` is the name of the native library that we're linking to, and in the second case `bar` is the type of native library that the compiler is linking to. There are currently three known types of native libraries:

- Dynamic - `#[link(name = "readline")]`
- Static - `#[link(name = "my_build_dependency", kind = "static")]`
- Frameworks - `#[link(name = "CoreFoundation", kind = "framework")]`

Note that frameworks are only available on OSX targets.

The different `kind` values are meant to differentiate how the native library participates in linkage. From a linkage perspective, the rust compiler creates two flavors of artifacts: partial (rlib/staticlib)

and final (dylib/binary). Native dynamic libraries and frameworks are propagated to the final artifact boundary, while static libraries are not propagated at all.

A few examples of how this model can be used are:

- A native build dependency. Sometimes some C/C++ glue is needed when writing some rust code, but distribution of the C/C++ code in a library format is just a burden. In this case, the code will be archived into `libfoo.a` and then the rust crate would declare a dependency via `#[link(name = "foo", kind = "static")]`.

Regardless of the flavor of output for the crate, the native static library will be included in the output, meaning that distribution of the native static library is not necessary.

- A normal dynamic dependency. Common system libraries (like `readline`) are available on a large number of systems, and often a static copy of these libraries cannot be found. When this dependency is included in a rust crate, partial targets (like rlibs) will not link to the library, but when the rlib is included in a final target (like a binary), the native library will be linked in.

On OSX, frameworks behave with the same semantics as a dynamic library.

3.1.6.1 The `link_args` attribute

There is one other way to tell rustc how to customize linking, and that is via the `link_args` attribute. This attribute is applied to `extern` blocks and specifies raw flags which need to get passed to the linker when producing an artifact. An example usage would be:

```
#![feature(link_args)]

#[link_args = "-foo -bar -baz"]
extern {}
# fn main() {}
```

Note that this feature is currently hidden behind the `feature(link_args)` gate because this is not a sanctioned way of performing linking. Right now rustc shells out to the system linker, so it makes sense to provide extra command line arguments, but this will not always be the case. In the future rustc may use LLVM directly to link native libraries in which case `link_args` will have no meaning.

It is highly recommended to *not* use this attribute, and rather use the more formal `#[link(...)]` attribute on `extern` blocks instead.

3.1.7 Unsafe blocks

Some operations, like dereferencing unsafe pointers or calling functions that have been marked unsafe are only allowed inside unsafe blocks. Unsafe blocks isolate unsafety and are a promise to the compiler that the unsafety does not leak out of the block.

Unsafe functions, on the other hand, advertise it to the world. An unsafe function is written like this:

```
unsafe fn kaboom(ptr: *const int) -> int { *ptr }
```

This function can only be called from an `unsafe` block or another `unsafe` function.

3.1.8 Accessing foreign globals

Foreign APIs often export a global variable which could do something like track global state. In order to access these variables, you declare them in `extern` blocks with the `static` keyword:

```
extern crate libc;

#[link(name = "readline")]
extern {
    static rl_readline_version: libc::c_int;
}

fn main() {
    println!("You have readline version {} installed.",
             rl_readline_version as int);
}
```

Alternatively, you may need to alter global state provided by a foreign interface. To do this, statics can be declared with `mut` so rust can mutate them.

```
extern crate libc;

use std::ffi::CString;
use std::ptr;

#[link(name = "readline")]
extern {
    static mut rl_prompt: *const libc::c_char;
}

fn main() {
    let prompt = CString::from_slice(b"[my-awesome-shell] $");
    unsafe { rl_prompt = prompt.as_ptr(); }
    // get a line, process it
    unsafe { rl_prompt = ptr::null(); }
}
```

3.1.9 Foreign calling conventions

Most foreign code exposes a C ABI, and Rust uses the platform's C calling convention by default when calling foreign functions. Some foreign functions, most notably the Windows API, use other calling conventions. Rust provides a way to tell the compiler which convention to use:

```
extern crate libc;

#[cfg(all(target_os = "win32", target_arch = "x86"))]
#[link(name = "kernel32")]
#[allow(non_snake_case)]
extern "stdcall" {
    fn SetEnvironmentVariableA(n: *const u8, v: *const u8) -> libc::c_int;
}
# fn main() { }
```

This applies to the entire `extern` block. The list of supported ABI constraints are:

- `stdcall`
- `aapcs`

- `cdecl`
- `fastcall`
- `Rust`
- `rust-intrinsic`
- `system`
- `C`
- `win64`

Most of the abis in this list are self-explanatory, but the `system` abi may seem a little odd. This constraint selects whatever the appropriate ABI is for interoperating with the target's libraries. For example, on win32 with a x86 architecture, this means that the abi used would be `stdcall`. On x86_64, however, windows uses the `C` calling convention, so `C` would be used. This means that in our previous example, we could have used `extern "system" { ... }` to define a block for all windows systems, not just x86 ones.

3.1.10 Interoperability with foreign code

Rust guarantees that the layout of a `struct` is compatible with the platform's representation in C only if the `#[repr(C)]` attribute is applied to it. `#[repr(C, packed)]` can be used to lay out struct members without padding. `#[repr(C)]` can also be applied to an enum.

Rust's owned boxes (`Box<T>`) use non-nullable pointers as handles which point to the contained object. However, they should not be manually created because they are managed by internal allocators. References can safely be assumed to be non-nullable pointers directly to the type. However, breaking the borrow checking or mutability rules is not guaranteed to be safe, so prefer using raw pointers (`*`) if that's needed because the compiler can't make as many assumptions about them.

Vectors and strings share the same basic memory layout, and utilities are available in the `vec` and `str` modules for working with C APIs. However, strings are not terminated with `\0`. If you need a NUL-terminated string for interoperability with C, you should use the `CString` type in the `std::ffi` module.

The standard library includes type aliases and function definitions for the C standard library in the `libc` module, and Rust links against `libc` and `libm` by default.

3.1.11 The "nullable pointer optimization"

Certain types are defined to not be `null`. This includes references (`&T`, `&mut T`), boxes (`Box<T>`), and function pointers (`extern "abi" fn()`). When interfacing with C, pointers that might be null are often used. As a special case, a generic `enum` that contains exactly two variants, one of which contains no data and the other containing a single field, is eligible for the "nullable pointer optimization". When such an enum is instantiated with one of the non-nullable types, it is represented as a single pointer, and the non-data variant is represented as the null pointer. So `Option<extern "C" fn(c_int) -> c_int>` is how one represents a nullable function pointer using the C ABI.

3.2 Unsafe Code

3.2.1 Introduction

Rust aims to provide safe abstractions over the low-level details of the CPU and operating system, but sometimes one needs to drop down and write code at that level. This guide aims to provide an overview of the dangers and power one gets with Rust's unsafe subset.

