Rust for Rubyists

Steve Klabnik

Contents

Preamble

Nobody should only learn one programming language. Before Ruby, I wrote things in C, C++, Java, Perl, PHP, and all sorts of things. I've always said that I love Ruby, but maybe someday, something would come along that'd be better for what I was building.

I'm not sure if Rust is that language yet, but it is damn interesting. One of the worst things about Rust is that there's next to no documentation, though. So, I decided that as I figured things out, I'd write them out so that you don't have to suffer the way I did. Maybe 'suffer' is too strong a word; Rust is worth the hassle, though.

In this book, we'll talk about why you should care about Rust, how to get up and running, the basics of writing software in Rust, and maybe even something like building a Ruby gem with Rust.

NOTE: While this book is called "Rust for Rubyists," it should be accessible to anyone who knows about OOP and programming in a dynamically typed language. I will make analogies that will make the most sense to Rubyists, but you can still get a lot out of this book if you are a programmer of a different sort. If your favorite language is static, then you're already ahead of the game. You can just disregard a lot of the analogies.

Why care about Rust?

You already write software in Ruby. It pays your bills. You enjoy it. Why should you care about Rust?

Alan Perlis once said:

> A language that doesn't affect the way you think about programming is not worth knowing.

Let's think about Ruby for a minute: what's its biggest weakness? For me, it's these things:

- Concurrency
- Safety guarantees
- Lots of mutable state
- Only vaguely functional
- Speed
- Complexity. (Smalltalk's semantics fit on an index card)
- Documentation
- nil

What's awesome about Ruby?

- Blocks
- Vaguely functional
- Syntax is pretty easy

- Focus on developer happiness
- Get up and running quickly
- Dynamically typed

So we could learn a lot from a language that handles concurrency well, has good safety guarantees, is immutable by default, and is fast and simple. We don't want to sacrifice anonymous functions, pretty syntax, or not making `AbstractFactoryFactoryImpls` just to get work done.

I think that that language is Rust.

Now: Rust is not perfect, by far. Its documentation is poor, but getting better, as I've been hired by Mozilla to fix it. It can feel quite complex. Fighting with a compiler can be frustrating. But the point is to *learn*. And using a language that's very familiar, yet very different, can teach us a lot.

Here's "Hello World" in Rust:

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

Here's a parallel "Hello World" in Rust:

```
fn main() {
    for _ in range(0u, 10) {
        spawn(proc() {
            let greeting_message = "Hello?";
            println!("{}", greeting_message);
        });
    }
}
```

Here's a rough port to Ruby:

```
10.times do
  Thread.new do
    greeting_message = "Hello?"

    # This is weird in Ruby but it's closer to the println! macro
    # usage in the Rust example.
    puts "#{greeting_message}"
  end
end
```

That's it. Note the stuff that's *similar* to Ruby:

- Variables are in `snake_case`
- We have 'blocks' that use `{}`. No `do/end` though.
- Variables, while statically typed, have inference, so we don't need to declare types

Here's some stuff that's *different*:

- `;` s everywhere. You don't always need them, but let's put them in for now.
- slightly different syntax, `fn` rather than `def`.
- Because we have no `do/end`, we use `{}` s instead.
- The compiler will yell at us harder if we mess up.

Oh, and:

```
$ time ./hello
./hello 0.01s user 0.01s system 91% cpu 0.014 total

$ time ruby hello.rb
ruby hello.rb 0.02s user 0.01s system 95% cpu 0.026 total
```

Twice as fast. Yay irrelevant microbenchmarks!

Anyway, I hope you get my point: There's lots of things about Rust that make it syntactically vaguely similar enough to Ruby that you can feel at home, at least at first. And its strengths are some of Ruby's greatest weaknesses. That's why I think you can learn a lot from playing with Rust, even if you don't do it as your day job.

Installing Rust

Binary installers

The Rust project provides official binary installers. You can get both releases and nightlies. Binary installers are the fastest and easiest way to get going with Rust. Because Rust is written in Rust, compiling the Rust compiler actually entails compiling it three times! This means it's quite slow. But a binary install should be snappy!

Rust now has a lovely downloads page, so I recommend you just go check that out and download the proper version.

Note that this book has been tested with Rust 0.11, and so if you use the latest nightly, something may have changed.

From Source

You will probably build the nightly version if you build from source, so be ready for some bugs in the code samples. This book was written for 0.11.

The Rust README has great instructions for building form source. Just got follow their instructions!

Future Proofing

The version this book is written for is 0.11. While the language itself is pretty stable, things like the standard library and some major subsystems are being revised. I'll be tweaking it with every new release.

If you run

```
$ rustc
```

and it spits out a bunch of help information, you're good to go with Rust.

Writing Your First Rust Program

Okay! Let's get down to it: in order to call yourself an "X Programmer," you must write "Hello, world" in X. So let's do it. Open up a text file: I'll use `vim` because I'm that kind of guy, but use whatever you want. Rust programs end in `.rs`:

```
$ vim hello.rs
```

Put this in it:

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

And compile it with `rustc`:

```
$ rustc hello.rs
```

It should compile without error. If you get one, double check that you have the semicolons, the curlies, double quotation marks, and the parentheses. Errors look like this:

```
$ rustc hello.rs
hello.rs:2:4: 2:11 error: expected `{` but found `println`
hello.rs:2     println("Hello, world.");
               ^~~~~~~
```

This happened when I left off the curly brace after the `main` function above.

This isn't an error:

```
$ rustc hello.rs
warning: no debug symbols in executable (-arch x86_64)
```

It happens on OSX for some versions of Rust. You can safely ignore it.

To run your program, do the Usual UNIX Thing:

```
$ ./hello
```

And you should see "Hello, world." print to the screen. Congrats!

Testing

Rubyists love testing, so before we go any farther, let's talk about testing. In Rust, there is a unit testing framework built in, and it's pretty simple. Let's write some very simple code and tests to exercise it.

In Rust, you annotate test methods like such:

```
#[test]
fn this_tests_code() {
    // SOMETHING HERE
}
```

You'll note that tests take no arguments and return nothing. If the function runs, the test passes, and if it errors in some way, the test fails. Let's give it a shot: Open up `testing.rs` and put this in it:

```
#[test]
fn this_tests_code() {
    println!("");
}
```

Then, use `rustc` with a special flag:

```
$ rustc --test testing.rs
```

This tells `rustc` to compile your tests, and replaces the `main` function with a test runner. Try it out:

```
$ ./testing
```

You should get some output that looks like this:

```
running 1 test

test this_tests_code ... ok

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

Bam! Now let's make it fail:

```
#[test]
fn this_tests_code() {
    fail!("Fail!");
}
```

Recompile, and the output should be:

```
running 1 test
test this_tests_code ... FAILED

failures:

---- this_tests_code stdout ----
    task 'this_tests_code' failed at 'Fail!', testing.rs:5


failures:
    this_tests_code

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

task '<main>' failed at 'Some tests failed', /some/path/to/something
```

Super simple. That's all you need to know to get started. Next up: FizzBuzz.

FizzBuzz

Of course, the first thing that your job interview for that cushy new Rust job will task you with is building FizzBuzz. Let's do it!

If you're not familiar, FizzBuzz is a simple programming problem:

> "Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz"."

This will give us a good excuse to go over some basics of Rust: Looping, tests, printing to standard output, and a host of other simple things.

First, a test. This will go in fizzbuzz.rs:

```
#[test]
fn test_div_by_three() {
    if div_by_three(1) {
        fail!("One is not three");
    }
}
```

And compile it:

```
$ rustc --test fizzbuzz.rs
fizzbuzz.rs:3:8: 3:20 error: unresolved name `div_by_three`.
fizzbuzz.rs:3     if div_by_three(1) {
                     ^~~~~~~~~~~~
error: aborting due to previous error
```

This makes sense: We haven't defined any functions yet. Let's define one:

