A Gentle Introduction To Rust

Rust
I'll be designated driver tonight!
we know
Don't talk while eating you could choke and die haha

thanks to David Marino

Why learn a new Programming Language?

The aim of this tutorial is to take you to a place where you can read and write enough Rust to fully appreciate the excellent learning resources available online, in particular The Book. It's an opportunity to *try before you buy*, and get enough feeling for the power of the language to want to go deeper.

As Einstein might have said, "As gentle as possible, but no gentler.". There is a lot of new stuff to learn here, and it's different enough to require some rearrangement of your mental furniture. By 'gentle' I mean that the features are presented practically with examples; as we encounter difficulties, I hope to show how Rust solves these problems. It is important to understand the problems before the solutions make sense. To put it in flowery language, we are going for a hike in hilly country and I will point out some interesting rock formations on the way, with only a few geology lectures. There will be some uphill but the view will be inspiring; the community is unusually pleasant and happy to help. There is the Rust Users Forum and an active subreddit which is unusually well-moderated. The FAQ is a good resource if you have specific questions.

First, why learn a new programming language? It is an investment of time and energy and that needs some justification. Even if you do not immediately land a cool job using that language, it stretches the mental muscles and makes you a better programmer. That seems a poor kind of return-on-investment but if you're not learning something *genuinely* new all the time then you will stagnate and be like the person who has ten years of experience in doing the same thing over and over.

Where Rust Shines

Rust is a statically and strongly typed systems programming language. *statically* means that all types are known at compile-time, *strongly* means that these types are designed to make it harder to write incorrect programs. A successful compilation means you have a much better guarantee of correctness than with a cowboy language like C. *systems* means generating the best possible machine code with full control of memory use. So the uses are pretty hardcore: operating systems, device drivers and embedded systems that might not even have an operating system. However, it's actually a very pleasant language to write normal application code in as well.

The big difference from C and C++ is that Rust is *safe by default*; all memory accesses are checked. It is not possible to corrupt memory by accident.

The unifying principles behind Rust are:

- strictly enforcing *safe borrowing* of data
- functions, methods and closures to operate on data
- tuples, structs and enums to aggregate data
- pattern matching to select and destructure data
- traits to define *behaviour* on data

There is a fast-growing ecosystem of available libraries through Cargo but here we will concentrate on the core principles of the language by learning to use the standard library. My advice is to write *lots of small programs*, so learning to use `rustc` directly is a core skill. When doing the examples in this tutorial I defined a little script called `rrun` which does a compilation and runs the result:

```
rustc $1.rs && ./$1
```

Setting Up

This tutorial assumes that you have Rust installed locally. Fortunately this is very straightforward.

```
$ curl https://sh.rustup.rs -sSf | sh
$ rustup component add rust-docs
```

I would recommend getting the default stable version; it's easy to download unstable versions later and to switch between.

This gets the compiler, the Cargo package manager, the API documentation, and the Rust Book. The journey of a thousand miles starts with one step, and this first step is painless.

`rustup` is the command you use to manage your Rust installation. When a new stable release appears, you just have to say `rustup update` to upgrade. `rustup doc` will open the offline documentation in your browser.

You will probably already have an editor you like, and basic Rust support is good. I'd suggest you start out with basic syntax highlighting at first, and work up as your programs get larger.

Personally I'm a fan of Geany which is one of the few editors with Rust support out-of-the-box; it's particularly easy on Linux since it's available through the package manager, but it works fine on other platforms.

The main thing is knowing how to edit, compile and run Rust programs. You learn to program with your *fingers*; type in the code yourself, and learn to rearrange things efficiently with your editor.

Zed Shaw's advice about learning to program in Python remains good, whatever the language. He says learning to program is like learning a musical instrument - the secret is practice and persistence. There's also good advice from Yoga and the soft martial arts like Tai Chi; feel the strain, but don't over-strain. You are not building dumb muscle here.

I'd like to thank the many contributors who caught bad English or bad Rust for me, and thanks to David Marino for his cool characterization of Rust as a friendly-but-hardcore no-nonsense knight in shining armour.

Basics

Hello, World!

The original purpose of "hello world", ever since the first C version was written, was to test the compiler and run an actual program.

```
// hello.rs
fn main() {
    println!("Hello, World!");
}
```

```
$ rustc hello.rs
$ ./hello
Hello, World!
```

Rust is a curly-braces language with semicolons, C++-style comments and a `main` function - so far, so familiar. The exclamation mark indicates that this is a *macro* call. For C++ programmers, this can be a turn-off, since they are used to seriously stupid C macros - but I can ensure you that these macros are more capable and sane.

For anybody else, it's probably "Great, now I have to remember when to say bang!". However, the compiler is unusually helpful; if you leave out that exclamation, you get:

```
error[E0425]: unresolved name `println`
 --> hello2.rs:2:5
  |
2 |     println("Hello, World!");
  |     ^^^^^^^ did you mean the macro `println!`?
```

Learning a language means getting comfortable with its errors. Try to see the compiler as a strict but friendly helper rather than a computer *shouting* at you, because you are going to see a lot of red ink in the beginning. It's much better for the compiler to catch you out than for your program to blow up in front of actual humans.

The next step is to introduce a *variable*:

```
// let1.rs
fn main() {
    let answer = 42;
    println!("Hello {}", answer);
}
```

Spelling mistakes are *compile* errors, not runtime errors like with dynamic languages like Python or JavaScript. This will save you a lot of stress later! And if I wrote 'answr' instead of 'answer', the compiler is actually *nice* about it:

```
4 |     println!("Hello {}", answr);
  |                          ^^^^^ did you mean `answer`?
```

The `println!` macro takes a format string and some values; it's very similar to the formatting used by Python 3.

Another very useful macro is `assert_eq!` . This is the workhorse of testing in Rust; you *assert* that two things must be equal, and if not, *panic*.

```
// let2.rs
fn main() {
    let answer = 42;
    assert_eq!(answer,42);
}
```

Which won't produce any output. But change 42 to 40:

```
thread 'main' panicked at
'assertion failed: `(left == right)` (left: `42`, right: `40`)',
let2.rs:4
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

And that's our first *runtime error* in Rust.

Looping and Ifing

Anything interesting can be done more than once:

```
// for1.rs
fn main() {
    for i in 0..5 {
        println!("Hello {}", i);
    }
}
```

The *range* is not inclusive, so `i` goes from 0 to 4. This is convenient in a language which *indexes* things like arrays from 0.

And interesting things have to be done *conditionally*:

```
// for2.rs
fn main() {
    for i in 0..5 {
        if i % 2 == 0 {
            println!("even {}", i);
        } else {
            println!("odd {}", i);
        }
    }
}
```

```
even 0
odd 1
even 2
odd 3
even 4
```

`i % 2` is zero if 2 can divide into `i` cleanly; Rust uses C-style operators. There are *no* brackets around the condition, just like in Go, but you *must* use curly brackets around the block.

This does the same, written in a more interesting way:

```
// for3.rs
fn main() {
    for i in 0..5 {
        let even_odd = if i % 2 == 0 {"even"} else {"odd"};
        println!("{} {}", even_odd, i);
    }
}
```

Traditionally, programming languages have *statements* (like `if`) and *expressions* (like `1+i`). In Rust, nearly everything has a value and can be an expression. The seriously ugly C 'ternary operator' `i % 2 == 0 ? "even" : "odd"` is not needed.

Note that there aren't any semi-colons in those blocks!

Adding Things Up

Computers are very good at arithmetic. Here is a first attempt at adding all the numbers from 0 to 4:

```
// add1.rs
fn main() {
    let sum = 0;
    for i in 0..5 {
        sum += i;
    }
    println!("sum is {}", sum);
}
```

But it fails to compile:

```
error[E0384]: re-assignment of immutable variable `sum`
 --> add1.rs:5:9
3 |     let sum = 0;
  |         --- first assignment to `sum`
4 |     for i in 0..5 {
5 |         sum += i;
  |         ^^^^^^^^ re-assignment of immutable variable
```

'Immutable'? A variable that cannot *vary*? `let` variables by default can only be assigned a value when declared. Adding the magic word `mut` (*please* make this variable mutable) does the trick:

```
// add2.rs
fn main() {
    let mut sum = 0;
    for i in 0..5 {
        sum += i;
    }
    println!("sum is {}", sum);
}
```

This can be puzzling when coming from other languages, where variables can be re-written by default. What makes something a 'variable' is that it gets assigned a computed value at run-time - it is not a *constant*. It is also how the word is used in mathematics, like when we say 'let n be the largest number in set S'.

There is a reason for declaring variables to be *read-only* by default. In a larger program, it gets hard to track where writes are taking place. So Rust makes things like mutability ('write-ability') explicit. There's a lot of cleverness in the language, but it tries not to hide anything.

Rust is both statically-typed and strongly-typed - these are often confused, but think of C (statically but weakly typed) and Python (dynamically but strongly typed). In static types the type is known at compile time, and dynamic types are only known at run time.

At the moment, however, it feels like Rust is *hiding* those types from you. What exactly is the type of `i`? The compiler can work it out, starting with 0, with *type inference*, and comes up with `i32` (four byte signed integer.)

Let's make exactly one change - turn that `0` into `0.0`. Then we get errors:

```
error[E0277]: the trait bound `{float}: std::ops::AddAssign<{integer}>` is not satisfied
 --> add3.rs:5:9
  |
5 |         sum += i;
  |         ^^^^^^^^ the trait `std::ops::AddAssign<{integer}>` is not implemented for `{float}`
  |
```

Ok, so the honeymoon is over: what does this mean? Each operator (like `+=`) corresponds to a *trait*, which is like an abstract interface that must be implemented for each concrete type. We'll deal with traits in detail later, but here all you need to know is that `AddAssign` is the name of the trait implementing the `+=` operator, and the error is saying that floating point numbers do not implement this operator for integers. (The full list of operator traits is here.)

Again, Rust likes to be explicit - it will not silently convert that integer into a float for you.

We have to *cast* that value to a floating-point value explicitly.