Rust provides an escape hatch in the form of the `unsafe { ... }` block which allows the programmer to dodge some of the compiler's checks and do a wide range of operations, such as:

- dereferencing raw pointers
- calling a function via FFI (covered by the FFI guide)
- casting between types bitwise (`transmute`, aka "reinterpret cast")
- inline assembly

Note that an `unsafe` block does not relax the rules about lifetimes of `&` and the freezing of borrowed data.

Any use of `unsafe` is the programmer saying "I know more than you" to the compiler, and, as such, the programmer should be very sure that they actually do know more about why that piece of code is valid. In general, one should try to minimize the amount of unsafe code in a code base; preferably by using the bare minimum `unsafe` blocks to build safe interfaces.

> **Note**: the low-level details of the Rust language are still in flux, and there is no guarantee of stability or backwards compatibility. In particular, there may be changes that do not cause compilation errors, but do cause semantic changes (such as invoking undefined behaviour). As such, extreme care is required.

3.2.2 Pointers

3.2.2.1 References

One of Rust's biggest features is memory safety. This is achieved in part via the ownership system, which is how the compiler can guarantee that every `&` reference is always valid, and, for example, never pointing to freed memory.

These restrictions on `&` have huge advantages. However, they also constrain how we can use them. For example, `&` doesn't behave identically to C's pointers, and so cannot be used for pointers in foreign function interfaces (FFI). Additionally, both immutable (`&`) and mutable (`&mut`) references have some aliasing and freezing guarantees, required for memory safety.

In particular, if you have an `&T` reference, then the `T` must not be modified through that reference or any other reference. There are some standard library types, e.g. `Cell` and `RefCell`, that provide inner mutability by replacing compile time guarantees with dynamic checks at runtime.

An `&mut` reference has a different constraint: when an object has an `&mut T` pointing into it, then that `&mut` reference must be the only such usable path to that object in the whole program. That is, an `&mut` cannot alias with any other references.

Using `unsafe` code to incorrectly circumvent and violate these restrictions is undefined behaviour. For example, the following creates two aliasing `&mut` pointers, and is invalid.

```
use std::mem;
let mut x: u8 = 1;

let ref_1: &mut u8 = &mut x;
let ref_2: &mut u8 = unsafe { mem::transmute(&mut *ref_1) };

// oops, ref_1 and ref_2 point to the same piece of data (x) and are
// both usable
*ref_1 = 10;
*ref_2 = 20;
```

3.2.2.2 Raw pointers

Rust offers two additional pointer types (*raw pointers*), written as `*const T` and `*mut T`. They're an approximation of C's `const T*` and `T*` respectively; indeed, one of their most common uses is for FFI, interfacing with external C libraries.

Raw pointers have much fewer guarantees than other pointer types offered by the Rust language and libraries. For example, they

- are not guaranteed to point to valid memory and are not even guaranteed to be non-null (unlike both `Box` and `&`);
- do not have any automatic clean-up, unlike `Box`, and so require manual resource management;
- are plain-old-data, that is, they don't move ownership, again unlike `Box`, hence the Rust compiler cannot protect against bugs like use-after-free;
- are considered sendable (if their contents is considered sendable), so the compiler offers no assistance with ensuring their use is thread-safe; for example, one can concurrently access a `*mut int` from two threads without synchronization.
- lack any form of lifetimes, unlike `&`, and so the compiler cannot reason about dangling pointers; and
- have no guarantees about aliasing or mutability other than mutation not being allowed directly through a `*const T`.

Fortunately, they come with a redeeming feature: the weaker guarantees mean weaker restrictions. The missing restrictions make raw pointers appropriate as a building block for implementing things like smart pointers and vectors inside libraries. For example, `*` pointers are allowed to alias, allowing them to be used to write shared-ownership types like reference counted and garbage collected pointers, and even thread-safe shared memory types (`Rc` and the `Arc` types are both implemented entirely in Rust).

There are two things that you are required to be careful about (i.e. require an `unsafe { ... }` block) with raw pointers:

- dereferencing: they can have any value: so possible results include a crash, a read of uninitialised memory, a use-after-free, or reading data as normal.
- pointer arithmetic via the `offset` intrinsic (or `.offset` method): this intrinsic uses so-called "in-bounds" arithmetic, that is, it is only defined behaviour if the result is inside (or one-byte-past-the-end) of the object from which the original pointer came.

The latter assumption allows the compiler to optimize more effectively. As can be seen, actually *creating* a raw pointer is not unsafe, and neither is converting to an integer.

3.2.2.2.1 References and raw pointers

At runtime, a raw pointer `*` and a reference pointing to the same piece of data have an identical representation. In fact, an `&T` reference will implicitly coerce to an `*const T` raw pointer in safe code and similarly for the `mut` variants (both coercions can be performed explicitly with, respectively, `value as *const T` and `value as *mut T`).

Going the opposite direction, from `*const` to a reference `&`, is not safe. A `&T` is always valid, and so, at a minimum, the raw pointer `*const T` has to point to a valid instance of type `T`. Furthermore, the resulting pointer must satisfy the aliasing and mutability laws of references. The compiler assumes these properties are true for any references, no matter how they are created, and so any conversion from raw pointers is asserting that they hold. The programmer *must* guarantee this.

The recommended method for the conversion is

```
let i: u32 = 1;
// explicit cast
let p_imm: *const u32 = &i as *const u32;
let mut m: u32 = 2;
// implicit coercion
let p_mut: *mut u32 = &mut m;

unsafe {
    let ref_imm: &u32 = &*p_imm;
    let ref_mut: &mut u32 = &mut *p_mut;
}
```

The `&*x` dereferencing style is preferred to using a `transmute`. The latter is far more powerful than necessary, and the more restricted operation is harder to use incorrectly; for example, it requires that `x` is a pointer (unlike `transmute`).

3.2.2.3 Making the unsafe safe(r)

There are various ways to expose a safe interface around some unsafe code:

- store pointers privately (i.e. not in public fields of public structs), so that you can see and control all reads and writes to the pointer in one place.
- use `assert!()` a lot: since you can't rely on the protection of the compiler & type-system to ensure that your `unsafe` code is correct at compile-time, use `assert!()` to verify that it is doing the right thing at run-time.
- implement the `Drop` for resource clean-up via a destructor, and use RAII (Resource Acquisition Is Initialization). This reduces the need for any manual memory management by users, and automatically ensures that clean-up is always run, even when the task panics.
- ensure that any data stored behind a raw pointer is destroyed at the appropriate time.