```
fn div_by_three(num: int) -> bool {
   true
}

#[test]
fn test_div_by_three() {
    if div_by_three(1) {
        fail!("One is not three");
    }
}
```

Okay. Here's some new syntax. The `num: int` says that we take one argument, `num`, and that it's of an integer type. The `-> bool` says that we return a boolean, and the `true`, well, returns true. Just like Ruby, the value of the last expression gets returned.

You'll also note we have an `if` expression. It's pretty close to what you'd expect, but we have curly braces rather than our friends `do/end`.

Now that we've got that cleared up, let's compile and run our tests:

```
$ rustc --test fizzbuzz.rs
fizzbuzz.rs:1:17: 1:18 warning: unused variable: `num`, #[warn(unused_variable)] on by defau
fizzbuzz.rs:1 fn div_by_three(num: int) -> bool {
                              ^


running 1 test
test test_div_by_three ... FAILED

failures:

---- test_div_by_three stdout ----
    task 'test_div_by_three' failed at 'One is not three', fizzbuzz.rs:8



failures:
    test_div_by_three

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

task '<main>' failed at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:243
```

Rust is kind enough to give us a warning: we never used the `num` argument. We then get our failure, "One is not three", because we returned true. Now that we have a failing test, let's make it pass:

```rust
fn div_by_three(num: int) -> bool {
   false
}

#[test]
fn test_div_by_three() {
    if div_by_three(1) {
        fail!("One is not three");
    }
}
```

TDD means do the simplest thing! Compile and run it:

```
$ rustc --test fizzbuzz.rs
fizzbuzz.rs:1:17: 1:18 warning: unused variable: `num`, #[warn(unused_variable)] on by defau
fizzbuzz.rs:1 fn div_by_three(num: int) -> bool {
                              ^


$ ./fizzbuzz
running 1 test
test test_div_by_three ... ok

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

Awesome! We pass! We still have that warning, though… let's write another test, and see what happens:

```
fn div_by_three(num: int) -> bool {
   false
}

#[test]
fn test_div_by_three() {
    if div_by_three(1) {
        fail!("One is not three");
    }
}

#[test]
fn test_div_by_three_with_three() {
    if !div_by_three(3) {
        fail!("Three should be three");
    }
}
```

```
$ rustc --test fizzbuzz.rs
fizzbuzz.rs:1:17: 1:18 warning: unused variable: `num`, #[warn(unused_variable)] on by defau
fizzbuzz.rs:1 fn div_by_three(num: int) -> bool {
                              ^



$ ./fizzbuzz
running 2 tests
test test_div_by_three ... ok
test test_div_by_three_with_three ... FAILED
```

```
failures:

---- test_div_by_three_with_three stdout ----
    task 'test_div_by_three_with_three' failed at 'Three should be three', fizzbuzz.rs:15



failures:
    test_div_by_three_with_three

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

task '<main>' failed at 'Some tests failed', /home/steve/src/rust/src/libtest/lib.rs:243
```

Great! It showed that our first test passed, and that our second one failed. Let's make both tests pass:

```
fn div_by_three(num: int) -> bool {
    if num % 3 == 0 {
        true
    } else {
        false
    }
}

#[test]
fn test_div_by_three() {
    if div_by_three(1) {
        fail!("One is not three");
    }
}

#[test]
fn test_div_by_three_with_three() {
    if !div_by_three(3) {
        fail!("Three should be three");
    }
}
```

```
$ rustc --test fizzbuzz.rs && ./fizzbuzz
running 2 tests
test test_div_by_three_with_three ... ok
test test_div_by_three ... ok

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

Awesome! This shows off how elses work, as well. It's probably what you expected. Go ahead and try to refactor this into a one-liner.

Done? How'd you do? Here's mine:

```
fn div_by_three(num: int) -> bool {
    num % 3 == 0
}
```

Wait, whaaaat? Remember,the last thing in a function is a return in Rust, but there's one wrinkle: note there's no semicolon here. If you had one, you'd get:

```
$ rustc --test fizzbuzz.rs
fizzbuzz.rs:2:15: 2:16 note: consider removing this semicolon:
fizzbuzz.rs:2     num % 3 == 0;
                            ^
fizzbuzz.rs:1:1: 3:2 error: not all control paths return a value
fizzbuzz.rs:1 fn div_by_three(num: int) -> bool {
fizzbuzz.rs:2     num % 3 == 0;
fizzbuzz.rs:3 }
error: aborting due to previous error
```

Basically, ending an expression in Rust with a semicolon ignores the value of that expression. Another way to think about it is that the semicolon turns the expression into a statement, and statements don't have values. This is kinda weird. It becomes natural after some use, though. And Rust is even smart enough to tell us that it's probably a problem!

Okay, now try to TDD out the `div_by_five` and `div_by_fifteen` methods. They should work the same way, but this will let you get practice actually writing it out. Once you see this, you're ready to advance:

```
$ rustc --test fizzbuzz.rs && ./fizzbuzz

running 6 tests
test test_div_by_fifteen ... ok
test test_div_by_five_with_five ... ok
test test_div_by_five ... ok
test test_div_by_fifteen_with_fifteen ... ok
test test_div_by_three ... ok
test test_div_by_three_with_three ... ok

test result: ok. 6 passed; 0 failed; 0 ignored; 0 measured
```

Okay! Let's talk about the main program now. We've got the tools to build FizzBuzz, let's make it work. First thing we need to do is print out all the numbers from one to 100. It's easy!

```
    fn main() {
        for num in range(1i, 100) {
            println!("num");
        }
    }
```

Step one: print **something** 100 times. If you run this via `rustc fizzbuzz.rs && ./fizzbuzz` you should see `num` printed 100 times. Note that our tests didn't actually run. Not only are they not run, they're actually not even in the executable:

Compile with the test flag:

```
$ rustc --test fizzbuzz.rs
```

On Linux:

```
$ nm -C fizzbuzz | grep test
```

On OS X:

```
$ nm fizzbuzz | c++filt -p -i | grep test
```

Expected output:

```
0000000000403cd0 t test_div_by_five_with_five::_79fbef3fc431adf6::_00
0000000000403ac0 t test_div_by_three_with_three::_79fbef3fc431adf6::_00
0000000000403c10 t test_div_by_five_with_not_five::_79fbef3fc431adf6::_00
0000000000403ee0 t test_div_by_fifteen_with_fifteen::_79fbef3fc431adf6::_00
0000000000403a00 t test_div_by_three_with_not_three::_79fbef3fc431adf6::_00
0000000000403e20 t test_div_by_fifteen_with_not_fifteen::_79fbef3fc431adf6::_00
                 U test::test_main_static::_e5d562a4bc8c4dd6::_06
000000000040fea0 T __test::main::_79fbef3fc431adf6::_00
0000000000614890 D __test::tests::_7c31a8a9617a6a::_00
```

Compile without the test flag:

```
$ rustc fizzbuzz.rs
```

On Linux:

```
$ nm -C fizzbuzz | grep test
```

On OS X:

```
$ nm fizzbuzz | c++filt -p -i | grep test
```

Expected output:

```
$
```

Neat, huh? Rust is smart.

Anyway, `nm`: The `nm` program lists all the symbols in a binary executable or library. The `-C` option is important on linux, it "de-mangles" the symbol names. On OS X, `nm` provides no symbol de-mangling option, so the output must be piped to `c++filt`. Rust uses the same mangling scheme as C++, so it's compatible with all the existing tools. How it works isn't that important, though. It's cool low-level stuff if you're into that sort of thing.

Anywho, where were we? Oh, iteration:

```
fn main() {
    for num in range(1i, 100) {
        println!("{:d}", num);
    }
}
```

This uses string interpolation: the double curlies tell Rust where to place `num` in the string.

Anyway, now we have 1 to 99. We need 1 to 100.