```
// add3.rs
fn main() {
    let mut sum = 0.0;
    for i in 0..5 {
        sum += i as f64;
    }
    println!("sum is {}", sum);
}
```

Function Types are Explicit

Functions are one place where the compiler will not work out types for you. And this in fact was a deliberate decision, since languages like Haskell have such powerful type inference that there are hardly any explicit type names. It's actually good Haskell style to put in explicit type signatures for functions. Rust requires this always.

Here is a simple user-defined function:

```
// fun1.rs

fn sqr(x: f64) -> f64 {
    return x * x;
}

fn main() {
    let res = sqr(2.0);
    println!("square is {}", res);
}
```

Rust goes back to an older style of argument declaration, where the type follows the name. This is how it was done in Algol-derived languages like Pascal.

Again, no integer-to-float conversions - if you replace the `2.0` with `2` then we get a clear error:

```
8 |     let res = sqr(2);
  |                   ^ expected f64, found integral variable
  |
```

You will actually rarely see functions written using a `return` statement. More often, it will look like this:

```
fn sqr(x: f64) -> f64 {
    x * x
}
```

This is because the body of the function (inside `{}`) has the value of its last expression, just like with if-as-an-expression.

Since semicolons are inserted semi-automatically by human fingers, you might add it here and get the following error:

```
  |
3 | fn sqr(x: f64) -> f64 {
  |                       ^ expected f64, found ()
  |
  = note: expected type `f64`
  = note:    found type `()`
help: consider removing this semicolon:
 --> fun2.rs:4:8
  |
4 |     x * x;
  |       ^
```

The `()` type is the empty type, nada, `void`, zilch, nothing. Everything in Rust has a value, but sometimes it's just nothing. The compiler knows this is a common mistake, and actually *helps* you. (Anybody who has spent time with a C++ compiler will know how *damn unusual* this is.)

A few more examples of this no-return expression style:

```
// absolute value of a floating-point number
fn abs(x: f64) -> f64 {
    if x > 0.0 {
        x
    } else {
        -x
    }
}

// ensure the number always falls in the given range
fn clamp(x: f64, x1: f64, x2: f64) -> f64 {
    if x < x1 {
        x1
    } else if x > x2 {
        x2
    } else {
        x
    }
}
```

It's not wrong to use `return`, but code is cleaner without it. You will still use `return` for *returning early* from a function.

Some operations can be elegantly expressed *recursively*:

```
fn factorial(n: u64) -> u64 {
    if n == 0 {
        1
    } else {
        n * factorial(n-1)
    }
}
```

This can be a little strange at first, and the best thing is then to use pencil and paper and work out some examples. It isn't usually the most *efficient* way to do that operation however.

Values can also be passed by *reference*. A reference is created by `&` and *dereferenced* by `*`.

```
fn by_ref(x: &i32) -> i32{
    *x + 1
}

fn main() {
    let i = 10;
    let res1 = by_ref(&i);
    let res2 = by_ref(&41);
    println!("{} {}", res1,res2);
}
// 11 42
```

What if you want a function to modify one of its arguments? Enter *mutable references*:

```
// fun4.rs

fn modifies(x: &mut f64) {
    *x = 1.0;
}

fn main() {
    let mut res = 0.0;
    modifies(&mut res);
    println!("res is {}", res);
}
```

This is more how C would do it than C++. You have to explicitly pass the reference (with `&`) and explicitly *dereference* with `*`. And then throw in `mut` because it's not the default. (I've always felt that C++ references are too easy to miss compared to C.)

Basically, Rust is introducing some *friction* here, and not-so-subtly pushing you towards returning values from functions directly. Fortunately, Rust has powerful ways to express things like "operation succeeded and here's the result" so `&mut` isn't needed that often. Passing by reference is important when we have a large object and don't wish to copy it.

The type-after-variable style applies to `let` as well, when you really want to nail down the type of a variable:

```
let bigint: i64 = 0;
```

Learning Where to Find the Ropes

It's time to start using the documentation. This will be installed on your machine, and you can use `rustup doc --std` to open it in a browser.

Note the *search* field at the top, since this is going to be your friend; it operates completely offline.

Let's say we want to see where the mathematical functions are, so search for 'cos'. The first two hits show it defined for both single and double-precision floating point numbers. It is defined on the *value itself* as a method, like so:

```
let pi: f64 = 3.1416;
let x = pi/2.0;
let cosine = x.cos();
```

And the result will be sort-of zero; we obviously need a more authoritative source of pi-ness!

(Why do we need an explicit `f64` type? Because without it, the constant could be either `f32` or `f64`, and these are very different.)

Let me quote the example given for `cos`, but written as a complete program (`assert!` is a cousin of `assert_eq!`; the expression must be true):

```
fn main() {
    let x = 2.0 * std::f64::consts::PI;

    let abs_difference = (x.cos() - 1.0).abs();

    assert!(abs_difference < 1e-10);
}
```

`std::f64::consts::PI` is a mouthful! `::` means much the same as it does in C++, (often written using '.' in other languages) - it is a *fully qualified name*. We get this full name from the second hit on searching for `PI`.

Up to now, our little Rust programs have been free of all that `import` and `include` stuff that tends to slow down the discussion of 'Hello World' programs. Let's make this program more readable with a `use` statement:

```
use std::f64::consts;

fn main() {
    let x = 2.0 * consts::PI;

    let abs_difference = (x.cos() - 1.0).abs();

    assert!(abs_difference < 1e-10);
}
```

Why haven't we needed to do this up to now? This is because Rust helpfully makes a lot of basic functionality visible without explicit `use` statements through the Rust *prelude*.

Arrays and Slices

All statically-typed languages have *arrays*, which are values packed nose to tail in memory. Arrays are *indexed* from zero:

```
// array1.rs
fn main() {
    let arr = [10, 20, 30, 40];
    let first = arr[0];
    println!("first {}", first);

    for i in 0..4 {
        println!("[{}] = {}", i,arr[i]);
    }
    println!("length {}", arr.len());
}
```

And the output is:

```
first 10
[0] = 10
[1] = 20
[2] = 30
[3] = 40
length 4
```

In this case, Rust knows *exactly* how big the array is and if you try to access `arr[4]` it will be a *compile error*.

Learning a new language often involves *unlearning* mental habits from languages you already know; if you are a Pythonista, then those brackets say `List`. We will come to the Rust equivalent of `List` soon, but arrays are not the droids you're looking for; they are *fixed in size*. They can be *mutable* (if we ask nicely) but you cannot add new elements.

Arrays are not used that often in Rust, because the type of an array includes its size. The type of the array in the example is `[i32; 4]`; the type of `[10, 20]` would be `[i32; 2]` and so forth: they have *different types*. So they are bastards to pass around as function arguments.

What *are* used often are *slices*. You can think of these as *views* into an underlying array of values. They otherwise behave very much like an array, and *know their size*, unlike those dangerous animals C pointers.

Note two important things here - how to write a slice's type, and that you have to use `&` to pass it to the function.

```
// array2.rs
// read as: slice of i32
fn sum(values: &[i32]) -> i32 {
    let mut res = 0;
    for i in 0..values.len() {
        res += values[i]
    }
    res
}

fn main() {
    let arr = [10,20,30,40];
    // look at that &
    let res = sum(&arr);
    println!("sum {}", res);
}
```

Ignore the code of `sum` for a while, and look at `&[i32]`. The relationship between Rust arrays and slices is similar to that between C arrays and pointers, except for two important differences - Rust slices keep track of their size (and will panic if you try to access outside that size) and you have to explicitly say that you want to pass an array as a slice using the `&` operator.

A C programmer pronounces `&` as 'address of'; a Rust programmer pronounces it 'borrow'. This is going to be the key word when learning Rust. Borrowing is the name given to a common pattern in programming; whenever you pass something by reference (as nearly always happens in dynamic languages) or pass a pointer in C. Anything borrowed remains owned by the original owner.

Slicing and Dicing

You cannot print out an array in the usual way with `{}` but you can do a *debug* print with `{:?}`.