As an example, we give a reimplementation of owned boxes by wrapping `malloc` and `free`. Rust's move semantics and lifetimes mean this reimplementation is as safe as the `Box` type.

```
#![feature(unsafe_destructor)]

extern crate libc;
use libc::{c_void, size_t, malloc, free};
use std::mem;
```

```
use std::ptr;
# use std::boxed::Box;

// Define a wrapper around the handle returned by the foreign code.
// Unique<T> has the same semantics as Box<T>
pub struct Unique<T> {
    // It contains a single raw, mutable pointer to the object in question.
    ptr: *mut T
}

// Implement methods for creating and using the values in the box.

// NB: For simplicity and correctness, we require that T has kind Send
// (owned boxes relax this restriction).
impl<T: Send> Unique<T> {
    pub fn new(value: T) -> Unique<T> {
        unsafe {
            let ptr = malloc(mem::size_of::<T>() as size_t) as *mut T;
            // we *need* valid pointer.
            assert!(!ptr.is_null());
            // `*ptr` is uninitialized, and `*ptr = value` would
            // attempt to destroy it `overwrite` moves a value into
            // this memory without attempting to drop the original
            // value.
            ptr::write(&mut *ptr, value);
            Unique{ptr: ptr}
        }
    }

    // the 'r lifetime results in the same semantics as `&*x` with
    // Box<T>
    pub fn borrow<'r>(&'r self) -> &'r T {
        // By construction, self.ptr is valid
        unsafe { &*self.ptr }
    }

    // the 'r lifetime results in the same semantics as `&mut *x` with
    // Box<T>
    pub fn borrow_mut<'r>(&'r mut self) -> &'r mut T {
        unsafe { &mut *self.ptr }
    }
}

// A key ingredient for safety, we associate a destructor with
// Unique<T>, making the struct manage the raw pointer: when the
// struct goes out of scope, it will automatically free the raw pointer.
//
// NB: This is an unsafe destructor, because rustc will not normally
// allow destructors to be associated with parameterized types, due to
// bad interaction with managed boxes. (With the Send restriction,
// we don't have this problem.) Note that the `#[unsafe_destructor]`
// feature gate is required to use unsafe destructors.
#[unsafe_destructor]
impl<T: Send> Drop for Unique<T> {
    fn drop(&mut self) {
        unsafe {
            // Copy the object out from the pointer onto the stack,
            // where it is covered by normal Rust destructor semantics
            // and cleans itself up, if necessary
            ptr::read(self.ptr as *const T);

            // clean-up our allocation
            free(self.ptr as *mut c_void)
```

```
        }
    }
}

// A comparison between the built-in `Box` and this reimplementation
fn main() {
    {
        let mut x = Box::new(5i);
        *x = 10;
    } // `x` is freed here

    {
        let mut y = Unique::new(5i);
        *y.borrow_mut() = 10;
    } // `y` is freed here
}
```

Notably, the only way to construct a `Unique` is via the `new` function, and this function ensures that the internal pointer is valid and hidden in the private field. The two `borrow` methods are safe because the compiler statically guarantees that objects are never used before creation or after destruction (unless you use some `unsafe` code…).

3.2.3 Inline assembly

For extremely low-level manipulations and performance reasons, one might wish to control the CPU directly. Rust supports using inline assembly to do this via the `asm!` macro. The syntax roughly matches that of GCC & Clang:

```
asm!(assembly template
   : output operands
   : input operands
   : clobbers
   : options
   );
```

Any use of `asm` is feature gated (requires `#![feature(asm)]` on the crate to allow) and of course requires an `unsafe` block.

Note: the examples here are given in x86/x86-64 assembly, but all platforms are supported.

3.2.3.1 Assembly template

The `assembly template` is the only required parameter and must be a literal string (i.e `""`)

```
#![feature(asm)]

#[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn foo() {
    unsafe {
        asm!("NOP");
    }
}

// other platforms
#[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
fn foo() { /* ... */ }
```

```
fn main() {
    // ...
    foo();
    // ...
}
```

(The `feature(asm)` and `#[cfg]`s are omitted from now on.)

Output operands, input operands, clobbers and options are all optional but you must add the right number of `:` if you skip them:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax"
    :
    :
    : "eax"
   );
# } }
```

Whitespace also doesn't matter:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
asm!("xor %eax, %eax" ::: "eax");
# } }
```

3.2.3.2 Operands

Input and output operands follow the same format: `: "constraints1"(expr1), "constraints2"(expr2), ..."`. Output operand expressions must be mutable lvalues:

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
fn add(a: int, b: int) -> int {
    let mut c = 0;
    unsafe {
        asm!("add $2, $0"
             : "=r"(c)
             : "0"(a), "r"(b)
             );
    }
    c
}
# #[cfg(not(any(target_arch = "x86", target_arch = "x86_64")))]
# fn add(a: int, b: int) -> int { a + b }

fn main() {
    assert_eq!(add(3, 14159), 14162)
}
```

3.2.3.3 Clobbers

Some instructions modify registers which might otherwise have held different values so we use the clobbers list to indicate to the compiler not to assume any values loaded into those registers will stay

valid.

```
# #![feature(asm)]
# #[cfg(any(target_arch = "x86", target_arch = "x86_64"))]
# fn main() { unsafe {
// Put the value 0x200 in eax
asm!("mov $$0x200, %eax" : /* no outputs */ : /* no inputs */ : "eax");
# } }
```

Input and output registers need not be listed since that information is already communicated by the given constraints. Otherwise, any other registers used either implicitly or explicitly should be listed.

If the assembly changes the condition code register `cc` should be specified as one of the clobbers. Similarly, if the assembly modifies memory, `memory` should also be specified.

3.2.3.4 Options

The last section, `options` is specific to Rust. The format is comma separated literal strings (i.e `:"foo", "bar", "baz"`). It's used to specify some extra info about the inline assembly:

Current valid options are:

1. *volatile* - specifying this is analogous to `__asm__ __volatile__ (...)` in gcc/clang.
2. *alignstack* - certain instructions expect the stack to be aligned a certain way (i.e SSE) and specifying this indicates to the compiler to insert its usual stack alignment code
3. *intel* - use intel syntax instead of the default AT&T.

3.2.4 Avoiding the standard library

By default, `std` is linked to every Rust crate. In some contexts, this is undesirable, and can be avoided with the `#![no_std]` attribute attached to the crate.