```
fn main() {
    for num in range(1i, 101) {
        println!("{:d}", num);
    }
}
```

Now we can put the two together:

```
fn main() {
    for num in range(1i, 101) {
        let mut answer = "";

        if div_by_fifteen(num){
            answer = "FizzBuzz";
        }
        else if div_by_three(num) {
            answer = "Fizz";
        }
        else if div_by_five(num) {
            answer = "Buzz";
        }
        else {
            answer = "";
        };

        println!("{:s}", answer);
    }
}
```

Uhhhh `let mut`? `let` is the way that we make a local variable. `mut` means we plan to mutate that variable: yes, variables are immutable by default.

Also, `:s` is the format string for a… string.

We can shorten this up a bit with this syntax:

```
fn main() {
    for num in range(1i, 101) {
        let mut answer =
            if div_by_fifteen(num){
                "FizzBuzz"
            }
            else if div_by_three(num) {
                "Fizz"
            }
            else if div_by_five(num) {
                "Buzz"
            }
            else {
                ""
            };

        println!("{:s}", answer);
    }
}
```

We've made the `if` assign the value to answer. Note that we had to remove the semicolons again; that lets the expression give its value to `answer.` Note that this _also_ makes answer immutable, so we can remove the `mut`:

```
fn main() {
    for num in range(1i, 101) {
        let answer =
            if div_by_fifteen(num){
                "FizzBuzz"
            }
            else if div_by_three(num) {
                "Fizz"
            }
            else if div_by_five(num) {
                "Buzz"
            }
            else {
                ""
            };

        println!("{:s}", answer);
    }
}
```

Not too shabby! I love eliminating mutable state.

Of course, this version gives us lots of empty lines, so what we actually want is:

```
fn main() {
    for num in range(1i, 101) {
        let answer =
            if div_by_fifteen(num){
                "FizzBuzz".to_str()
            }
            else if div_by_three(num) {
                "Fizz".to_str()
            }
            else if div_by_five(num) {
                "Buzz".to_str()
            }
            else {
                num.to_str()
            };

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

Why the "`to_str()`"s? There are two types of Strings in Rust: `Str`, which is a heap allocated string with dynamic length, and `&str`, which is a borrowed, immutable view into a string. The literal is of type `&str`, but we want a `Str`. `to_str()` turns a `&str` into a `String`.

Before, we could get away with a `&str`, because they all had the same type. But since we've added an arm with an `int`, we need to make them all the same type, and there's no way to convert an `int` into a `&str`.

Because the `if` returns a value, we could also do something like this:

```
fn main() {
    for num in range(1i, 101) {
        println!("{:s}",
            if div_by_fifteen(num) { "FizzBuzz".to_str() }
            else if div_by_three(num) { "Fizz".to_str() }
            else if div_by_five(num) { "Buzz".to_str() }
            else { num.to_str() }
        );
    }
}
```

It's more compact, and removes the intermediate variable all together.

We can do one other thing too: this whole `if/fail!` thing so common in tests seems too complex. Why do we have to write if over and over and over again? Meet `assert!`:

```
#[test]
fn test_div_by_fifteen_with_fifteen() {
    assert!(div_by_fifteen(15))
}
```

This will fail if it gets false, and pass if it gets true. Simple! You can also give it a message to be printed when the assertion fails, mostly useful when you are using `assert!` to test for preconditions and such:

```
fn main() {
    assert!(1 == 0, "1 does not equal 0!");
}
```

Try running it.

Anyway, awesome! We've conquered FizzBuzz.

Tasks in Rust

One of the things that Rust is super good at is concurrency. In order to understand Rust's strengths, you have to understand its approach to concurrency, and then its approach to memory.

Tasks

The fundamental unit of computation in Rust is called a 'task.' Tasks are like threads, but you can choose the low-level details of how they operate. Rust now supports both 1:1 scheduled and N:M scheduled threads. Rust uses 1:1 threads by default. The details of what *exactly* that means are out of the scope of this tutorial, but the Wikipedia page has a good overview.

Here's some code that prints "Hello" 500 times:

```
fn main() {
    for num in range(0u, 500) {
        println!("Hello");
    }
}
```

You may remember this from earlier. This loops 500 times, printing "Hello." Now let's make it roflscale with tasks:

```
fn main() {
    for num in range(0u, 500) {
        spawn(proc() {
            println!("Hello");
        });
    }
}
```

That's it! We spin up 500 tasks that print stuff. If you inspect your output, you can tell it's working:

```
Hello
HelloHello

Hello
```

Ha! Printing to the screen is obviously something that tasks can step over each other with (if you're curious, it's because it is printing the string and the newline separately. Sometimes, another task gets to print its string before this task prints its newline). But the vast majority of things aren't like that. Let's take a look at the type signature of `spawn`:

```
fn spawn(f: proc())
```

Spawn is a function that takes a proc: a closure that can only be run once. This means that Rust can do what it wants, moving the closure to another task, or other optimizations. The details aren't particularly important at this stage, and Rust will be undergoing some reform with regards to closures soon, so just think of it as a closure, and that's good enough.

Pipes, Channels, and Ports

If our tasks are 100% isolated, they wouldn't be that useful: we need some kind of communication between tasks in order to get back useful results. We can communicate between tasks with pipes. Pipes have two ends: a channel that sends info down the pipe, and a port that receives info. If you've used these concepts in other languages, Rust's are similar, except that Rust's are explicitly typed. Some implementations of this pattern in other languages do not make this distinction. Otherwise, they're very similar.

Here's an example of a task that sends us back a 10:

```
fn main() {
    let (chan, port) = channel();

    spawn(proc() {
        chan.send(10u);
    });

    println!("{:s}", port.recv().to_str());
}
```

The `channel` function, imported by the prelude, creates both sides of this pipe. You can imagine that instead of sending 10, we might be doing some sort of complex calculation. It could be doing that work in the background while we did more important things.

What about that `chan.send` bit? Well, the task captures the `chan` variable we set up before, so it's just matter of using it. This is similar to Ruby's blocks:

```
foo = 10
2.times do
  puts foo
end
```

This is really only one-way transit, though: what if we want to communicate back and forth? Setting up two ports and channels each time would be pretty annoying, so we have some standard library code for this.

We make a function that just loops forever, gets an `int` off of the port, and sends the number plus 1 back down the channel. In the main function, we make a channel, send one end to a new task, and then send it a `22`, and print out the result. Because this task is running in the background, we can send it bunches of values:

```
use std::comm::{channel, Sender, Receiver};

fn plus_one(sender: &Sender<int>, receiver: &Receiver<int>) {
    let mut value: int;
    loop {
        value = receiver.recv();
        sender.send(value + 1);
        if value == 0 { break; }
    }
}

fn main () {
    let (fromParentSender, fromParentReceiver) = channel();
    let (fromChildSender, fromChildReceiver) = channel();

    spawn(proc() {
        plus_one(&fromChildSender, &fromParentReceiver);
    });

    fromParentSender.send(22);
    fromParentSender.send(23);
    fromParentSender.send(24);
    fromParentSender.send(25);

    for _ in range(0u, 4) {
        let answer = fromChildReceiver.recv();
        println!("{:s}", answer.to_str());
    }
}
```

The `use` statement imports other modules. In this case, there's a `std::comm` module that we'll use parts of.