```
// array3.rs
fn main() {
    let ints = [1, 2, 3];
    let floats = [1.1, 2.1, 3.1];
    let strings = ["hello", "world"];
    let ints_ints = [[1, 2], [10, 20]];
    println!("ints {:?}", ints);
    println!("floats {:?}", floats);
    println!("strings {:?}", strings);
    println!("ints_ints {:?}", ints_ints);
}
```

Which gives:

```
ints [1, 2, 3]
floats [1.1, 2.1, 3.1]
strings ["hello", "world"]
ints_ints [[1, 2], [10, 20]]
```

So, arrays of arrays are no problem, but the important thing is that an array contains values of *only one type*. The values in an array are arranged next to each other in memory so that they are *very* efficient to access.

If you are curious about the actual types of these variables, here is a useful trick. Just declare a variable with an explicit type which you know will be wrong:

```
let var: () = [1.1, 1.2];
```

Here is the informative error:

```
3 |     let var: () = [1.1, 1.2];
  |                   ^^^^^^^^^^ expected (), found array of 2 elements
  |
  = note: expected type `()`
  = note:    found type `[{float}; 2]`
```

(`{float}` means 'some floating-point type which is not fully specified yet')

Slices give you different *views* of the *same* array:

```
// slice1.rs
fn main() {
    let ints = [1, 2, 3, 4, 5];
    let slice1 = &ints[0..2];
    let slice2 = &ints[1..];  // open range!

    println!("ints {:?}", ints);
    println!("slice1 {:?}", slice1);
    println!("slice2 {:?}", slice2);
}
```

```
ints [1, 2, 3, 4, 5]
slice1 [1, 2]
slice2 [2, 3, 4, 5]
```

This is a neat notation which looks similar to Python slices but with a big difference: a copy of the data is never made. These slices all *borrow* their data from their arrays. They have a very intimate relationship with that array, and Rust spends a lot of effort to make sure that relationship does not break down.

Optional Values

Slices, like arrays, can be *indexed*. Rust knows the size of an array at compile-time, but the size of a slice is only known at run-time. So `s[i]` can cause an out-of-bounds error when running and will *panic*. This is really not what you want to happen - it can be the difference between a safe launch abort and scattering pieces of a very expensive satellite all over Florida. And there are *no exceptions*.

Let that sink in, because it comes as a shock. You cannot wrap dodgy-may-panic code in some try-block and 'catch the error' - at least not in a way you'd want to use every day. So how can Rust be safe?

There is a slice method `get` which does not panic. But what does it return?

```
// slice2.rs
fn main() {
    let ints = [1, 2, 3, 4, 5];
    let slice = &ints;
    let first = slice.get(0);
    let last = slice.get(5);

    println!("first {:?}", first);
    println!("last {:?}", last);
}
// first Some(1)
// last None
```

`last` failed (we forgot about zero-based indexing), but returned something called `None`. `first` is fine, but appears as a value wrapped in `Some`. Welcome to the `Option` type! It may be *either* `Some` or `None`.

The `Option` type has some useful methods:

```
    println!("first {} {}", first.is_some(), first.is_none());
    println!("last {} {}", last.is_some(), last.is_none());
    println!("first value {}", first.unwrap());

// first true false
// last false true
// first value 1
```

If you were to *unwrap* `last`, you would get a panic. But at least you can call `is_some` first to make sure - for instance, if you had a distinct no-value default:

```
    let maybe_last = slice.get(5);
    let last = if maybe_last.is_some() {
        *maybe_last.unwrap()
    } else {
        -1
    };
```

Note the `*` - the precise type inside the `Some` is `&i32`, which is a reference. We need to dereference this to get back to a `i32` value.

Which is long-winded, so there's a shortcut - `unwrap_or` will return the value it is given if the `Option` was `None`. The types must match up - `get` returns a reference. so you have to make up a `&i32` with `&-1`. Finally, again use `*` to get the value as `i32`.

```
    let last = *slice.get(5).unwrap_or(&-1);
```

It's easy to miss the `&`, but the compiler has your back here. If it was `-1`, `rustc` says 'expected &{integer}, found integral variable' and then 'help: try with `&-1`'.

You can think of `Option` as a box which may contain a value, or nothing (`None`). (It is called `Maybe` in Haskell). It may contain *any* kind of value, which is its *type parameter*. In this case, the full type is `Option<&i32>`, using C++-style notation for *generics*. Unwrapping this box may cause an explosion, but unlike Schroedinger's Cat, we can know in advance if it contains a value.

It is very common for Rust functions/methods to return such maybe-boxes, so learn how to use them comfortably.

Vectors

We'll return to slice methods again, but first: vectors. These are *re-sizeable* arrays and behave much like Python `List` and C++ `std::vector`. The Rust type `Vec` (pronounced 'vector') behaves very much like an slice in fact; the difference is that you can append extra values to a vector - note that it must be declared as mutable.

```
// vec1.rs
fn main() {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v.push(30);

    let first = v[0];  // will panic if out-of-range
    let maybe_first = v.get(0);

    println!("v is {:?}", v);
    println!("first is {}", first);
    println!("maybe_first is {:?}", maybe_first);
}
// v is [10, 20, 30]
// first is 10
// maybe_first is Some(10)
```

A common beginner mistake is to forget the `mut`; you will get a helpful error message:

```
3 |     let v = Vec::new();
  |         - use `mut v` here to make mutable
4 |     v.push(10);
  |     ^ cannot borrow mutably
```

There is a very intimate relation between vectors and slices:

```
// vec2.rs
fn dump(arr: &[i32]) {
    println!("arr is {:?}", arr);
}

fn main() {
    let mut v = Vec::new();
    v.push(10);
    v.push(20);
    v.push(30);

    dump(&v);

    let slice = &v[1..];
    println!("slice is {:?}", slice);
}
```

That little, so-important borrow operator `&` is *coercing* the vector into a slice. And it makes complete sense, because the vector is also looking after an array of values, with the difference that the array is allocated *dynamically*.

If you come from a dynamic language, now is time for that little talk. In systems languages, program memory comes in two kinds: the stack and the heap. It is very fast to allocate data on the stack, but the stack is limited; typically of the order of megabytes. The heap can be gigabytes, but allocating is relatively expensive, and such memory must be *freed* later. In so-called 'managed' languages (like Java, Go and the so-called 'scripting' languages) these details are hidden from you by that convenient municipal utility called the *garbage collector*. Once the system is sure that data is no longer referenced by other data, it goes back into the pool of available memory.

Generally, this is a price worth paying. Playing with the stack is terribly unsafe, because if you make one mistake you can override the return address of the current function, and you die an ignominious death or (worse) got pwned by some guy living in his Mom's basement in Minsk.

The first C program I wrote (on an DOS PC) took out the whole computer. Unix systems always behaved better, and only the process died with a *segfault*. Why is this worse than a Rust (or Go) program panicking? Because a panic happens when the original problem happens, not when the program has become hopelessly confused and eaten all your homework. Panics are *memory safe* because they happen before any illegal access to memory. This is a common cause of security problems in C, because all memory accesses are unsafe and a cunning attacker can exploit this weakness.

Panicking sounds desperate and unplanned, but Rust panics are structured - the stack is *unwound* just as with exceptions. All allocated objects are dropped, and a backtrace is generated.

The downsides of garbage collection? The first is that it is wasteful of memory, which matters in those small embedded microchips which increasingly rule our world. The second is that it will decide, at the worst possible time, that a clean up must happen *now*. (The Mom analogy is that she wants to clean your room when you are at a delicate stage with a new lover). Those embedded systems need to respond to things *when they happen* ('real-time') and can't tolerate unscheduled outbreaks of cleaning. Roberto Ierusalimschy, the chief designer of Lua (one of the most elegant dynamic languages ever) said that he would not like to fly on an airplane that relied on garbage-collected software.

Back to vectors: when a vector is modified or created, it allocates from the heap and becomes the *owner* of that memory. The slice *borrows* the memory from the vector. When the vector dies or *drops*, it lets the memory go.

Iterators

We have got so far without mentioning a key part of the Rust puzzle - iterators. The for-loop over a range was using an iterator (`0..n` is actually similar to the Python 3 `range` function).

An iterator is easy to define informally. It is an 'object' with a `next` method which returns an `Option`. As long as that value is not `None`, we keep calling `next`:

```
// iter1.rs
fn main() {
    let mut iter = 0..3;
    assert_eq!(iter.next(), Some(0));
    assert_eq!(iter.next(), Some(1));
    assert_eq!(iter.next(), Some(2));
    assert_eq!(iter.next(), None);
}
```

And that is exactly what `for var in iter {}` does.

This may seem an inefficient way to define a for-loop, but `rustc` does crazy-ass optimizations in release mode and it will be just as fast as a `while` loop.

Here is the first attempt to iterate over an array:

```
// iter2.rs
fn main() {
    let arr = [10, 20, 30];
    for i in arr {
        println!("{}", i);
    }
}
```

which fails, but helpfully:

```
4 |     for i in arr {
  |     ^ the trait `std::iter::Iterator` is not implemented for `[{integer}; 3]`
  |
  = note: `[{integer}; 3]` is not an iterator; maybe try calling
   `.iter()` or a similar method
  = note: required by `std::iter::IntoIterator::into_iter`
```

Following `rustc`'s advice, the following program works as expected.