```
// a minimal library
#![crate_type="lib"]
#![no_std]
# // fn main() {} tricked you, rustdoc!
```

Obviously there's more to life than just libraries: one can use `#[no_std]` with an executable, controlling the entry point is possible in two ways: the `#[start]` attribute, or overriding the default shim for the C `main` function with your own.

The function marked `#[start]` is passed the command line parameters in the same format as C:

```
#![no_std]
#![feature(lang_items)]

// Pull in the system libc library for what crt0.o likely requires
extern crate libc;

// Entry point for this program
#[start]
fn start(_argc: int, _argv: *const *const u8) -> int {
    0
}
```

```
// These functions and traits are used by the compiler, but not
// for a bare-bones hello world. These are normally
// provided by libstd.
#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
# // fn main() {} tricked you, rustdoc!
```

To override the compiler-inserted `main` shim, one has to disable it with `#![no_main]` and then create the appropriate symbol with the correct ABI and the correct name, which requires overriding the compiler's name mangling too:

```
#![no_std]
#![no_main]
#![feature(lang_items)]

extern crate libc;

#[no_mangle] // ensure that this symbol is called `main` in the output
pub extern fn main(argc: int, argv: *const *const u8) -> int {
    0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
# // fn main() {} tricked you, rustdoc!
```

The compiler currently makes a few assumptions about symbols which are available in the executable to call. Normally these functions are provided by the standard library, but without it you must define your own.

The first of these three functions, `stack_exhausted`, is invoked whenever stack overflow is detected. This function has a number of restrictions about how it can be called and what it must do, but if the stack limit register is not being maintained then a task always has an "infinite stack" and this function shouldn't get triggered.

The second of these three functions, `eh_personality`, is used by the failure mechanisms of the compiler. This is often mapped to GCC's personality function (see the libstd implementation for more information), but crates which do not trigger a panic can be assured that this function is never called. The final function, `panic_fmt`, is also used by the failure mechanisms of the compiler.

3.2.4.1 Using libcore

> **Note**: the core library's structure is unstable, and it is recommended to use the standard library instead wherever possible.

With the above techniques, we've got a bare-metal executable running some Rust code. There is a good deal of functionality provided by the standard library, however, that is necessary to be productive in Rust. If the standard library is not sufficient, then libcore is designed to be used instead.

The core library has very few dependencies and is much more portable than the standard library itself. Additionally, the core library has most of the necessary functionality for writing idiomatic and

effective Rust code.

As an example, here is a program that will calculate the dot product of two vectors provided from C, using idiomatic Rust practices.

```
#![no_std]
#![feature(globs)]
#![feature(lang_items)]

# extern crate libc;
extern crate core;

use core::prelude::*;

use core::mem;

#[no_mangle]
pub extern fn dot_product(a: *const u32, a_len: u32,
                          b: *const u32, b_len: u32) -> u32 {
    use core::raw::Slice;

    // Convert the provided arrays into Rust slices.
    // The core::raw module guarantees that the Slice
    // structure has the same memory layout as a &[T]
    // slice.
    //
    // This is an unsafe operation because the compiler
    // cannot tell the pointers are valid.
    let (a_slice, b_slice): (&[u32], &[u32]) = unsafe {
        mem::transmute((
            Slice { data: a, len: a_len as uint },
            Slice { data: b, len: b_len as uint },
        ))
    };

    // Iterate over the slices, collecting the result
    let mut ret = 0;
    for (i, j) in a_slice.iter().zip(b_slice.iter()) {
        ret += (*i) * (*j);
    }
    return ret;
}

#[lang = "panic_fmt"]
extern fn panic_fmt(args: &core::fmt::Arguments,
                    file: &str,
                    line: uint) -> ! {
    loop {}
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
# #[start] fn start(argc: int, argv: *const *const u8) -> int { 0 }
# fn main() {}
```

Note that there is one extra lang item here which differs from the examples above, `panic_fmt`. This must be defined by consumers of libcore because the core library declares panics, but it does not define it. The `panic_fmt` lang item is this crate's definition of panic, and it must be guaranteed to never return.

As can be seen in this example, the core library is intended to provide the power of Rust in all circumstances, regardless of platform requirements. Further libraries, such as liballoc, add functionality to libcore which make other platform-specific assumptions, but continue to be more portable than the standard library itself.

3.2.5 Interacting with the compiler internals

> **Note**: this section is specific to the `rustc` compiler; these parts of the language may never be fully specified and so details may differ wildly between implementations (and even versions of `rustc` itself).
>
> Furthermore, this is just an overview; the best form of documentation for specific instances of these features are their definitions and uses in `std`.

The Rust language currently has two orthogonal mechanisms for allowing libraries to interact directly with the compiler and vice versa:

- intrinsics, functions built directly into the compiler providing very basic low-level functionality,
- lang-items, special functions, types and traits in libraries marked with specific `#[lang]` attributes

3.2.5.1 Intrinsics

> **Note**: intrinsics will forever have an unstable interface, it is recommended to use the stable interfaces of libcore rather than intrinsics directly.

These are imported as if they were FFI functions, with the special `rust-intrinsic` ABI. For example, if one was in a freestanding context, but wished to be able to `transmute` between types, and perform efficient pointer arithmetic, one would import those functions via a declaration like

```
# #![feature(intrinsics)]
# fn main() {}

extern "rust-intrinsic" {
    fn transmute<T, U>(x: T) -> U;

    fn offset<T>(dst: *const T, offset: int) -> *const T;
}
```

As with any other FFI functions, these are always `unsafe` to call.

3.2.5.2 Lang items

> **Note**: lang items are often provided by crates in the Rust distribution, and lang items themselves have an unstable interface. It is recommended to use officially distributed crates instead of defining your own lang items.

The `rustc` compiler has certain pluggable operations, that is, functionality that isn't hard-coded into the language, but is implemented in libraries, with a special marker to tell the compiler it exists. The marker is the attribute `#[lang="..."]` and there are various different values of `...`, i.e. various different "lang items".

For example, `Box` pointers require two lang items, one for allocation and one for deallocation. A freestanding program that uses the `Box` sugar for dynamic allocations via `malloc` and `free`:

```
#![no_std]
#![feature(lang_items, box_syntax)]

extern crate libc;

extern {
    fn abort() -> !;
}

#[lang = "owned_box"]
pub struct Box<T>(*mut T);

#[lang="exchange_malloc"]
unsafe fn allocate(size: uint, _align: uint) -> *mut u8 {
    let p = libc::malloc(size as libc::size_t) as *mut u8;

    // malloc failed
    if p as uint == 0 {
        abort();
    }

    p
}
#[lang="exchange_free"]
unsafe fn deallocate(ptr: *mut u8, _size: uint, _align: uint) {
    libc::free(ptr as *mut libc::c_void)
}

#[start]
fn main(argc: int, argv: *const *const u8) -> int {
    let x = box 1i;