Pretty simple. Our task is always waiting for work. If you run this, you'll get some weird output at the end:

```
$ rustc tasks.rs && ./tasks
23
24
25
```

```
26
task '<unnamed>' failed at 'receiving on a closed channel', /home/steveklabnik/src/rust/src/
```

`task failed at 'receiving on closed channel'`. Basically, we quit the program without closing our child task, and so it died when our main task (the one running `main`) died. By default, Rust tasks are bidirectionally linked, which means if one task fails, all of its children and parents fail too. We can fix this for now by telling our child to die:

```rust
use std::comm::{channel, Sender, Receiver};

fn plus_one(sender: &Sender<int>, receiver: &Receiver<int>) {
    let mut value: int;
    loop {
        value = receiver.recv();
        sender.send(value + 1);
        if value == 0 { break; }
    }
}

fn main () {
    let (fromParentSender, fromParentReceiver) = channel();
    let (fromChildSender, fromChildReceiver) = channel();

    spawn(proc() {
        plus_one(&fromChildSender, &fromParentReceiver);
    });

    fromParentSender.send(22);
    fromParentSender.send(23);
    fromParentSender.send(24);
    fromParentSender.send(24);

    fromParentSender.send(0);

    for _ in range(0i, 4) {
        let answer = fromChildReceiver.recv();
        println!("{:s}", answer.to_str());
    }
}
```

Now when we send a zero, our child task terminates. If you run this, you'll get no errors at the end. We can also change our failure mode. Rust also provides unidirectional and unlinked failure modes as well, but I don't want to talk about them right now. This would give you patterns like "Spin up a management task that is bidirectionally linked to main, but have it spin up children who are unlinked." Neato.

Rust tasks are so lightweight that you can conceivably spin up a ton of tasks, maybe even one per entity in your system. Servo is a prototype browser rendering engine from Mozilla, and it spins up a **ton** of tasks. Parallel rendering, parsing, downloading, everything.

I'm imagining that most production Rust programs will eventually have a main that spins up some sort of global task setup, and all the work gets done inside these tasks that communicate with each other. Like, for a video game:

```
fn main() {

    spawn(proc() {
        player_handler();
    });

    spawn(proc() {
        world_handler();
    });

    spawn(proc() {
        rendering_handler();
    });

    spawn(proc() {
        io_handler();
    });
}
```

. . . with the associated channels, of course. This feels very Actor-y to me. I like it.

Pointers, and ownership, oh my!

Since you program in Ruby, you probably don't know about pointers, nor care. If you're going to work in a language like Rust, though, you gotta know about them. So let's talk about the concept real quick, then discuss how Rust handles pointers.

Pointer recap

When you create a variable, you're really giving a name to a chunk of memory somewhere. We'll use C syntax for these examples:

```
int i = 5; int j = 6;

  location   value
  ---------- -------
  0x000000   5
  0x000001   6
```

This is of course slightly simplified. Anyway, we can introduce indirection by making a pointer:

```
int i = 5; int j = 6; int *pi = &i;

  location   value
  ---------- ----------
  0x000000   5
  0x000001   6
  0x000002   0x000000
```

`pi` has a pointer to another memory location. We can access the value of the thing that `pi` points at by using the `*`:

```
printf("The value of the thing pi points to is: %d\n", *pi);
```

The `*` dereferences the pointer, and gives us the value. Simple!

Here's the issue: you have no idea if the data that is being pointed to is good. What do you think this code does?:

```
int *pi;
printf("The value of the thing pi points to is: %d\n", *pi);
```

Who knows!?!? Probably something bad, but **certainly** not something expected. Rust introduces two different kinds of pointers: 'owned' and 'borrowed.' They indicate different levels of access, so that you know that different people aren't messing with the things that are being pointed to. Imagine we spun up ten tasks, passed the same pointer to all of them, and let them go wild. We'd have no idea what was going on.

Owned Pointer

An owned pointer tells Rust that you own a reference to something. We can create one with the `box` keyword:

```
fn main() {
    let x = box 10i;
    println!("{:d}", *x);
}
```

You can't make another owned pointer to this value:

```
fn main() {
    let x = box 10i;
    let y = x;
    println!("{:d}", *x);
}
```

This yields:

```
$ rustc owned.rs && ./owned
owned.rs:4:22: 4:24 error: use of partially moved value: `*x`
owned.rs:4     println!("{:d}", *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!
owned.rs:4:5: 4:26 note: expansion site
owned.rs:3:9: 3:10 note: `x` moved here because it has type `Box<int>`, which is moved by de
owned.rs:3     let y = x;
                      ^
error: aborting due to previous error
```

It tells us that we moved the value of `x` to `y` and points out where the move happens. Neat. We can make a copy:

```
fn main() {
    let x = box 10i;
    let y = x.clone();
    println!("{:d}", *x);
}
```

This will work, though it will tell us that `y` was never used. And they point at two different copies of 10, not the same one.

That said, you generally don't need to use an owned pointer. You generally need them for recursive data structures, or when you have a *huge* chunk of data that you're passing around between many functions.

Instead, use a borrowed pointer.

Borrowed Pointers

Enter borrowed pointers:

```
fn plus_one(x: &int) -> int {
    *x + 1
}

fn main() {
    let y = box 10i;

    println!("{:d}", plus_one(y));
}
```

Borrowed pointers use an `&`, as you can see. They don't change any ownership semantics. They do let you write functions that take any other kind of pointer, without caring about those details. The compiler makes sure that all borrowed pointers do not outlive the thing they point to, which means you don't have to worry about use-after-free or any of the other hairy pointer issues in C.

Borrowed pointers can get a lot more complex, but this is the gist of them. Always use borrowed pointers when you can, they should be your go-to solution for all your pointer needs.

Pointer strategy

Basically, idiomatic Rust code will... not use pointers at all, and just use values. If you're writing a function that needs to take a pointer, use borrowed pointers, rather than being specific.

There are some more complex heap-allocated types in Rust, but they're outside the scope of this introduction.

Structs and Methods

I'd like to talk about structs and methods, so let's build a fun little project: DwemthysArray. One of _why's sillier examples, we make an array of monsters, and then fight them. We won't be building the Array _exactly_, but something like it.

Structs

Structs are ways of packaging up multiple values into one:

```
struct Monster {
    health: int,
    attack: int
}

fn main() {
    let m = Monster { health: 10, attack: 20 };

    println!("{:s}", m.health.to_str());
    println!("{:s}", m.attack.to_str());
}
```

This gives:

```
$ rustc dwemthysarray.rs && ./dwemthysarray
10
20
```

Seems simple enough!

Methods

Methods are basically functions that take a first argument named `self`. Python people who are reading will be high fiving each other in droves. Let's add a method for our `Monster` s:

```
struct Monster {
    health: int,
    attack: int
}

impl Monster {
    fn attack(&self) {
        println!("The monster attacks for {:d} damage.", self.attack);
    }
}

fn main() {
    let m = Monster { health: 10, attack: 20 };

    m.attack();
}
```

This gives:

```
The monster attacks for 20 damage.
```

Methods will want to take a borrowed pointer. We don't care what the ownership semantics are. That's the `&self`, if you forgot.

You can define associated functions (class methods, in Ruby, static methods, in Java) as well:

```
struct Monster {
    health: int,
    attack: int
}

impl Monster {
    fn attack(&self) {
        println!("The monster attacks for {:d} damage.", self.attack);
    }

    fn count() {
        println!("There are a bunch of monsters out tonight.");
    }
}

fn main() {
    let m = Monster { health: 10, attack: 20 };

    m.attack();
    Monster::count();
}
```

Constructors are a good reason to use associated functions:

```
struct Monster {
    health: int,
    attack: int
}

impl Monster {
    fn new(health: int, attack: int) -> Monster {
        Monster { health:health, attack:attack }
    }

    fn attack(&self) {
        println!("The monster attacks for {:d} damage.", self.attack);
    }
    fn count() {
        println!("There are a bunch of monsters out tonight.");
    }

}

fn main() {
    Monster::new(20, 40).attack();
}
```

Note the lack of a semicolon inside `new`, so it's acting as an expression. `new` is just a function that creates a new `Monster` struct and returns it. This gives:

```
The monster attacks for 40 damage.
```

as you'd expect.