```
// iter3.rs
fn main() {
    let arr = [10, 20, 30];
    for i in arr.iter() {
        println!("{}", i);
    }

    // slices will be converted implicitly to iterators...
    let slice = &arr;
    for i in slice {
        println!("{}", i);
    }
}
```

In fact, it is more efficient to iterate over an array or slice this way than to use `for i in 0..slice.len() {}` because Rust does not have to obsessively check every index operation.

We had an example of summing up a range of integers earlier. It involved a `mut` variable and a loop. Here's the *idiomatic*, pro-level way of doing the sum:

```
// sum1.rs
fn main() {
    let sum: i32  = (0..5).sum();
    println!("sum was {}", sum);

    let sum: i64 = [10, 20, 30].iter().sum();
    println!("sum was {}", sum);
}
```

Note that this is one of those cases where you need to be explicit about the *type* of the variable, since otherwise Rust doesn't have enough information. Here we do sums with two different integer sizes, no problem. (It is also no problem to create a new variable of the same name if you run out of names to give things.)

With this background, some more of the slice methods will make more sense. (Another documentation tip; on the right-hand side of every doc page there's a '[-]' which you can click to collapse the method list. You can then expand the details of anything that looks interesting. Anything that looks too weird, just ignore for now.)

The `windows` method gives you an iterator of slices - overlapping windows of values!

```
// slice4.rs
fn main() {
    let ints = [1, 2, 3, 4, 5];
    let slice = &ints;

    for s in slice.windows(2) {
        println!("window {:?}", s);
    }
}
// window [1, 2]
// window [2, 3]
// window [3, 4]
// window [4, 5]
```

Or `chunks`:

```
    for s in slice.chunks(2) {
        println!("chunks {:?}", s);
    }
// chunks [1, 2]
// chunks [3, 4]
// chunks [5]
```

More about vectors...

There is a useful little macro `vec!` for initializing a vector. Note that you can *remove* values from the end of a vector using `pop`, and *extend* a vector using any compatible iterator.

```
// vec3.rs
fn main() {
    let mut v1 = vec![10, 20, 30, 40];
    v1.pop();

    let mut v2 = Vec::new();
    v2.push(10);
    v2.push(20);
    v2.push(30);

    assert_eq!(v1, v2);

    v2.extend(0..2);
    assert_eq!(v2, &[10, 20, 30, 0, 1]);
}
```

Vectors compare with each other and with slices by value.

You can insert values into a vector at arbitrary positions with `insert`, and remove with `remove`. This is not as efficient as pushing and popping since the values will have to be moved to make room, so watch out for these operations on big vectors.

Vectors have a size and a *capacity*. If you `clear` a vector, its size becomes zero, but it still retains its old capacity. So refilling it with `push`, etc only requires reallocation when the size gets larger than that capacity.

Vectors can be sorted, and then duplicates can be removed - these operate in-place on the vector. (If you want to make a copy first, use `clone` .)

```
// vec4.rs
fn main() {
    let mut v1 = vec![1, 10, 5, 1, 2, 11, 2, 40];
    v1.sort();
    v1.dedup();
    assert_eq!(v1, &[1, 2, 5, 10, 11, 40]);
}
```

Strings

Strings in Rust are a little more involved than in other languages; the `String` type, like `Vec` , allocates dynamically and is resizeable. (So it's like C++'s `std::string` but not like the immutable strings of Java and Python.) But a program may contain a lot of *string literals* (like "hello") and a system language should be able to store these statically in the executable itself. In embedded micros, that could mean putting them in cheap ROM rather than expensive RAM (for low-power devices, RAM is also expensive in terms of power consumption.) A *system* language has to have two kinds of string, allocated or static.

So "hello" is not of type `String` . It is of type `&str` (pronounced 'string slice'). It's like the distinction between `const char*` and `std::string` in C++, except `&str` is much more intelligent. In fact, `&str` and `String` have a very similar relationship to each other as do `&[T]` to `Vec<T>` .

```
// string1.rs
fn dump(s: &str) {
    println!("str '{}'", s);
}

fn main() {
    let text = "hello dolly";  // the string slice
    let s = text.to_string();  // it's now an allocated string

    dump(text);
    dump(&s);
}
```

Again, the borrow operator can coerce `String` into `&str` , just as `Vec<T>` could be coerced into `&[T]` .

Under the hood, `String` is basically a `Vec<u8>` and `&str` is `&[u8]` , but those bytes *must* represent valid UTF-8 text.

Like a vector, you can `push` a character and `pop` one off the end of `String` :

```rust
// string5.rs
fn main() {
    let mut s = String::new();
    // initially empty!
    s.push('H');
    s.push_str("ello");
    s.push(' ');
    s += "World!"; // short for `push_str`
    // remove the last char
    s.pop();

    assert_eq!(s, "Hello World");
}
```

You can convert many types to strings using `to_string` (if you can display them with '{}' then they can be converted). The `format!` macro is a very useful way to build up more complicated strings using the same format strings as `println!`.

```rust
// string6.rs
fn array_to_str(arr: &[i32]) -> String {
    let mut res = '['.to_string();
    for v in arr {
        res += &v.to_string();
        res.push(',');
    }
    res.pop();
    res.push(']');
    res
}

fn main() {
    let arr = array_to_str(&[10, 20, 30]);
    let res = format!("hello {}", arr);

    assert_eq!(res, "hello [10,20,30]");
}
```

Note the `&` in front of `v.to_string()` - the operator is defined on a string slice, not a `String` itself, so it needs a little persuasion to match.

The notation used for slices works with strings as well:

```rust
// string2.rs
fn main() {
    let text = "static";
    let string = "dynamic".to_string();

    let text_s = &text[1..];
    let string_s = &string[2..4];

    println!("slices {:?} {:?}", text_s, string_s);
}
// slices "tatic" "na"
```

But, you cannot index strings! This is because they use the One True Encoding, UTF-8, where a 'character' may be a number of bytes.

```
// string3.rs
fn main() {
    let multilingual = "Hi! ¡Hola! привет!";
    for ch in multilingual.chars() {
        print!("'{}' ", ch);
    }
    println!("");
    println!("len {}", multilingual.len());
    println!("count {}", multilingual.chars().count());

    let maybe = multilingual.find('п');
    if maybe.is_some() {
        let hi = &multilingual[maybe.unwrap()..];
        println!("Russian hi {}", hi);
    }
}
// 'H' 'i' '!' ' ' '¡' 'H' 'o' 'l' 'a' '!' ' ' 'п' 'р' 'и' 'в' 'е' 'т' '!'
// len 25
// count 18
// Russian hi привет!
```

Now, let that sink in - there are 25 bytes, but only 18 characters! However, if you use a method like `find`, you will get a valid index (if found) and then any slice will be fine.

(The Rust `char` type is a 4-byte Unicode code point. Strings are *not* arrays of chars!)

String slicing may explode like vector indexing, because it uses byte offsets. In this case, the string consists of two bytes, so trying to pull out the first byte is a Unicode error. So be careful to only slice strings using valid offsets that come from string methods.

```
let s = "¡";
println!("{}", &s[0..1]); <-- bad, first byte of a multibyte character
```

Breaking up strings is a popular and useful pastime. The string `split_whitespace` method returns an *iterator*, and we then choose what to do with it. A common need is to create a vector of the split substrings.

`collect` is very general and so needs some clues about *what* it is collecting - hence the explicit type.

```
let text = "the red fox and the lazy dog";
let words: Vec<&str> = text.split_whitespace().collect();
// ["the", "red", "fox", "and", "the", "lazy", "dog"]
```

You could also say it like this, passing the iterator into the `extend` method:

```
let mut words = Vec::new();
words.extend(text.split_whitespace());
```

In most languages, we would have to make these *separately allocated strings*, whereas here each slice in the vector is borrowing from the original string. All we allocate is the space to keep the slices.

Have a look at this cute two-liner; we get an iterator over the chars, and only take those characters which are not space. Again, `collect` needs a clue (we may have wanted a vector of chars, say):

```
    let stripped: String = text.chars()
        .filter(|ch| ! ch.is_whitespace()).collect();
    // theredfoxandthelazydog
```

The `filter` method takes a *closure*, which is Rust-speak for lambdas or anonymous functions. Here the argument type is clear from the context, so the explicit rule is relaxed.

Yes, you can do this as an explicit loop over chars, pushing the returned slices into a mutable vector, but this is shorter, reads well (*when* you are used to it, of course) and just as fast. It is not a *sin* to use a loop, however, and I encourage you to write that version as well.

Interlude: Getting Command Line Arguments

Up to now our programs have lived in blissful ignorance of the outside world; now it's time to feed them data.

`std::env::args` is how you access command-line arguments; it returns an iterator over the arguments as strings, including the program name.

```
// args0.rs
fn main() {
    for arg in std::env::args() {
        println!("'{}'", arg);
    }
}
```

```
src$ rustc args0.rs
src$ ./args0 42 'hello dolly' frodo
'./args0'
'42'
'hello dolly'
'frodo'
```

Would it have been better to return a `Vec`? It's easy enough to use `collect` to make that vector, using the iterator `skip` method to move past the program name.