    0
}

#[lang = "stack_exhausted"] extern fn stack_exhausted() {}
#[lang = "eh_personality"] extern fn eh_personality() {}
#[lang = "panic_fmt"] fn panic_fmt() -> ! { loop {} }
```

Note the use of `abort`: the `exchange_malloc` lang item is assumed to return a valid pointer, and so needs to do the check internally.

Other features provided by lang items include:

- overloadable operators via traits: the traits corresponding to the `==`, `<`, dereferencing (`*`) and `+` (etc.) operators are all marked with lang items; those specific four are `eq`, `ord`, `deref`, and `add` respectively.
- stack unwinding and general failure; the `eh_personality`, `fail` and `fail_bounds_checks` lang items.
- the traits in `std::markers` used to indicate types of various kinds; lang items `send`, `sync` and `copy`.
- the marker types and variance indicators found in `std::markers`; lang items `covariant_type`, `contravariant_lifetime`, `no_sync_bound`, etc.

Lang items are loaded lazily by the compiler; e.g. if one never uses `Box` then there is no need to define functions for `exchange_malloc` and `exchange_free`. `rustc` will emit an error when an item is needed but not found in the current crate or any that it depends on.

3.3 Macros

3.3.1 Introduction

Functions are the primary tool that programmers can use to build abstractions. Sometimes, however, programmers want to abstract over compile-time syntax rather than run-time values. Macros provide syntactic abstraction. For an example of how this can be useful, consider the following two code fragments, which both pattern-match on their input and both return early in one case, doing nothing otherwise:

```
# enum T { SpecialA(uint), SpecialB(uint) }
# fn f() -> uint {
# let input_1 = T::SpecialA(0);
# let input_2 = T::SpecialA(0);
match input_1 {
    T::SpecialA(x) => { return x; }
    _ => {}
}
// ...
match input_2 {
    T::SpecialB(x) => { return x; }
    _ => {}
}
# return 0u;
# }
```

This code could become tiresome if repeated many times. However, no function can capture its functionality to make it possible to abstract the repetition away. Rust's macro system, however, can eliminate the repetition. Macros are lightweight custom syntax extensions, themselves defined using the `macro_rules!` syntax extension. The following `early_return` macro captures the pattern in the above code:

```
# enum T { SpecialA(uint), SpecialB(uint) }
# fn f() -> uint {
# let input_1 = T::SpecialA(0);
# let input_2 = T::SpecialA(0);
macro_rules! early_return {
    ($inp:expr, $sp:path) => ( // invoke it like `(input_5 SpecialE)`
        match $inp {
            $sp(x) => { return x; }
            _ => {}
        }
    );
}
// ...
early_return!(input_1, T::SpecialA);
// ...
early_return!(input_2, T::SpecialB);
# return 0;
# }
# fn main() {}
```

Macros are defined in pattern-matching style: in the above example, the text `($inp:expr $sp:ident)` that appears on the left-hand side of the `=>` is the *macro invocation syntax*, a pattern denoting how to write a call to the macro. The text on the right-hand side of the `=>`, beginning with `match $inp`, is the *macro transcription syntax*: what the macro expands to.

3.3.2 Invocation syntax

The macro invocation syntax specifies the syntax for the arguments to the macro. It appears on the left-hand side of the `=>` in a macro definition. It conforms to the following rules:

1. It must be surrounded by parentheses.
2. `$` has special meaning (described below).
3. The `()`s, `[]`s, and `{}`s it contains must balance. For example, `([)` is forbidden.

Otherwise, the invocation syntax is free-form.

To take a fragment of Rust code as an argument, write `$` followed by a name (for use on the right-hand side), followed by a `:`, followed by a *fragment specifier*. The fragment specifier denotes the sort of fragment to match. The most common fragment specifiers are:

- `ident` (an identifier, referring to a variable or item. Examples: `f`, `x`, `foo`.)
- `expr` (an expression. Examples: `2 + 2`; `if true then { 1 } else { 2 }`; `f(42)`.)
- `ty` (a type. Examples: `int`, `Vec<(char, String)>`, `&T`.)
- `pat` (a pattern, usually appearing in a `match` or on the left-hand side of a declaration. Examples: `Some(t)`; `(17, 'a')`; `_`.)
- `block` (a sequence of actions. Example: `{ log(error, "hi"); return 12; }`)

The parser interprets any token that's not preceded by a `$` literally. Rust's usual rules of tokenization apply,

So `($x:ident -> (($e:expr)))`, though excessively fancy, would designate a macro that could be invoked like: `my_macro!(i->(( 2+2 )))`.

3.3.2.1 Invocation location

A macro invocation may take the place of (and therefore expand to) an expression, item, statement, or pattern. The Rust parser will parse the macro invocation as a "placeholder" for whichever syntactic form is appropriate for the location.

At expansion time, the output of the macro will be parsed as whichever of the three nonterminals it stands in for. This means that a single macro might, for example, expand to an item or an expression, depending on its arguments (and cause a syntax error if it is called with the wrong argument for its location). Although this behavior sounds excessively dynamic, it is known to be useful under some circumstances.

3.3.3 Transcription syntax

The right-hand side of the `=>` follows the same rules as the left-hand side, except that a `$` need only be followed by the name of the syntactic fragment to transcribe into the macro expansion; its type

need not be repeated.

The right-hand side must be enclosed by delimiters, which the transcriber ignores. Therefore `() => ((1,2,3))` is a macro that expands to a tuple expression, `() => (let $x=$val)` is a macro that expands to a statement, and `() => (1,2,3)` is a macro that expands to a syntax error (since the transcriber interprets the parentheses on the right-hand-size as delimiters, and `1,2,3` is not a valid Rust expression on its own).

Except for permissibility of `$name` (and `$(...)*`, discussed below), the right-hand side of a macro definition is ordinary Rust syntax. In particular, macro invocations (including invocations of the macro currently being defined) are permitted in expression, statement, and item locations. However, nothing else about the code is examined or executed by the macro system; execution still has to wait until run-time.

3.3.3.1 Interpolation location

The interpolation `$argument_name` may appear in any location consistent with its fragment specifier (i.e., if it is specified as `ident`, it may be used anywhere an identifier is permitted).

3.3.4 Multiplicity

3.3.4.1 Invocation

Going back to the motivating example, recall that `early_return` expanded into a `match` that would `return` if the `match`'s scrutinee matched the "special case" identifier provided as the second argument to `early_return`, and do nothing otherwise. Now suppose that we wanted to write a version of `early_return` that could handle a variable number of "special" cases.

The syntax `$(...)*` on the left-hand side of the `=>` in a macro definition accepts zero or more occurrences of its contents. It works much like the `*` operator in regular expressions. It also supports a separator token (a comma-separated list could be written `$(...),*`), and `+` instead of `*` to mean "at least one".