Enums

What if we want to define a few different types of things? In other languages, we'd use inheritance. In Rust, it seems like Enums are a better idea. Here's an enum:

```
enum Monster {
    ScubaArgentine(int, int, int, int),
    IndustrialRaverMonkey(int, int, int, int)
}


impl Monster {
    fn attack(&self) {
        match *self {
            ScubaArgentine(l, s, c, w) => println!("The monster attacks for {:d} damage.", w
            IndustrialRaverMonkey(l, s, c, w) => println!("The monster attacks for {:d} dama
        }
    }
}

fn main() {
    let irm = IndustrialRaverMonkey(46, 35, 91, 2);
    irm.attack();
}
```

Okay, few new things here: We can see that there's some duplication here. Obviously this isn't the best way to do it, but I wanted to try this out before we got to the better implementation. We make an `Enum` that defines two different things, and then we use this `match` expression to "destructure" them and get at their... well, members, sorta.

If you haven't used pattern matching in another language, you're missing out. It's awesome. Here's a simpler match expression:

```
fn message(i: int) {
    match i {
        1 => println!("ONE!"),
        2 => println!("Two is a prime."),
        3 => println!("THREE!"),
        _ => println!("no idea what that is, boss")
    }
}

fn main() {
    message(1);
    message(2);
    message(3);
}
```

Does that make sense? It's sorta like a `case` statement, but it's more powerful. If we leave off the `_` case, Rust will complain:

```
$ rustc match.rs && ./match
match.rs:2:4: 6:5 error: non-exhaustive patterns: `_` not covered
match.rs:2     match i {
match.rs:3         1 => println("ONE!"),
match.rs:4         2 => println("Two is a prime."),
match.rs:5         3 => println("THREE!"),
match.rs:6     }
error: aborting due to previous error
```

Neat. The reason we didn't need to specify a `_` case in our monster code is that because we were matching an `enum`, rust knew we had covered all the possible cases. But since we're matching an `int`, what would happen if we called, say, `message(349)`? Rust makes us specify a default case with `_` so that it knows exactly what we want. You also have to put the `_` case last, after any other cases, because Rust looks at them top-to-bottom, and will complain about unreachable patterns that come after `_`.

The cool thing is that when pattern matching on a struct, the `match` can destructure it:

```
match p {
    Point{x:x, y:y} => println!("X: {:d}, Y: {:d}", x, y)
}
```

We name the two fields of a `Point` `x` and `y`, and those names are valid within the match expression. Match is a lot more powerful (they can express ranges, options, and even variable binding), but this is its common use.

Let's build monsters!

Before we build some monsters, let's look at the Right Way to implement them. We can do this with Traits, but that's the next chapter.

Vectors

Before getting into generic functions that could handle multiple kinds of Monster, let's first talk about a format that you end up using them with often: Vectors. Vectors are the 'array' in Dwemthy's Array: they're lists of things, but unlike in Ruby, the elements must all be of the same type. You can have any of the three kinds of pointers to vectors, and you'll sometimes hear a borrowed pointer to a vector called a 'slice.'

Examples

See if this looks familiar:

```
    fn main() {
        let your_favorite_numbers = vec!(1i, 2i, 3i);
        let my_favorite_numbers = vec!(4i, 5i, 6i);

        let our_favorite_numbers = your_favorite_numbers + my_favorite_numbers;

        println!("The third favorite number is {:d}.", *our_favorite_numbers.get(2))
    }
```

Seems like business as usual: `+` adds two vectors, `get()` does an indexing operation.

Mutability inheritance

You can mutate vectors if you make them so:

```
fn main() {
    let mut another_vector = vec!(4i);
    another_vector.push_all([1, 2, 3]);

    println!("The second number is {:d}.", *another_vector.get(1))
}
```

Of course, changing an element of a vector doesn't make sense:

```
fn main() {
    let a_vector = vec!(1i, 2i, 3i);
    a_vector.get(0) = 5; // error: illegal left-hand side expression

    println!("The first number is {:d}.", *a_vector.get(0))
}
```

But you can move it to a mutable one and then change it:

```
fn main() {
    let a_vector = vec!(1i, 2i, 3i);
    let mut mut_vector = a_vector;
    *mut_vector.get_mut(0) = 5;

    println!("The first number is {:d}.", *mut_vector.get(0))
}
```

When you make an immutable vector mutable, it's called 'thawing' the vector, and the opposite is 'freezing' a vector.

That's it! Vectors are pretty simple.

Traits and Generics

Now that we understand a type that's sorta generic, vectors, we can talk about how generic functions work. Then, we can use traits to make functions that work on generic monsters.

Writing functions that work with vectors

Because you're still getting used to Rust code, let's implement two versions of a method that print everything in a vector, and then refactor that into the generic form.

Let's do an exercise. You have this code:

```
fn main() {
    let vec = [1i, 2i,3i];

    print_vec(vec);
}
```

Implement `print_vec` so that it puts out `1 2 3` with newlines between them. Hint: You can write 'I want an array of ints' with `&[int]`. Remember how functions can often use borrowed pointers?

I'll wait.

Done? I got this:

```
fn print_vec(v: &[int]) {
    for i in v.iter() {
        println!("{:d}", *i)
    }
}

fn main() {
    let vec = [1i ,2i ,3i];

    print_vec(vec);
}
```

Pretty straightforward. We take a slice (remember, 'borrowed vector' == 'slice') of ints, get a borrowed pointer to each of them, and print them out.

Round two: Implement this:

```
fn main() {
    let vec = [1i ,2i ,3i];

    print_vec(vec);

    let str_vec = ["hey", "there", "yo"];

    print_vec_str(str_vec);
}
```

You'll often be seeing owned pointers with strings. Go ahead. You can do it!

I got this:

```
fn print_vec(v: &[int]) {
    for i in v.iter() {
        println!("{:d}", *i)
    }
}

fn print_vec_str(v: &[&str]) {
    for i in v.iter() {
        println!("{:s}", *i)
    }
}

fn main() {
    let vec = [1i ,2i ,3i];

    print_vec(vec);

    let str_vec = ["hey", "there", "yo"];

    print_vec_str(str_vec);
}
```

You'll notice we had to declare what type of `str` we had. See, strings are actually implemented as vectors of characters (encoded in UTF-8), so while they are sorta a type, you can't have just `str` as a type. You gotta say `&str`. As I mentioned before, there is also a mutable, heap-allocated string type, `Str`.

Okay, obviously, this situation sucks! What can we do?

We can use generics!

```
    fn print_vec<T>(v: &[T]) {
        for i in v.iter() {
            println!("{}", i)
        }
    }

    fn main() {
        let vec = [1i, 2i, 3i];

        print_vec(vec);

        let str_vec = ["hey", "there", "yo"];
        print_vec(str_vec);
    }
```

This won't compile, but it is closer. Let's examine that signature more closely.

- `<T>` says that we're going to be making this function polymorphic over the type T.
- We then use it later to say we take a borrowed pointer of a vector of `T` s, `&[T]`

If you try to compile this, you'll get an error:

```
$ rustc traits.rs && ./traits
fizzbuzz.rs:5:28: 5:29 error: failed to find an implementation of trait std::fmt::Show for
fizzbuzz.rs:5             println!("{}", i)
```

This is a problem. Our generic type T does not have any restrictions on what kind of thing it is, which means we can't guarantee that we'll get something that has the ability to be displayed.

For that, we need Traits.

Traits

This **will** work:

```
fn print_vec<T: std::fmt::Show>(v: &[T]) {
    for i in v.iter() {
        println!("{}", i)
    }
}

fn main() {
    let vec = [1i ,2i ,3i];

    print_vec(vec);

    let str_vec = ["hey", "there", "yo"];

    print_vec(str_vec);
}
```

The `<T: std::fmt::Show>` says: “We take any type `T` that implements the `Show` trait.