```
    let args: Vec<String> = std::env::args().skip(1).collect();
    if args.len() > 0 { // we have args!
        ...
    }
```

Which is fine; it's pretty much how you would do it in most languages.

A more Rust-y approach to reading a single argument (together with parsing an integer value):

```
// args1.rs
use std::env;

fn main() {
    let first = env::args().nth(1).expect("please supply an argument");
    let n: i32 = first.parse().expect("not an integer!");
    // do your magic
}
```

`nth(1)` gives you the second value of the iterator, and `expect` is like an `unwrap` with a readable message.

Converting a string into a number is straightforward, but you do need to specify the type of the value - how else could `parse` know?

This program can panic, which is fine for dinky test programs. But don't get too comfortable with this convenient habit.

Matching

The code in `string3.rs` where we extract the Russian greeting is not how it would be usually written. Enter *match*:

```
match multilingual.find('п') {
    Some(idx) => {
        let hi = &multilingual[idx..];
        println!("Russian hi {}", hi);
    },
    None => println!("couldn't find the greeting, Товарищ")
};
```

`match` consists of several *patterns* with a matching value following the fat arrow, separated by commas. It has conveniently unwrapped the value from the `Option` and bound it to `idx`. You *must* specify all the possibilities, so we have to handle `None`.

Once you are used to it (and by that I mean, typed it out in full a few times) it feels more natural than the explicit `is_some` check which needed an extra variable to store the `Option`.

But if you're not interested in failure here, then `if let` is your friend:

```
if let Some(idx) = multilingual.find('п') {
    println!("Russian hi {}", &multilingual[idx..]);
}
```

This is convenient if you want to do a match and are *only* interested in one possible result.

`match` can also operate like a C `switch` statement, and like other Rust constructs can return a value:

```
    let text = match n {
        0 => "zero",
        1 => "one",
        2 => "two",
        _ => "many",
    };
```

The `_` is like C `default` - it's a fall-back case. If you don't provide one then `rustc` will consider it an error. (In C++ the best you can expect is a warning, which says a lot about the respective languages).

Rust `match` statements can also match on ranges. Note that these ranges have *three* dots and are inclusive ranges, so that the first condition would match 3.

```
    let text = match n {
        0...3 => "small",
        4...6 => "medium",
        _ => "large",
     };
```

Reading from Files

The next step to exposing our programs to the world is to *reading files*.

Recall that `expect` is like `unwrap` but gives a custom error message. We are going to throw away a few errors here:

```
// file1.rs
use std::env;
use std::fs::File;
use std::io::Read;

fn main() {
    let first = env::args().nth(1).expect("please supply a filename");

    let mut file = File::open(&first).expect("can't open the file");

    let mut text = String::new();
    file.read_to_string(&mut text).expect("can't read the file");

    println!("file had {} bytes", text.len());

}
```

```
src$ file1 file1.rs
file had 366 bytes
src$ ./file1 frodo.txt
thread 'main' panicked at 'can't open the file: Error { repr: Os { code: 2, message: "No such file
or directory" } }', ../src/libcore/result.rs:837
note: Run with `RUST_BACKTRACE=1` for a backtrace.
src$ file1 file1
thread 'main' panicked at 'can't read the file: Error { repr: Custom(Custom { kind: InvalidData,
error: StringError("stream did not contain valid UTF-8") }) }', ../src/libcore/result.rs:837
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```

So `open` can fail because the file doesn't exist or we aren't allowed to read it, and `read_to_string` can fail because the file doesn't contain valid UTF-8. (Which is fair enough, you can use `read_to_end` and put the contents into a vector of bytes instead.) For files that aren't too big, reading them in one gulp is useful and straightforward.

If you know anything about file handling in other languages, you may wonder when the file is *closed*. If we were writing to this file, then not closing it could result in loss of data. But the file here is closed when the function ends and the `file` variable is *dropped*.

This 'throwing away errors' thing is getting too much of a habit. You do not want to put this code into a function, knowing that it could so easily crash the whole program. So now we have to talk about exactly what `File::open` returns. If `Option` is a value that may contain something or nothing, then `Result` is a value that may contain something or an error. They both understand `unwrap` (and its cousin `expect`) but they are quite different. `Result` is defined by *two* type parameters, for the `Ok` value and the `Err` value. The `Result` 'box' has two compartments, one labelled `Ok` and the other `Err`.

```rust
fn good_or_bad(good: bool) -> Result<i32,String> {
    if good {
        Ok(42)
    } else {
        Err("bad".to_string())
    }
}

fn main() {
    println!("{:?}",good_or_bad(true));
    //Ok(42)
    println!("{:?}",good_or_bad(false));
    //Err("bad")

    match good_or_bad(true) {
        Ok(n) => println!("Cool, I got {}",n),
        Err(e) => println!("Huh, I just got {}",e)
    }
    // Cool, I got 42

}
```

(The actual 'error' type is arbitrary - a lot of people use strings until they are comfortable with Rust error types.) It's a convenient way to *either* return one value *or* another.

This version of the file reading function does not crash. It returns a `Result` and it is the *caller* who must decide how to handle the error.

```
// file2.rs
use std::env;
use std::fs::File;
use std::io::Read;
use std::io;

fn read_to_string(filename: &str) -> Result<String,io::Error> {
    let mut file = match File::open(&filename) {
        Ok(f) => f,
        Err(e) => return Err(e),
    };
    let mut text = String::new();
    match file.read_to_string(&mut text) {
        Ok(_) => Ok(text),
        Err(e) => Err(e),
    }
}

fn main() {
    let file = env::args().nth(1).expect("please supply a filename");

    let text = read_to_string(&file).expect("bad file man!");

    println!("file had {} bytes", text.len());
}
```

The first match safely extracts the value from `Ok`, which becomes the value of the match. If it's `Err` it returns the error, rewrapped as an `Err`.

The second match returns the string, wrapped up as an `Ok`, otherwise (again) the error. The actual value in the `Ok` is unimportant, so we ignore it with `_`.

This is not so pretty; when most of a function is error handling, then the 'happy path' gets lost. Go tends to have this problem, with lots of explicit early returns, or just *ignoring errors*. (That is, by the way, the closest thing to evil in the Rust universe.)

Fortunately, there is a shortcut.

The `std::io` module defines a type alias `io::Result<T>` which is exactly the same as `Result<T,io::Error>` and easier to type.

```
fn read_to_string(filename: &str) -> io::Result<String> {
    let mut file = File::open(&filename)?;
    let mut text = String::new();
    file.read_to_string(&mut text)?;
    Ok(text)
}
```

That `?` operator does almost exactly what the match on `File::open` does; if the result was an error, then it will immediately return that error. Otherwise, it returns the `Ok` result. At the end, we still need to wrap up the string as a result.

2017 was a good year for Rust, and `?` was one of the cool things that became stable. You will still see the macro `try!` used in older code:

```
fn read_to_string(filename: &str) -> io::Result<String> {
    let mut file = try!(File::open(&filename));
    let mut text = String::new();
    try!(file.read_to_string(&mut text));
    Ok(text)
}
```

In summary, it's possible to write perfectly safe Rust that isn't ugly, without needing exceptions.

Structs, Enums and Matching

Rust likes to Move It, Move It

I'd like to move back a little, and show you something surprising:

```
// move1.rs
fn main() {
    let s1 = "hello dolly".to_string();
    let s2 = s1;
    println!("s1 {}", s1);
}
```

And we get the following error:

```
error[E0382]: use of moved value: `s1`
 --> move1.rs:5:22
  |
4 |     let s2 = s1;
  |         -- value moved here
5 |     println!("s1 {}", s1);
  |                      ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`,
  which does not implement the `Copy` trait
```

Rust has different behaviour than other languages. In a language where variables are always references (like Java or Python), `s2` becomes yet another reference to the string object referenced by `s1`. In C++, `s1` is a value, and it is *copied* to `s2`. But Rust moves the value. It doesn't see strings as copyable ("does not implement the Copy trait").

We would not see this with 'primitive' types like numbers, since they are just values; they are allowed to be copyable because they are cheap to copy. But `String` has allocated memory containing "Hello dolly", and copying will involve allocating some more memory and copying the characters. Rust will not do this silently.

Consider a `String` containing the whole text of 'Moby-Dick'. It's not a big struct, just has the address in memory of the text, its size, and how big the allocated block is. Copying this is going to be expensive, because that memory is allocated on the heap and the copy will need its own allocated block.

```
String
| addr | ---------> Call me Ishmael.....
| size |                  |
| cap  |                  |
                          |
&str                      |
| addr | -------------------|
| size |

f64
| 8 bytes |
```

The second value is a string slice (`&str`) which refers to the same memory as the string, with a size - just the guy's name. Cheap to copy!

The third value is an `f64` - just 8 bytes. It does not refer to any other memory, so it's just as cheap to copy as to move.

`Copy` values are only defined by their representation in memory, and when Rust copies, it just copies those bytes elsewhere. Similarly, a non-`Copy` value is also *just moved*. There is no cleverness in copying and moving, unlike in C++.

Re-writing with a function call reveals exactly the same error:

```rust
// move2.rs

fn dump(s: String) {
    println!("{}", s);
}

fn main() {
    let s1 = "hello dolly".to_string();
    dump(s1);
    println!("s1 {}", s1); // <---error: 'value used here after move'
}
```

Here, you have a choice. You may pass a reference to that string, or explicitly copy it using its `clone` method. Generally, the first is the better way to go.

```
fn dump(s: &String) {
    println!("{}", s);
}

fn main() {
    let s1 = "hello dolly".to_string();
    dump(&s1);
    println!("s1 {}", s1);
}
```

The error goes away. But you'll rarely see a plain `String` reference like this, since to pass a string literal is really ugly *and* involves creating a temporary string.

```
    dump(&"hello world".to_string());
```

So altogether the best way to declare that function is:

```
fn dump(s: &str) {
    println!("{}", s);
}
```

And then both `dump(&s1)` and `dump("hello world")` work properly. (Here `Deref` coercion kicks in and Rust will convert `&String` to `&str` for you.)