```
# enum T { SpecialA(uint),SpecialB(uint),SpecialC(uint),SpecialD(uint)}
# fn f() -> uint {
# let input_1 = T::SpecialA(0);
# let input_2 = T::SpecialA(0);
macro_rules! early_return {
    ($inp:expr, [ $($sp:path),+ ]) => (
        match $inp {
            $(
                $sp(x) => { return x; }
            )+
            _ => {}
        }
    )
}
// ...
early_return!(input_1, [T::SpecialA,T::SpecialC,T::SpecialD]);
// ...
early_return!(input_2, [T::SpecialB]);
# return 0;
```

```
# }
# fn main() {}
```

3.3.4.1.1 Transcription

As the above example demonstrates, `$(...)*` is also valid on the right-hand side of a macro definition. The behavior of `*` in transcription, especially in cases where multiple `*`s are nested, and multiple different names are involved, can seem somewhat magical and unintuitive at first. The system that interprets them is called "Macro By Example". The two rules to keep in mind are (1) the behavior of `$(...)*` is to walk through one "layer" of repetitions for all of the `$name`s it contains in lockstep, and (2) each `$name` must be under at least as many `$(...)*`s as it was matched against. If it is under more, it'll be repeated, as appropriate.

3.3.4.2 Parsing limitations

For technical reasons, there are two limitations to the treatment of syntax fragments by the macro parser:

1. The parser will always parse as much as possible of a Rust syntactic fragment. For example, if the comma were omitted from the syntax of `early_return!` above, `input_1 [` would've been interpreted as the beginning of an array index. In fact, invoking the macro would have been impossible.
2. The parser must have eliminated all ambiguity by the time it reaches a `$name:fragment_specifier` declaration. This limitation can result in parse errors when declarations occur at the beginning of, or immediately after, a `$(...)*`. For example, the grammar `$($t:ty)* $e:expr` will always fail to parse because the parser would be forced to choose between parsing `t` and parsing `e`. Changing the invocation syntax to require a distinctive token in front can solve the problem. In the above example, `$(T $t:ty)* E $e:exp` solves the problem.

3.3.5 Macro argument pattern matching

3.3.5.1 Motivation

Now consider code like the following:

```
# enum T1 { Good1(T2, uint), Bad1}
# struct T2 { body: T3 }
# enum T3 { Good2(uint), Bad2}
# fn f(x: T1) -> uint {
match x {
    T1::Good1(g1, val) => {
        match g1.body {
            T3::Good2(result) => {
                // complicated stuff goes here
                return result + val;
            },
            _ => panic!("Didn't get good_2")
        }
    }
    _ => return 0 // default value
}
```

```
# }
# fn main() {}
```

All the complicated stuff is deeply indented, and the error-handling code is separated from matches that fail. We'd like to write a macro that performs a match, but with a syntax that suits the problem better. The following macro can solve the problem:

```
macro_rules! biased_match {
    // special case: `let (x) = ...` is illegal, so use `let x = ...` instead
    ( ($e:expr) -> ($p:pat) else $err:stmt ;
      binds $bind_res:ident
    ) => (
        let $bind_res = match $e {
            $p => ( $bind_res ),
            _ => { $err }
        };
    );
    // more than one name; use a tuple
    ( ($e:expr) -> ($p:pat) else $err:stmt ;
      binds $( $bind_res:ident ),*
    ) => (
        let ( $( $bind_res ),* ) = match $e {
            $p => ( $( $bind_res ),* ),
            _ => { $err }
        };
    )
}

# enum T1 { Good1(T2, uint), Bad1}
# struct T2 { body: T3 }
# enum T3 { Good2(uint), Bad2}
# fn f(x: T1) -> uint {
biased_match!((x)       -> (T1::Good1(g1, val)) else { return 0 };
              binds g1, val );
biased_match!((g1.body) -> (T3::Good2(result) )
                  else { panic!("Didn't get good_2") };
              binds result );
// complicated stuff goes here
return result + val;
# }
# fn main() {}
```

This solves the indentation problem. But if we have a lot of chained matches like this, we might prefer to write a single macro invocation. The input pattern we want is clear:

```
# fn main() {}
# macro_rules! b {
    ( $( ($e:expr) -> ($p:pat) else $err:stmt ; )*
      binds $( $bind_res:ident ),*
    )
# => (0) }
```

However, it's not possible to directly expand to nested match statements. But there is a solution.

3.3.5.2 The recursive approach to macro writing

A macro may accept multiple different input grammars. The first one to successfully match the actual argument to a macro invocation is the one that "wins".

In the case of the example above, we want to write a recursive macro to process the semicolon-terminated lines, one-by-one. So, we want the following input patterns:

```
# macro_rules! b {
    ( binds $( $bind_res:ident ),* )
# => (0) }
# fn main() {}
```

…and:

```
# fn main() {}
# macro_rules! b {
    (    ($e     :expr) -> ($p     :pat) else $err     :stmt ;
      $( ($e_rest:expr) -> ($p_rest:pat) else $err_rest:stmt ; )*
      binds  $( $bind_res:ident ),*
    )
# => (0) }
```

The resulting macro looks like this. Note that the separation into `biased_match!` and `biased_match_rec!` occurs only because we have an outer piece of syntax (the `let`) which we only want to transcribe once.

```
# fn main() {

macro_rules! biased_match_rec {
    // Handle the first layer
    (   ($e     :expr) -> ($p     :pat) else $err     :stmt ;
     $( ($e_rest:expr) -> ($p_rest:pat) else $err_rest:stmt ; )*
     binds $( $bind_res:ident ),*
    ) => (
        match $e {
            $p => {
                // Recursively handle the next layer
                biased_match_rec!($( ($e_rest) -> ($p_rest) else $err_rest ; )*
                                  binds $( $bind_res ),*
                )
            }
            _ => { $err }
        }
    );
    // Produce the requested values
    ( binds $( $bind_res:ident ),* ) => ( ($( $bind_res ),*) )
}

// Wrap the whole thing in a `let`.
macro_rules! biased_match {
    // special case: `let (x) = ...` is illegal, so use `let x = ...` instead
    ( $( ($e:expr) -> ($p:pat) else $err:stmt ; )*
      binds $bind_res:ident
    ) => (
        let $bind_res = biased_match_rec!(
            $( ($e) -> ($p) else $err ; )*
            binds $bind_res
        );
    );
    // more than one name: use a tuple
    ( $( ($e:expr) -> ($p:pat) else $err:stmt ; )*
      binds  $( $bind_res:ident ),*
    ) => (
        let ( $( $bind_res ),* ) = biased_match_rec!(
            $( ($e) -> ($p) else $err ; )*
```