Traits are sort of like ‘static duck typing’ or ‘structural typing.’ We get away with this in Ruby by just trusting the code we write, and for most of it, it just works out. Think about this:

```
def print_each(arr)
  arr.each do |i|
    puts i
  end
end
```

We trust that this will always work, because `Object` implements `#to_s`. But if we had this:

```
def print_each(arr)
  arr.each do |i|
    puts i + 1
  end
end
```

We have an implicit type here: `arr` must contain things that `respond_to?(:+)`. In many ways, Rust is sorta like:

```
def print_each(arr)
  assert arr.respond_to?(:+)

  arr.each do |i|
    puts i + 1
  end
end
```

But it happens at compile time, not run time.

Now, I’ve never written code where I felt the need to check for a `NoMethodError` or `TypeError`, as you’d get in Ruby:

```
irb(main):007:0> print_each(["a","b","c"])
TypeError: can't convert Fixnum into String
  from (irb):3:in `+'
  from (irb):3:in `block in print_each'
  from (irb):2:in `each'
  from (irb):2:in `print_each'
  from (irb):7
  from /usr/local/ruby-1.9.3-p327/bin/irb:12:in `<main>'
```

But I think that safety is the wrong way to look at this kind of static typing. The right way to look at it is that by giving the compiler more information about our code, it can make certain optimizations. Check this out:

```
$ cat traits.rs

    fn print_vec<T: std::fmt::Show>(v: &[T]) {
        for i in v.iter() {
            println!("{}", i)
        }
    }

    fn main() {
        let vec = [1i, 2i, 3i];

        print_vec(vec);

        let str_vec = ["hey", "there", "yo"];

        print_vec(str_vec);
    }

$ rustc traits.rs && ./traits
1
2
3
hey
there
yo

$ nm -C traits | grep vec
0000000000401500 t print_vec_2912::_85e5a3bc2d3e1a83::_00
0000000000401ee0 t print_vec_2912::anon::expr_fn_2970
0000000000404cd0 t print_vec_3218::_f1e1b4437dbb28a::_00
0000000000405480 t print_vec_3218::anon::expr_fn_3252
0000000000402c50 t vec::__extensions__::reserve_3030::_de1a9d6344b57ab::_00
0000000000402d70 t vec::__extensions__::capacity_3032::_824484774e7757::_00
0000000000404b50 t
vec::__extensions__::push_fast_3194::_5cf6fa3bfa6090d7::_00
0000000000404ae0 t
vec::__extensions__::reserve_at_least_3192::_de1a9d6344b57ab::_00
0000000000404840 t
vec::__extensions__::reserve_no_inline_3182::_24c451fdab89623e::_00
0000000000401c50 t vec::__extensions__::len_2959::_824484774e7757::_00
0000000000401e80 t vec::__extensions__::len_2959::anon::expr_fn_2968
```

```
00000000004048b0 t vec::__extensions__::len_3185::_824484774e7757::_00
0000000000404a80 t vec::__extensions__::len_3185::anon::expr_fn_3190
00000000004051f0 t vec::__extensions__::len_3243::_824484774e7757::_00
0000000000405420 t vec::__extensions__::len_3243::anon::expr_fn_3250
0000000000401a50 t vec::__extensions__::iter_2947::_d7a5bdd54e5e6f77::_00
00000000004050a0 t vec::__extensions__::iter_3237::_55446721964a82e1::_00
0000000000401680 t vec::__extensions__::next_2919::_5079d793a0f371c9::_00
0000000000404e50 t vec::__extensions__::next_3224::_b423b136d356fe1d::_00
0000000000404790 t vec::__extensions__::push_3179::_a91dd4803fb62a::_00
0000000000401d00 t vec::as_imm_buf_2961::_caa46d7965b990b9::_00
0000000000404970 t vec::as_imm_buf_3187::_62a416e4b98acea8::_00
00000000004052a0 t vec::as_imm_buf_3245::_cb6b3bad8005286::_00
0000000000401b30 t vec::raw::to_ptr_2950::_1df29a3554bbd95b::_00
0000000000405180 t vec::raw::to_ptr_3240::_8c11f86a3948f562::_00
                 U
                 vec::rustrt::vec_reserve_shared_actual::_c688b9b8fd5bf21::_07


$ mvim traits.rs
....editing...
$ cat traits.rs

    fn print_vec<T: std::fmt::Show>(v: &[T]) {
        for i in v.iter() {
            println!("{}", i)
        }
    }

    fn main() {
        let vec = [1,2,3];

        print_vec(vec);
    }

$ rustc traits.rs && ./traits

$ nm -C traits | grep vec
00000000004012d0 t print_vec_2908::_85e5a3bc2d3e1a83::_00
0000000000401cb0 t print_vec_2908::anon::expr_fn_2966
0000000000402a20 t vec::__extensions__::reserve_3026::_de1a9d6344b57ab::_00
0000000000402b40 t vec::__extensions__::capacity_3028::_824484774e7757::_00
0000000000404920 t
vec::__extensions__::push_fast_3190::_5cf6fa3bfa6090d7::_00
00000000004048b0 t
vec::__extensions__::reserve_at_least_3188::_de1a9d6344b57ab::_00
```

```
0000000000404610 t
vec::__extensions__::reserve_no_inline_3178::_24c451fdab89623e::_00
0000000000401a20 t vec::__extensions__::len_2955::_824484774e7757::_00
0000000000401c50 t vec::__extensions__::len_2955::anon::expr_fn_2964
0000000000404680 t vec::__extensions__::len_3181::_824484774e7757::_00
0000000000404850 t vec::__extensions__::len_3181::anon::expr_fn_3186
0000000000401820 t vec::__extensions__::iter_2943::_d7a5bdd54e5e6f77::_00
0000000000401450 t vec::__extensions__::next_2915::_5079d793a0f371c9::_00
0000000000404560 t vec::__extensions__::push_3175::_a91dd4803fb62a::_00
0000000000401ad0 t vec::as_imm_buf_2957::_caa46d7965b990b9::_00
0000000000404740 t vec::as_imm_buf_3183::_62a416e4b98acea8::_00
0000000000401900 t vec::raw::to_ptr_2946::_1df29a3554bbd95b::_00
                 U
                 vec::rustrt::vec_reserve_shared_actual::_c688b9b8fd5bf21::_07
```

Okay. So the first time we have our code, we have two calls to `print_vec`, one for a vector of strings and one for a vector of ints. The call to `nm`…

Oh wait, I mentioned `nm` before, but let me tell you some more about it now!

A diversion about nm

Here's what my manpage says:

```
$ man nm

NAME
       nm - display name list (symbol table)

SYNOPSIS
       nm  [  -agnoprumxjlfPA  [  s segname sectname ]] [ - ] [ -t format ] [[
       -arch arch_flag ]...] [ file ... ]

DESCRIPTION
       Nm displays the name list (symbol table) of each  object  file  in  the
       argument list.
```

Cool! You've never had to think about symbol tables before, so let's talk about them.

When your compiler compiles something, you get an 'object file' out of it. This is the binary that you run: `rustc fizzbuzz.rs` produces `fizzbuzz`. This object file will contain a list of `symbols` and where they exist in memory. This matters when we want to write two bits of code that work together: If my library exposes a function called `my_function`, and you want to use it, the compiler needs to know where to find `my_function` in my library's code. The compiler 'mangles' the names to fit its own scheme. This is called an "ABI", or application binary interface. Have you ever seen this:

```
/Users/Steve/.rvm/rubies/ruby-1.9.3-p286/lib/ruby/1.9.1
```

And wondered why that 1.9.1 is there? That's because Ruby 1.9.3 and Ruby 1.9.1 both share the same ABI, so gems that are linked against 1.9.1 can also be used with 1.9.3. They use the same scheme to generate symbols.

Anyway, `nm` can show us this information. The first column is the location in memory, the second is the (mangled) name:

```
0000000100001bb8 S _rust_abi_version
```

That's a fun, recursive symbol ;) Anyway, we can examine what symbols Rust exports to see some information about our executable, that's my intention with `nm` in this case.