To summarise, assignment of a non-Copy value moves the value from one location to another. Otherwise, Rust would be forced to *implicitly* do a copy and break its promise to make allocations explicit.

Scope of Variables

So, the rule of thumb is to prefer to keep references to the original data - to 'borrow' it.

But a reference must *not* outlive the owner!

First, Rust is a *block-scoped* language. Variables only exist for the duration of their block:

```rust
{
    let a = 10;
    let b = "hello";
    {
        let c = "hello".to_string();
        // a,b and c are visible
    }
    // the string c is dropped
    // a,b are visible
    for i in 0..a {
        let b = &b[1..];
        // original b is no longer visible - it is shadowed.
    }
    // the slice b is dropped
    // i is _not_ visible!
}
```

Loop variables (like `i`) are a little different, they are only visible in the loop block. It is not an error to create a new variable using the same name ('shadowing') but it can be confusing.

When a variable 'goes out of scope' then it is *dropped*. Any memory used is reclaimed, and any other *resources* owned by that variable are given back to the system - for instance, dropping a `File` closes it. This is a Good Thing. Unused resources are reclaimed immediately when not needed.

(A further Rust-specific issue is that a variable may appear to be in scope, but its value has moved.)

Here a reference `rs1` is made to a value `tmp` which only lives for the duration of its block:

```rust
01 // ref1.rs
02 fn main() {
03    let s1 = "hello dolly".to_string();
04    let mut rs1 = &s1;
05    {
06        let tmp = "hello world".to_string();
07        rs1 = &tmp;
08    }
09    println!("ref {}", rs1);
10 }
```

We borrow the value of `s1` and then borrow the value of `tmp`. But `tmp`'s value does not exist outside that block!

```
error: `tmp` does not live long enough
  --> ref1.rs:8:5
   |
7  |         rs1 = &tmp;
   |                --- borrow occurs here
8  |     }
   |     ^ `tmp` dropped here while still borrowed
9  |     println!("ref {}", rs1);
10 | }
   | - borrowed value needs to live until here
```

Where is `tmp`? Gone, dead, gone back to the Big Heap in the Sky: *dropped*. Rust is here saving you from the dreaded 'dangling pointer' problem of C - a reference that points to stale data.

Tuples

It's sometimes very useful to return multiple values from a function. Tuples are a convenient solution:

```
// tuple1.rs

fn add_mul(x: f64, y: f64) -> (f64,f64) {
    (x + y, x * y)
}

fn main() {
    let t = add_mul(2.0,10.0);

    // can debug print
    println!("t {:?}", t);

    // can 'index' the values
    println!("add {} mul {}", t.0,t.1);

    // can _extract_ values
    let (add,mul) = t;
    println!("add {} mul {}", add,mul);
}
// t (12, 20)
// add 12 mul 20
// add 12 mul 20
```

Tuples may contain *different* types, which is the main difference from arrays.

```
let tuple = ("hello", 5, 'c');

assert_eq!(tuple.0, "hello");
assert_eq!(tuple.1, 5);
assert_eq!(tuple.2, 'c');
```

They appear in some `Iterator` methods. `enumerate` is like the Python generator of the same name:

```
    for t in ["zero","one","two"].iter().enumerate() {
        print!(" {} {};",t.0,t.1);
    }
    //  0 zero; 1 one; 2 two;
```

`zip` combines two iterators into a single iterator of tuples containing the values from both:

```
    let names = ["ten","hundred","thousand"];
    let nums = [10,100,1000];
    for p in names.iter().zip(nums.iter()) {
        print!(" {} {};", p.0,p.1);
    }
    //  ten 10; hundred 100; thousand 1000;
```

Structs

Tuples are convenient, but saying `t.1` and keeping track of the meaning of each part is tedious for anything that isn't straightforward.

Rust *structs* contain named *fields*:

```
// struct1.rs

struct Person {
    first_name: String,
    last_name: String
}

fn main() {
    let p = Person {
        first_name: "John".to_string(),
        last_name: "Smith".to_string()
    };
    println!("person {} {}", p.first_name,p.last_name);
}
```

The values of a struct will be placed next to each other in memory, although you should not assume any particular memory layout, since the compiler will organize the memory for efficiency, not size, and there may be padding.

Initializing this struct is a bit clumsy, so we want to move the construction of a `Person` into its own function. This function can be made into an *associated function* of `Person` by putting it into a `impl` block:

```
// struct2.rs

struct Person {
    first_name: String,
    last_name: String
}

impl Person {

    fn new(first: &str, name: &str) -> Person {
        Person {
            first_name: first.to_string(),
            last_name: name.to_string()
        }
    }

}

fn main() {
    let p = Person::new("John","Smith");
    println!("person {} {}", p.first_name,p.last_name);
}
```

There is nothing magic or reserved about the name `new` here. Note that it's accessed using a C++-like notation using double-colon `::`.

Here's a `Person` *method*, that takes a *reference self* argument:

```
impl Person {
    ...

    fn full_name(&self) -> String {
        format!("{} {}", self.first_name, self.last_name)
    }

}
...
    println!("fullname {}", p.full_name());
// fullname John Smith
```

The `self` is used explicitly and is passed as a reference. (You can think of `&self` as short for `self: &Person`.)

The keyword `Self` refers to the struct type - you can mentally substitute `Person` for `Self` here:

```
    fn copy(&self) -> Self {
        Self::new(&self.first_name,&self.last_name)
    }
```

Methods may allow the data to be modified using a *mutable self* argument:

```
    fn set_first_name(&mut self, name: &str) {
        self.first_name = name.to_string();
    }
```

And the data will *move* into the method when a plain self argument is used:

```
    fn to_tuple(self) -> (String,String) {
        (self.first_name, self.last_name)
    }
```

(Try that with `&self` - structs will not let go of their data without a fight!)

Note that after `v.to_tuple()` is called, then `v` has moved and is no longer available.

To summarize:

- no `self` argument: you can associate functions with structs, like the `new` "constructor".
- `&self` argument: can use the values of the struct, but not change them
- `&mut self` argument: can modify the values
- `self` argument: will consume the value, which will move.

If you try to do a debug dump of a `Person`, you will get an informative error:

```
error[E0277]: the trait bound `Person: std::fmt::Debug` is not satisfied
  --> struct2.rs:23:21
   |
23 |     println!("{:?}", p);
   |                     ^ the trait `std::fmt::Debug` is not implemented for `Person`
   |
   = note: `Person` cannot be formatted using `:?`; if it is defined in your crate,
    add `#[derive(Debug)]` or manually implement it
   = note: required by `std::fmt::Debug::fmt`
```

The compiler is giving advice, so we put `#[derive(Debug)]` in front of `Person`, and now there is sensible output:

```
Person { first_name: "John", last_name: "Smith" }
```

The *directive* makes the compiler generate a `Debug` implementation, which is very helpful. It's good practice to do this for your structs, so they can be printed out (or written as a string using `format!`). (Doing so *by default* would be very un-Rustlike.)

Here is the final little program:

```
// struct4.rs
use std::fmt;

#[derive(Debug)]
struct Person {
    first_name: String,
    last_name: String
}

impl Person {

    fn new(first: &str, name: &str) -> Person {
        Person {
            first_name: first.to_string(),
            last_name: name.to_string()
        }
    }

    fn full_name(&self) -> String {
        format!("{} {}",self.first_name, self.last_name)
    }

    fn set_first_name(&mut self, name: &str) {
        self.first_name = name.to_string();
    }

    fn to_tuple(self) -> (String,String) {
        (self.first_name, self.last_name)
    }
}

fn main() {
    let mut p = Person::new("John","Smith");

    println!("{:?}", p);

    p.set_first_name("Jane");

    println!("{:?}", p);

    println!("{:?}", p.to_tuple());
    // p has now moved.

}
// Person { first_name: "John", last_name: "Smith" }
// Person { first_name: "Jane", last_name: "Smith" }
// ("Jane", "Smith")
```

Lifetimes Start to Bite

Usually structs contain values, but often they also need to contain references. Say we want to put a string slice, not a string value, in a struct.

```
// life1.rs

#[derive(Debug)]
struct A {
    s: &str
}

fn main() {
    let a = A { s: "hello dammit" };

    println!("{:?}", a);
}
```

```
error[E0106]: missing lifetime specifier
 --> life1.rs:5:8
  |
5 |     s: &str
  |        ^ expected lifetime parameter
```

To understand the complaint, you have to see the problem from the point of view of Rust. It will not allow a reference to be stored without knowing its lifetime. All references are borrowed from some value, and all values have lifetimes. The lifetime of a reference cannot be longer than the lifetime of that value. Rust cannot allow a situation where that reference could suddenly become invalid.

Now, string slices borrow from *string literals* like "hello" or from `String` values. String literals exist for the duration of the whole program, which is called the 'static' lifetime.