```
            binds $( $bind_res ),*
        );
    )
}


# enum T1 { Good1(T2, uint), Bad1}
# struct T2 { body: T3 }
# enum T3 { Good2(uint), Bad2}
# fn f(x: T1) -> uint {
biased_match!(
    (x)       -> (T1::Good1(g1, val)) else { return 0 };
    (g1.body) -> (T3::Good2(result) ) else { panic!("Didn't get Good2") };
    binds val, result );
// complicated stuff goes here
return result + val;
# }
# }
```

This technique applies to many cases where transcribing a result all at once is not possible. The resulting code resembles ordinary functional programming in some respects, but has some important differences from functional programming.

The first difference is important, but also easy to forget: the transcription (right-hand) side of a `macro_rules!` rule is literal syntax, which can only be executed at run-time. If a piece of transcription syntax does not itself appear inside another macro invocation, it will become part of the final program. If it is inside a macro invocation (for example, the recursive invocation of `biased_match_rec!`), it does have the opportunity to affect transcription, but only through the process of attempted pattern matching.

The second, related, difference is that the evaluation order of macros feels "backwards" compared to ordinary programming. Given an invocation `m1!(m2!())`, the expander first expands `m1!`, giving it as input the literal syntax `m2!()`. If it transcribes its argument unchanged into an appropriate position (in particular, not as an argument to yet another macro invocation), the expander will then proceed to evaluate `m2!()` (along with any other macro invocations `m1!(m2!())` produced).

3.3.6 Hygiene

To prevent clashes, rust implements hygienic macros.

As an example, `loop` and `for-loop` labels (discussed in the lifetimes guide) will not clash. The following code will print "Hello!" only once:

```
macro_rules! loop_x {
    ($e: expr) => (
        // $e will not interact with this 'x
        'x: loop {
            println!("Hello!");
            $e
        }
    );
}

fn main() {
    'x: loop {
        loop_x!(break 'x);
        println!("I am never printed.");
```

```
    }
}
```

The two `'x` names did not clash, which would have caused the loop to print “I am never printed” and to run forever.

3.3.7 Scoping and macro import/export

Macros are expanded at an early stage in compilation, before name resolution. One downside is that scoping works differently for macros, compared to other constructs in the language.

Definition and expansion of macros both happen in a single depth-first, lexical-order traversal of a crate’s source. So a macro defined at module scope is visible to any subsequent code in the same module, which includes the body of any subsequent child `mod` items.

A macro defined within the body of a single `fn`, or anywhere else not at module scope, is visible only within that item.

If a module has the `macro_use` attribute, its macros are also visible in its parent module after the child’s `mod` item. If the parent also has `macro_use` then the macros will be visible in the grandparent after the parent’s `mod` item, and so forth.

The `macro_use` attribute can also appear on `extern crate`. In this context it controls which macros are loaded from the external crate, e.g.

```
#[macro_use(foo, bar)]
extern crate baz;
```

If the attribute is given simply as `#[macro_use]`, all macros are loaded. If there is no `#[macro_use]` attribute then no macros are loaded. Only macros defined with the `#[macro_export]` attribute may be loaded.

To load a crate’s macros *without* linking it into the output, use `#[no_link]` as well.

An example:

```
macro_rules! m1 { () => (()) }

// visible here: m1

mod foo {
    // visible here: m1

    #[macro_export]
    macro_rules! m2 { () => (()) }

    // visible here: m1, m2
}

// visible here: m1

macro_rules! m3 { () => (()) }

// visible here: m1, m3
```

```
#[macro_use]
mod bar {
    // visible here: m1, m3

    macro_rules! m4 { () => (()) }

    // visible here: m1, m3, m4
}

// visible here: m1, m3, m4
# fn main() { }
```

When this library is loaded with `#[use_macros] extern crate`, only `m2` will be imported.

The Rust Reference has a listing of macro-related attributes.

3.3.8 The variable `$crate`

A further difficulty occurs when a macro is used in multiple crates. Say that `mylib` defines

```
pub fn increment(x: uint) -> uint {
    x + 1
}

#[macro_export]
macro_rules! inc_a {
    ($x:expr) => ( ::increment($x) )
}

#[macro_export]
macro_rules! inc_b {
    ($x:expr) => ( ::mylib::increment($x) )
}
# fn main() { }
```

`inc_a` only works within `mylib`, while `inc_b` only works outside the library. Furthermore, `inc_b` will break if the user imports `mylib` under another name.

Rust does not (yet) have a hygiene system for crate references, but it does provide a simple workaround for this problem. Within a macro imported from a crate named `foo`, the special macro variable `$crate` will expand to `::foo`. By contrast, when a macro is defined and then used in the same crate, `$crate` will expand to nothing. This means we can write

```
#[macro_export]
macro_rules! inc {
    ($x:expr) => ( $crate::increment($x) )
}
# fn main() { }
```

to define a single macro that works both inside and outside our library. The function name will expand to either `::increment` or `::mylib::increment`.

To keep this system simple and correct, `#[macro_use] extern crate ...` may only appear at the root of your crate, not inside `mod`. This ensures that `$crate` is a single identifier.

3.3.9 A final note

Macros, as currently implemented, are not for the faint of heart. Even ordinary syntax errors can be more difficult to debug when they occur inside a macro, and errors caused by parse problems in generated code can be very tricky. Invoking the `log_syntax!` macro can help elucidate intermediate states, invoking `trace_macros!(true)` will automatically print those intermediate states out, and passing the flag `--pretty expanded` as a command-line argument to the compiler will show the result of expansion.

If Rust's macro system can't do what you need, you may want to write a compiler plugin instead. Compared to `macro_rules!` macros, this is significantly more work, the interfaces are much less stable, and the warnings about debugging apply ten-fold. In exchange you get the flexibility of running arbitrary Rust code within the compiler. Syntax extension plugins are sometimes called *procedural macros* for this reason.

3.4 Compiler Plugins

Warning: Plugins are an advanced, unstable feature! For many details, the only available documentation is the `libsyntax` and `librustc` API docs, or even the source code itself. These internal compiler APIs are also subject to change at any time.

For defining new syntax it is often much easier to use Rust's built-in macro system.

The code in this document uses language features not covered in the Rust Guide. See the Reference Manual for more information.

3.4.1 Introduction

`rustc` can load compiler plugins, which are user-provided libraries that extend the compiler's behavior with new syntax extensions, lint checks, etc.

A plugin is a dynamic library crate with a designated *registrar* function that registers extensions with `rustc`. Other crates can use these extensions by loading the plugin crate with `#[plugin] extern crate`. See the `rustc::plugin` documentation for more about the mechanics of defining and loading a plugin.

Arguments passed as `#[plugin=...]` or `#[plugin(...)]` are not interpreted by rustc itself. They are provided to the plugin through the `Registry`'s `args` method.

3.4.2 Syntax extensions

Plugins can extend Rust's syntax in various ways. One kind of syntax extension is the procedural macro. These are invoked the same way as ordinary macros, but the expansion is performed by arbitrary Rust code that manipulates syntax trees at compile time.

Let's write a plugin `roman_numerals.rs` that implements Roman numeral integer literals.