Back to our regularly scheduled investigation

Here's the important part of the two outputs of nm:

```
0000000000401500 t print_vec_2912::_85e5a3bc2d3e1a83::_00
0000000000401ee0 t print_vec_2912::anon::expr_fn_2970
0000000000404cd0 t print_vec_3218::_f1e1b4437dbb28a::_00
0000000000405480 t print_vec_3218::anon::expr_fn_3252
```

and:

```
00000000004012d0 t print_vec_2908::_85e5a3bc2d3e1a83::_00
0000000000401cb0 t print_vec_2908::anon::expr_fn_2966
```

See how they both have `print_vec`? These are the functions we made. And without even knowing what's happening, you can see the difference: in the version of our code where we call `print_vec` on strings and ints, we have two versions of the function, and on the version where we just call it on ints, we have one version.

Neat! We get specialized versions, but only specialized for the types we actually use. No generating code that's useless. This process is called 'monomorphization,' which basically means we take something that can work with things of different types and change it (morph) into specialized (mono) versions. To simplify, the compiler takes this code:

```
fn print_vec<T: std::fmt::Show>(v: &[T]) {
    for i in v.iter() {
        println!("{}", i);
    }
}

fn main() {
    let vec = [1i ,2i ,3i];

    print_vec(vec);

    let str_vec = ["hey", "there", "yo"];

    print_vec(str_vec);
}
```

And turns it into:

```
fn print_vec_str(v: &[&str]) {
    for i in v.iter() {
        println!("{}", i);
    }
}

fn print_vec_int(v: &[int]) {
    for i in v.iter() {
        println!("{}", i);
    }
}

fn main() {
    let vec = [1i ,2i ,3i];

    print_vec_int(vec);

    let str_vec = ["hey", "there", "yo"];

    print_vec_str(str_vec);
}
```

Complete with changing the calls at each call site to call the special version of the function. We call this 'static dispatch,' as opposed to the 'dynamic dispatch' that'd happen at runtime.

(I am fudging a bit here with the `println!` macro line, but it's the correct mental model. `{}` doesn't actually work on `int`.)

These are the kinds of optimizations that we get with static typing. Neat! I will say that there are efforts to bring this kind of optimization into dynamically typed languages as well, through analyzing the call site. So, for example:

```
def foo(arg)
  puts arg
end
```

If we call `foo` with a `String arg` a bunch of times in a row, the interpreter will JIT compile a version of `foo` specialized for `Strings`, and then replace the call site with something like:

```
if arg.kind_of? String
  __super_optimized_foo_string(arg)
else
  foo(arg)
end
```

This would give you the same benefit, without the human typing. Not just that, but a sufficiently smart runtime would be able to actually determine more complex situations that a person may not. And, maybe after, say, 1000 calls with a String, just remove the check entirely.

Anyway.

Making our own Traits

We want all of our monsters to implement `attack`. So let's make `Monster` a Trait. The syntax looks like this:

```
trait Monster {
    fn attack(&self);
}
```

This says that the `Monster` trait guarantees we have one method available on any type that implements the trait, `attack`. Here's how we make one:

```
trait Monster {
    fn attack(&self);
}

struct IndustrialRaverMonkey {
    strength: int
}

impl Monster for IndustrialRaverMonkey {
    fn attack(&self) {
        println!("The monkey attacks for {:d}.", self.strength)
    }
}

fn main() {
    let monkey = IndustrialRaverMonkey {strength:35};

    monkey.attack();
}
```

Now we're cooking with gas! Remember our old implementation?:

```
impl Monster {
    fn attack(&self) {
        match *self {
            ScubaArgentine(l, s, c, w) => println!("The monster attacks for {:d} damage
            IndustrialRaverMonkey(l, s, c, w) => println!("The monster attacks for {:d}
        }
    }
}
```

Ugh. This is way better. No destructuring on types. We can write an implementation for absolutely anything:

```
trait Monster {
    fn attack(&self);
}

struct IndustrialRaverMonkey {
    strength: int
}

impl Monster for IndustrialRaverMonkey {
    fn attack(&self) {
        println!("The monkey attacks for {:d}.", self.strength)
    }
}

impl Monster for int {
    fn attack(&self) {
        println!("The int attacks for {:d}.", *self)
    }
}

fn main() {
    let monkey = IndustrialRaverMonkey {strength:35};
    monkey.attack();

    let i = 10;
    i.attack();
}
```

Heh. Check it:

```
$ rustc dwemthy.rs && ./dwemthy
The monkey attacks for 35.
The int attacks for 10.
```

Amusing.

Okay, exercise: Make six different monsters, and create a vector with all of them in it. Then write a method that takes the vector, and prints out all of the monsters and their stats.

I'll wait. It took me a little while to write this: this is the hardest part of the book so far. Work through it; it'll be painful. Don't be afraid to ask for help. I had to ask the rust IRC for help once while doing it. They're friendly, don't worry.

Done? Here's mine:

```
trait Monster {
    fn attack(&self);
    fn new() -> Self;
}

struct IndustrialRaverMonkey {
    life: int,
    strength: int,
    charisma: int,
    weapon: int,
}

struct DwarvenAngel {
    life: int,
    strength: int,
    charisma: int,
    weapon: int,
}

struct AssistantViceTentacleAndOmbudsman {
    life: int,
    strength: int,
    charisma: int,
    weapon: int,
}

struct TeethDeer {
    life: int,
    strength: int,
    charisma: int,
    weapon: int,
}

struct IntrepidDecomposedCyclist {
    life: int,
    strength: int,
    charisma: int,
    weapon: int,
}

struct Dragon {
    life: int,
    strength: int,
    charisma: int,
    weapon: int,
}

impl Monster for IndustrialRaverMonkey {
    fn attack(&self) {
        println!("The monkey attacks for {:d}.", self.strength)
    }

    fn new() -> IndustrialRaverMonkey {
        IndustrialRaverMonkey {life: 46, strength: 35, charisma: 91, weapon: 2}
    }
}

impl Monster for DwarvenAngel {
    fn attack(&self) {
        println!("The angel attacks for {:d}.", self.strength)
    }
    fn new() -> DwarvenAngel {
        DwarvenAngel {life: 540, strength: 6, charisma: 144, weapon: 50}
    }
}

impl Monster for AssistantViceTentacleAndOmbudsman {
    fn attack(&self) {
        println!("The tentacle attacks for {:d}.", self.strength)
    }
    fn new() -> AssistantViceTentacleAndOmbudsman {
        AssistantViceTentacleAndOmbudsman {life: 320, strength: 6, charisma: 144, weapon
    }
}

impl Monster for TeethDeer {
    fn attack(&self) {
        println!("The deer attacks for {:d}.", self.strength)
    }
    fn new() -> TeethDeer {
        TeethDeer {life: 655, strength: 192, charisma: 19, weapon: 109}
    }
}

impl Monster for IntrepidDecomposedCyclist {
    fn attack(&self) {
        println!("The cyclist attacks for {:d}.", self.strength)
    }
    fn new() -> IntrepidDecomposedCyclist {
        IntrepidDecomposedCyclist {life: 901, strength: 560, charisma: 422, weapon: 105
    }
}

impl Monster for Dragon {
    fn attack(&self) {
        println!("The dragon attacks for {:d}.", self.strength)
    }
    fn new() -> Dragon {
        Dragon {life: 1340, strength: 451, charisma: 1020, weapon: 939}
    }
}

fn monsters_attack(monsters: &[&Monster]) {
    for monster in monsters.iter() {
        monster.attack();
    }
}
```

```
    fn main() {
        let monkey: &IndustrialRaverMonkey               = &Monster::new();
        let angel: &DwarvenAngel                         = &Monster::new();
        let tentacle: &AssistantViceTentacleAndOmbudsman = &Monster::new();
        let deer: &TeethDeer                             = &Monster::new();
        let cyclist: &IntrepidDecomposedCyclist          = &Monster::new();
        let dragon: &Dragon                              = &Monster::new();

        let dwemthys_vector: &[&Monster] = [monkey as &Monster, angel as &Monster, tentacle

        monsters_attack(dwemthys_vector);
    }
```

Congrats! You've mastered Traits. They're pretty awesome, right?