So this works - we assure Rust that the string slice always refers to such static strings:

```
// life2.rs

#[derive(Debug)]
struct A {
    s: &'static str
}

fn main() {
    let a = A { s: "hello dammit" };

    println!("{:?}", a);
}
// A { s: "hello dammit" }
```

It is not the most *pretty* notation, but sometimes ugliness is the necessary price of being precise.

This can also be used to specify a string slice that is returned from a function:

```
fn how(i: u32) -> &'static str {
    match i {
    0 => "none",
    1 => "one",
    _ => "many"
    }
}
```

That works for the special case of static strings, but this is very restrictive.

However we can specify that the lifetime of the reference is *at least as long* as that of the struct itself.

```
// life3.rs

#[derive(Debug)]
struct A <'a> {
    s: &'a str
}

fn main() {
    let s = "I'm a little string".to_string();
    let a = A { s: &s };

    println!("{:?}", a);
}
```

Lifetimes are conventionally called 'a','b',etc but you could just as well called it 'me' here.

After this point, our `a` struct and the `s` string are bound by a strict contract: `a` borrows from `s`, and cannot outlive it.

With this struct definition, we would like to write a function that returns an `A` value:

```
fn makes_a() -> A {
    let string = "I'm a little string".to_string();
    A { s: &string }
}
```

But `A` needs a lifetime - "expected lifetime parameter":

```
  = help: this function's return type contains a borrowed value,
   but there is no value for it to be borrowed from
  = help: consider giving it a 'static lifetime
```

`rustc` is giving advice, so we follow it:

```
fn makes_a() -> A<'static> {
    let string = "I'm a little string".to_string();
    A { s: &string }
}
```

And now the error is

```
8 |     A { s: &string }
  |            ^^^^^^ does not live long enough
9 | }
  | - borrowed value only lives until here
```

There is no way that this could safely work, because `string` will be dropped when the function ends, and no reference to `string` can outlast it.

You can usefully think of lifetime parameters as being part of the type of a value.

Sometimes it seems like a good idea for a struct to contain a value *and* a reference that borrows from that value. It's basically impossible because structs must be *moveable*, and any move will invalidate the reference. It isn't necessary to do this - for instance, if your struct has a string field, and needs to provide slices, then it could keep indices and have a method to generate the actual slices.

Traits

Please note that Rust does not spell `struct` *class*. The keyword `class` in other languages is so overloaded with meaning that it effectively shuts down original thinking.

Let's put it like this: Rust structs cannot *inherit* from other structs; they are all unique types. There is no *sub-typing*. They are dumb data.

So how *does* one establish relationships between types? This is where *traits* come in.

`rustc` often talks about `implementing X trait` and so it's time to talk about traits properly.

Here's a little example of defining a trait and *implementing* it for a particular type.

```rust
// trait1.rs

trait Show {
    fn show(&self) -> String;
}

impl Show for i32 {
    fn show(&self) -> String {
        format!("four-byte signed {}", self)
    }
}

impl Show for f64 {
    fn show(&self) -> String {
        format!("eight-byte float {}", self)
    }
}

fn main() {
    let answer = 42;
    let maybe_pi = 3.14;
    let s1 = answer.show();
    let s2 = maybe_pi.show();
    println!("show {}", s1);
    println!("show {}", s2);
}
// show four-byte signed 42
// show eight-byte float 3.14
```

It's pretty cool; we have *added a new method* to both `i32` and `f64`!

Getting comfortable with Rust involves learning the basic traits of the standard library (they tend to hunt in packs.)

`Debug` is very common. We gave `Person` a default implementation with the convenient `#[derive(Debug)]`, but say we want a `Person` to display as its full name:

```rust
use std::fmt;

impl fmt::Debug for Person {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "{}", self.full_name())
    }
}
...
    println!("{:?}", p);
    // John Smith
```

`write!` is a very useful macro - here `f` is anything that implements `Write`. (This would also work with a `File` - or even a `String`.)

`Display` controls how values are printed out with "{}" and is implemented just like `Debug`. As a useful side-effect, `ToString` is automatically implemented for anything implementing `Display`. So if we implement `Display` for `Person`, then `p.to_string()` also works.

`Clone` defines the method `clone`, and can simply be defined with "#[derive(Clone)]" if all the fields themselves implement `Clone`.

Example: iterator over floating-point range

We have met ranges before (`0..n`) but they don't work for floating-point values. (You can *force* this but you'll end up with a step of 1.0 which is uninteresting.)

Recall the informal definition of an iterator; it is an struct with a `next` method which may return `Some`-thing or `None`. In the process, the iterator itself gets modified, it keeps the state for the iteration (like next index and so forth.) The data that is being iterated over doesn't change usually, (But see `Vec::drain` for an interesting iterator that does modify its data.)

And here is the formal definition: the Iterator trait.

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    ...
}
```

Here we meet an associated type of the `Iterator` trait. This trait must work for any type, so you must specify that return type somehow. The method `next` can then be written without using a particular type - instead it refers to that type parameter's `Item` via `Self`.

The iterator trait for `f64` is written `Iterator<Item=f64>`, which can be read as "an Iterator with its associated type Item set to f64".

The `...` refers to the *provided methods* of `Iterator`. You only need to define `Item` and `next`, and the provided methods are defined for you.

```rust
// trait3.rs

struct FRange {
    val: f64,
    end: f64,
    incr: f64
}

fn range(x1: f64, x2: f64, skip: f64) -> FRange {
    FRange {val: x1, end: x2, incr: skip}
}

impl Iterator for FRange {
    type Item = f64;

    fn next(&mut self) -> Option<Self::Item> {
        let res = self.val;
        if res >= self.end {
            None
        } else {
            self.val += self.incr;
            Some(res)
        }
    }
}


fn main() {
    for x in range(0.0, 1.0, 0.1) {
        println!("{} ", x);
    }
}
```

And the rather messy looking result is

```
0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6
0.7
0.7999999999999999
0.8999999999999999
0.9999999999999999
```

This is because 0.1 is not precisely representable as a float, so a little formatting help is needed. Replace the `println!` with this

```rust
println!("{:.1} ", x);
```

And we get cleaner output (this format means 'one decimal after dot'.)

All of the default iterator methods are available, so we can collect these values into a vector, map them, and so forth.

```rust
    let v: Vec<f64> = range(0.0, 1.0, 0.1).map(|x| x.sin()).collect();
```

Generic Functions

We want a function which will dump out any value that implements `Debug`. Here is a first attempt at a generic function, where we can pass a reference to *any* type of value. `T` is a type parameter, which needs to be declared just after the function name:

```
fn dump<T> (value: &T) {
    println!("value is {:?}",value);
}

let n = 42;
dump(&n);
```

However, Rust clearly knows nothing about this generic type `T`:

```
error[E0277]: the trait bound `T: std::fmt::Debug` is not satisfied
...
   = help: the trait `std::fmt::Debug` is not implemented for `T`
   = help: consider adding a `where T: std::fmt::Debug` bound
```

For this to work, Rust needs to be told that `T` does in fact implement `Debug`!

```
fn dump<T> (value: &T)
where T: std::fmt::Debug {
    println!("value is {:?}",value);
}

let n = 42;
dump(&n);
// value is 42
```

Rust generic functions need *trait bounds* on types - we are saying here that "T is any type that implements Debug". `rustc` is being very helpful, and suggests exactly what bound needs to be provided.

Now that Rust knows the trait bounds for `T`, it can give you sensible compiler messages:

```
struct Foo {
    name: String
}

let foo = Foo{name: "hello".to_string()};

dump(&foo)
```

And the error is "the trait `std::fmt::Debug` is not implemented for `Foo`".

Functions are already generic in dynamic languages because values carry their actual type around, and the type checking happens at run-time - or fails miserably. For larger programs, we really do want to know about problems at compile-time rather! Rather than sitting down calmly with compiler errors, a programmer in these languages has to deal with problems that only show up when the program is running. Murphy's Law then implies that these problems will tend to happen at the most inconvenient/disastrous time.

The operation of squaring a number is generic: `x*x` will work for integers, floats and generally for anything that knows about the multiplication operator `*`. But what are the type bounds?

```
// gen1.rs

fn sqr<T> (x: T) -> T {
    x * x
}

fn main() {
    let res = sqr(10.0);
    println!("res {}",res);
}
```

The first problem is that Rust does not know that `T` can be multiplied:

```
error[E0369]: binary operation `*` cannot be applied to type `T`
 --> gen1.rs:4:5
  |
4 |     x * x
  |     ^
  |
note: an implementation of `std::ops::Mul` might be missing for `T`
 --> gen1.rs:4:5
  |
4 |     x * x
  |     ^
```

Following the advice of the compiler, let's constrain that type parameter using that trait, which is used to implement the multiplication operator `*`:

```
fn sqr<T> (x: T) -> T
where T: std::ops::Mul {
    x * x
}
```

Which still doesn't work:

```
rror[E0308]: mismatched types
 --> gen2.rs:6:5
  |
6 |     x * x
  |     ^^^ expected type parameter, found associated type
  |
  = note: expected type `T`
  = note:    found type `<T as std::ops::Mul>::Output`
```

What `rustc` is saying that the type of `x*x` is the associated type `T::Output`, not `T`. There's actually no reason that the type of `x*x` is the same as the type of `x`, e.g. the dot product of two vectors is a scalar.