```
#![crate_type="dylib"]
#![feature(plugin_registrar)]

extern crate syntax;
extern crate rustc;
```

```
use syntax::codemap::Span;
use syntax::parse::token;
use syntax::ast::{TokenTree, TtToken};
use syntax::ext::base::{ExtCtxt, MacResult, DummyResult, MacExpr};
use syntax::ext::build::AstBuilder;  // trait for expr_uint
use rustc::plugin::Registry;

fn expand_rn(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult + 'static> {

    static NUMERALS: &'static [(&'static str, uint)] = &[
        ("M", 1000), ("CM", 900), ("D", 500), ("CD", 400),
        ("C",  100), ("XC",  90), ("L",  50), ("XL",  40),
        ("X",   10), ("IX",   9), ("V",   5), ("IV",   4),
        ("I",    1)];

    let text = match args {
        [TtToken(_, token::Ident(s, _))] => token::get_ident(s).to_string(),
        _ => {
            cx.span_err(sp, "argument should be a single identifier");
            return DummyResult::any(sp);
        }
    };

    let mut text = text.as_slice();
    let mut total = 0u;
    while !text.is_empty() {
        match NUMERALS.iter().find(|&&(rn, _)| text.starts_with(rn)) {
            Some(&(rn, val)) => {
                total += val;
                text = text.slice_from(rn.len());
            }
            None => {
                cx.span_err(sp, "invalid Roman numeral");
                return DummyResult::any(sp);
            }
        }
    }

    MacExpr::new(cx.expr_uint(sp, total))
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_macro("rn", expand_rn);
}
```

Then we can use `rn!()` like any other macro:

```
#![feature(plugin)]

#[plugin] extern crate roman_numerals;

fn main() {
    assert_eq!(rn!(MMXV), 2015);
}
```

The advantages over a simple `fn(&str) -> uint` are:

- The (arbitrarily complex) conversion is done at compile time.

- Input validation is also performed at compile time.
- It can be extended to allow use in patterns, which effectively gives a way to define new literal syntax for any data type.

In addition to procedural macros, you can define new `deriving`-like attributes and other kinds of extensions. See `Registry::register_syntax_extension` and the `SyntaxExtension` enum. For a more involved macro example, see `regex_macros`.

3.4.2.1 Tips and tricks

To see the results of expanding syntax extensions, run `rustc --pretty expanded`. The output represents a whole crate, so you can also feed it back in to `rustc`, which will sometimes produce better error messages than the original compilation. Note that the `--pretty expanded` output may have a different meaning if multiple variables of the same name (but different syntax contexts) are in play in the same scope. In this case `--pretty expanded,hygiene` will tell you about the syntax contexts.

You can use `syntax::parse` to turn token trees into higher-level syntax elements like expressions:

```
fn expand_foo(cx: &mut ExtCtxt, sp: Span, args: &[TokenTree])
        -> Box<MacResult+'static> {

    let mut parser = cx.new_parser_from_tts(args);

    let expr: P<Expr> = parser.parse_expr();
```

Looking through `libsyntax` parser code will give you a feel for how the parsing infrastructure works.

Keep the `Span`s of everything you parse, for better error reporting. You can wrap `Spanned` around your custom data structures.

Calling `ExtCtxt::span_fatal` will immediately abort compilation. It's better to instead call `ExtCtxt::span_err` and return `DummyResult`, so that the compiler can continue and find further errors.

The example above produced an integer literal using `AstBuilder::expr_uint`. As an alternative to the `AstBuilder` trait, `libsyntax` provides a set of quasiquote macros. They are undocumented and very rough around the edges. However, the implementation may be a good starting point for an improved quasiquote as an ordinary plugin library.

3.4.3 Lint plugins

Plugins can extend Rust's lint infrastructure with additional checks for code style, safety, etc. You can see `src/test/auxiliary/lint_plugin_test.rs` for a full example, the core of which is reproduced here:

```
declare_lint!(TEST_LINT, Warn,
              "Warn about items named 'lintme'")

struct Pass;
```

```
impl LintPass for Pass {
    fn get_lints(&self) -> LintArray {
        lint_array!(TEST_LINT)
    }

    fn check_item(&mut self, cx: &Context, it: &ast::Item) {
        let name = token::get_ident(it.ident);
        if name.get() == "lintme" {
            cx.span_lint(TEST_LINT, it.span, "item is named 'lintme'");
        }
    }
}

#[plugin_registrar]
pub fn plugin_registrar(reg: &mut Registry) {
    reg.register_lint_pass(box Pass as LintPassObject);
}
```

Then code like

```
#[plugin] extern crate lint_plugin_test;

fn lintme() { }
```

will produce a compiler warning:

```
foo.rs:4:1: 4:16 warning: item is named 'lintme', #[warn(test_lint)] on by default
foo.rs:4 fn lintme() { }
         ^~~~~~~~~~~~~~~
```

The components of a lint plugin are:

- one or more `declare_lint!` invocations, which define static `Lint` structs;
- a struct holding any state needed by the lint pass (here, none);
- a `LintPass` implementation defining how to check each syntax element. A single `LintPass` may call `span_lint` for several different `Lint`s, but should register them all through the `get_lints` method.

Lint passes are syntax traversals, but they run at a late stage of compilation where type information is available. `rustc`'s built-in lints mostly use the same infrastructure as lint plugins, and provide examples of how to access type information.

Lints defined by plugins are controlled by the usual attributes and compiler flags, e.g. `#[allow(test_lint)]` or `-A test-lint`. These identifiers are derived from the first argument to `declare_lint!`, with appropriate case and punctuation conversion.

You can run `rustc -W help foo.rs` to see a list of lints known to `rustc`, including those provided by plugins loaded by `foo.rs`.

4 Conclusion

We covered a lot of ground here. When you've mastered everything in this Guide, you will have a firm grasp of Rust development. There's a whole lot more out there, though, we've just covered the surface. There's tons of topics that you can dig deeper into, e.g. by reading the API documentation of the standard library, by discovering solutions for common problems on Rust by Example, or by browsing crates written by the community on crates.io.

Happy hacking!