Standard Input

If we want to make this little text-based game, we need to figure out how to get text off of standard in. So let's do another little programming project I enjoy when learning a new language: the numbers guessing game.

Guessing Game

The guessing game is really simple: You enter in a number between 1 and 100. The computer tells you if you're too low, too high, or just right. You get five tries, after which the computer tells you the answer if you haven't gotten it yet.

I pick this example because it's fun, not too hard, and lets us do text-based I/O with a teeny bit of logic. Let's go!

Using `stdin()`

Turns out getting text input is pretty simple. Just try this:

```
use std::io;

fn main() {
    println!("INPUT:");
    let mut reader = io::stdin();

    let input = reader.read_line().ok().expect("Failed to read line");
    println!("YOU TYPED:");
    println!("{:s}", input);
}
```

Give that a run. It should prompt you to type something in, and then echo out what you typed. Simple enough!

I want to talk about that import, but first, let's go over this `stdin()` business. Basically. `io::stdin()` will give you a reference to standard in. Next, the `read_line()` method. This reads stuff up to a `\n` from whatever it's implemented on. So we grab that line, save it in a variable, and then print it out again. Super simple.

This `ok().expect()` business we'll talk about in a minute. First, what's up with this `use` shenanigans?

How to use `use`

Let's talk about modules. One of the big things that sorta sucks about C (and Ruby) is that 'modules' are basically based on files. You include the file, and that's about it. There's no way to really qualify "I want this bit of code," you say "I want this file that happens to have this code in it."

Rust basically pretends that it has these two lines at the beginning of every program:

```
extern crate std;
use std::prelude::*;
```

Two things here. The first line is this `extern crate` business. I wanted to clarify my understanding, so I jumped into the ever helpful Rust IRC and asked:

```
pcwalton: basically "extern mod" is where you put the stuff you'd put on the compiler link l
```

(`extern crate` was `extern mod` back then.)

Right. So we're saying 'please link against this library.' Rust uses a load path to find where those libraries are, which you can modify with the `-L` command-line flag. For instance:

```
$ rustc -L ./lib -o foo foo.rs
```

Would compile `foo.rs` into `foo` while also looking for extra libraries in the `lib` directory. These libraries are called 'crates' in Rust, and you can make one of your own with the `--lib` flag to `rustc`:

```
$ rustc --lib -o bar bar.rs
```

This would make a shared library crate named `bar`. Technically, any time you compile something, it makes a crate: the `--lib` flag just says that we're making a shared library explicitly, so Rust won't look for a `main()`. When you invoke `rustc` normally, you're also building a crate, it's just not shared.

Okay, so, once you've imported a crate, what do you get? Well, it will put a module with the same name as the crate into the current scope. But crates can also have other modules, which only get imported when you qualify them.

Modules?

Every Rust file can contain one top-level module, and modules can contain other modules. Modules look like this:

```
    mod foo {
        pub fn bar() { "bar" }
        pub fn baz() { "baz" }
        pub fn qux() { "qux" }
    }
```

You just shove a `mod` around everything that goes in the module. To bring `bar` into scope, you:

```
    use foo::bar;
```

To bring them all into scope, you:

```
    use foo::*;
```

To bring `bar` and `baz` into scope, but not `qux` you do either one of these:

```
    use foo::bar;
    use foo::baz;

    use foo::{bar,baz};
```

Pretty simple. So now we can see why the code acts like it has these two lines at the top:

```
    extern crate std;
    use std::prelude::*;
```

We want to link against the core library, and then import all the default io stuff into scope (that's what the prelude is). This is why we need:

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

Casting to integer

So, I was trying to cast a string to an integer to get this program going. So I wrote this:

```
use std::io;

fn main() {
    println!("INPUT:");
    let mut reader = io::stdin();

    let input = reader.read_line().ok().expect("Failed to read line");
    let input_num: Option<int> = from_str(input.as_slice().trim());

    println!("YOU TYPED:");
    println!("{}", input_num);
}
```

I was gonna convert the string to an int, then back to a string to print it out to the screen.

This gave an odd result:

```
$ rustc casting.rs && ./casting
INPUT:
5
YOU TYPED:
Some(5)
```

Wait, huh? Here's the thing: Rust **knows** that we might have a string that doesn't make any sense as an integer. For example: `"foo"`. So it doesn't actually return a string, it returns an `Option`. We can then use pattern matching to handle both cases. Observe:

```
use std::io;

fn main() {
    println!("INPUT:");
    let mut reader = io::stdin();

    let input = reader.read_line().ok().expect("Failed to read line");
    let input_num: Option<int> = from_str(input.as_slice().trim());

    match input_num {
        Some(number) => println!("{:d}", number),
        None          => println!("Hey, put in a number.")
    }
}
```

Remember `match`? It's really good for matching against some kind of type and breaking it up. Here we match against our `Option` type. `Option` looks like this:

```
enum Option<T> {
    Some(T),
    None
}
```

`Option` is called `Maybe` in some other languages, but basically, you can think of it as a type that handles what we'd use `nil` for in Ruby. We may have `Some(int)`, but we also may have `None`. Computations that may fail in some way should return `None` if it fails. Simple. We can't ever ignore a possible failure: the type system makes us handle it.

Looping forever

Looping forever is possible with `while true`, but like in Ruby, that's kinda silly. Rust gives us `loop` to loop forever:

```
loop {
    println!("HELLO")
}
```

Obviously you don't want to actually run that. You can use `break` to break out of the loop:

```
let mut i = 0;
loop {
    i += 1;
    if i == 5 { break; }
    println!("hi");
}
```

This will print `"hi"` four times. You're going to want to do this, because if someone mis-types a number, we don't want to count it against them: we should just ask them to put in another number.

Random Number Generation

Random number generation isn't too bad:

```
use std::rand::Rng;

fn main() {
    let secret_number = std::rand::task_rng().gen_range(1i, 101);
    println!("{:d}", secret_number);
}
```

This will print out a different number each time you run it. This will get us a random number between 1 and 100.

Okay! You should have all the tools you need to implement the guessing game. Have it it. I'm starting… now.

My version

Okay! That took me… about half an hour. Maybe 45 minutes. I found some helpful stuff in the standard library we didn't talk about: `cmp`, mainly:

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

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

    let secret_number = std::rand::task_rng().gen_range(1i, 101);
    println!("Secret number is {}", secret_number);

    let max_number_of_tries = 5
    let mut guesses: int = 0;
    let mut reader = io::stdin();

    loop {
        if guesses == max_number_of_tries {
          println!("You failed to guess within the limit of {:d} guesses!", NUM_OF_TRIES);
          break;
        }
        println!("Please input guess number {}", guesses + 1);

        let input = reader.read_line().ok().expect("Failed to read line");
        let input_num: Option<int> = from_str(input.as_slice().trim());
        let num = match input_num  {
            Some(num) => num,
            None      => {
                println!("Please input a number.");
                continue;
            }
        };

        println!("You guessed: {}", num);
        guesses += 1;

        match num.cmp(&secret_number) {
            Less    => println!("Too small!"),
            Greater => println!("Too big!"),
            Equal   => {
                println!("You win!");
                println!("You took {} guesses!", guesses);
                break;
            },
        }
    }
}
```

Conclusion

I'm pretty sure at this point we have basically everything I was able to do as a child when programming stuff. You know enough of Rust now to be able to make silly little games and scripts. This is obviously neat, but from this point on, it's more about libraries, style, and solving things in an idiomatic way than it is learning syntax. Of course, this was not a complete introduction to the language, but this is the end of the 'beginner level' stuff. You should have a basic idea of how to write many programs by this point. Pick a few projects, try them out.