```
fn sqr<T> (x: T) -> T::Output
where T: std::ops::Mul {
    x * x
}
```

and now the error is:

```
error[E0382]: use of moved value: `x`
 --> gen2.rs:6:7
  |
6 |     x * x
  |     - ^ value used here after move
  |     |
  |     value moved here
  |
  = note: move occurs because `x` has type `T`, which does not implement the `Copy` trait
```

So, we need to constrain the type even further!

```
fn sqr<T> (x: T) -> T::Output
where T: std::ops::Mul + Copy {
    x * x
}
```

And that (finally) works. Calmly listening to the compiler will often get you closer to the magic point when ... things compile cleanly.

It *is* a bit simpler in C++:

```
template <typename T>
T sqr(x: T) {
    return x * x;
}
```

but (to be honest) C++ is adopting cowboy tactics here. C++ template errors are famously bad, because all the compiler knows (ultimately) is that some operator or method is not defined. The C++ committee knows this is a problem and so they are working toward concepts, which are pretty much like trait-constrained type parameters in Rust.

Rust generic functions may look a bit overwhelming at first, but being explicit means you will know exactly what kind of values you can safely feed it, just by looking at the definition.

These functions are called *monomorphic*, in constrast to *polymorphic*. The body of the function is compiled separately for each unique type. With polymorphic functions, the same machine code works with each matching type, dynamically *dispatching* the correct method.

Monomorphic produces faster code, specialized for the particular type, and can often be *inlined*. So when `sqr(x)` is seen, it's effectively replaced with `x*x`. The downside is that large generic functions produce a lot of code, for each type used, which can result in *code bloat*. As always, there are trade-offs; an experienced person learns to make the right choice for the job.

Simple Enums

Enums are types which have a few definite values. For instance, a direction has only four possible values.

```
enum Direction {
    Up,
    Down,
    Left,
    Right
}
...
    // `start` is type `Direction`
    let start = Direction::Left;
```

They can have methods defined on them, just like structs. The `match` expression is the basic way to handle `enum` values.

```
impl Direction {
    fn as_str(&self) -> &'static str {
        match *self { // *self has type Direction
            Direction::Up => "Up",
            Direction::Down => "Down",
            Direction::Left => "Left",
            Direction::Right => "Right"
        }
    }
}
```

Punctuation matters. Note that `*` before `self`. It's easy to forget, because often Rust will assume it (we said `self.first_name`, not `(*self).first_name`). However, matching is a more exact business. Leaving it out would give a whole spew of messages, which boil down to this type mismatch:

```
   = note: expected type `&Direction`
   = note:    found type `Direction`
```

This is because `self` has type `&Direction`, so we have to throw in the `*` to *deference* the type.

Like structs, enums can implement traits, and our friend `#[derive(Debug)]` can be added to `Direction`:

```
        println!("start {:?}",start);
        // start Left
```

So that `as_str` method isn't really necessary, since we can always get the name from `Debug`. (But `as_str` does *not allocate*, which may be important.)

You should not assume any particular ordering here - there's no implied integer 'ordinal' value.

Here's a method which defines the 'successor' of each `Direction` value. The very handy *wildcard use* temporarily puts the enum names into the method context:

```
fn next(&self) -> Direction {
    use Direction::*;
    match *self {
        Up => Right,
        Right => Down,
        Down => Left,
        Left => Up
    }
}
...

let mut d = start;
for _ in 0..8 {
    println!("d {:?}", d);
    d = d.next();
}
// d Left
// d Up
// d Right
// d Down
// d Left
// d Up
// d Right
// d Down
```

So this will cycle endlessly through the various directions in this particular, arbitrary, order. It is (in fact) a very simple *state machine*.

These enum values can't be compared:

```
assert_eq!(start, Direction::Left);

error[E0369]: binary operation `==` cannot be applied to type `Direction`
  --> enum1.rs:42:5
   |
42 |     assert_eq!(start, Direction::Left);
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
note: an implementation of `std::cmp::PartialEq` might be missing for `Direction`
  --> enum1.rs:42:5
```

The solution is to say `#[derive(Debug,PartialEq)]` in front of `enum Direction`.

This is an important point - Rust user-defined types start out fresh and unadorned. You give them sensible default behaviours by implementing the common traits. This applies also to structs - if you ask for Rust to derive `PartialEq` for a struct it will do the sensible thing, assume that all fields implement it and build up a comparison. If this isn't so, or you want to redefine equality, then you are free to define `PartialEq` explicitly.

Rust does 'C style enums' as well:

```
// enum2.rs

enum Speed {
    Slow = 10,
    Medium = 20,
    Fast = 50
}

fn main() {
    let s = Speed::Slow;
    let speed = s as u32;
    println!("speed {}", speed);
}
```

They are initialized with an integer value, and can be converted into that integer with a type cast.

You only need to give the first name a value, and thereafter the value goes up by one each time:

```
enum Difficulty {
    Easy = 1,
    Medium,  // is 2
    Hard   // is 3
}
```

By the way, 'name' is too vague, like saying 'thingy' all the time. The proper term here is *variant* - `Speed` has variants `Slow`, `Medium` and `Fast`.

These enums *do* have a natural ordering, but you have to ask nicely. After placing `#[derive(PartialEq,PartialOrd)]` in front of `enum Speed`, then it's indeed true that `Speed::Fast > Speed::Slow` and `Speed::Medium != Speed::Slow`.

Enums in their Full Glory

Rust enums in their full form are like C unions on steroids, like a Ferrari compared to a Fiat Uno. Consider the problem of storing different values in a type-safe way.

```
// enum3.rs

#[derive(Debug)]
enum Value {
    Number(f64),
    Str(String),
    Bool(bool)
}

fn main() {
    use Value::*;
    let n = Number(2.3);
    let s = Str("hello".to_string());
    let b = Bool(true);

    println!("n {:?} s {:?} b {:?}", n,s,b);
}
// n Number(2.3) s Str("hello") b Bool(true)
```

Again, this enum can only contain *one* of these values; its size will be the size of the largest variant.

So far, not really a supercar, although it's cool that enums know how to print themselves out. But they also know how *what kind* of value they contain, and *that* is the superpower of `match`:

```
fn eat_and_dump(v: Value) {
    use Value::*;
    match v {
        Number(n) => println!("number is {}", n),
        Str(s) => println!("string is '{}'", s),
        Bool(b) => println!("boolean is {}", b)
    }
}
....
eat_and_dump(n);
eat_and_dump(s);
eat_and_dump(b);
//number is 2.3
//string is 'hello'
//boolean is true
```

(And that's what `Option` and `Result` are - enums.)

We like this `eat_and_dump` function, but we want to pass the value as a reference, because currently a move takes place and the value is 'eaten':

```
fn dump(v: &Value) {
    use Value::*;
    match *v {  // type of *v is Value
        Number(n) => println!("number is {}", n),
        Str(s) => println!("string is '{}'", s),
        Bool(b) => println!("boolean is {}", b)
    }
}

error[E0507]: cannot move out of borrowed content
  --> enum3.rs:12:11
   |
12 |     match *v {
   |           ^^ cannot move out of borrowed content
13 |     Number(n) => println!("number is {}",n),
14 |     Str(s) => println!("string is '{}'",s),
   |         - hint: to prevent move, use `ref s` or `ref mut s`
```

There are things you cannot do with borrowed references. Rust is not letting you *extract* the string contained in the original value. It did not complain about `Number` because it's happy to copy `f64`, but `String` does not implement `Copy`.

I mentioned earlier that `match` is picky about *exact* types; here we follow the hint and things will work; now we are just borrowing a reference to that contained string.

```
fn dump(v: &Value) {
    use Value::*;
    match *v {
        Number(n) => println!("number is {}", n),
        Str(ref s) => println!("string is '{}'", s),
        Bool(b) => println!("boolean is {}", b)
    }
}
    ....

    dump(&s);
    // string is 'hello'
```

Before we move on, filled with the euphoria of a successful Rust compilation, let's pause a little. `rustc` is unusually good at generating errors that have enough context for a human to *fix* the error without necessarily *understanding* the error.

The issue is a combination of the exactness of matching, with the determination of the borrow checker to foil any attempt to break the Rules. One of those Rules is that you cannot yank out a value which belongs to some owning type. Some knowledge of C++ is a hindrance here, since C++ will copy its way out of the problem, whether that copy even *makes sense*. You will get exactly the same error if you try to pull out a string from a vector, say with `*v.get(0).unwrap()` (`*` because indexing returns references.) It will simply not let you do this. (Sometimes `clone` isn't such a bad solution to this.)

(By the way, `v[0]` does not work for non-copyable values like strings for precisely this reason. You must either borrow with `&v[0]` or clone with `v[0].clone()`)

As for `match`, you can see `Str(s) =>` as short for `Str(s: String) =>`. A local variable (often called a *binding*) is created. Often that inferred type is cool, when you eat up a value and extract its contents. But here we really needed is `s: &String`, and the `ref` is a hint that ensures this: we just want to borrow that string.

Here we do want to extract that string, and don't care about the enum value afterwards. `_` as usual will match anything.

```
impl Value {
    fn to_str(self) -> Option<String> {
        match self {
        Value::Str(s) => Some(s),
        _ => None
        }
    }
}
    ...
    println!("s? {:?}", s.to_str());
    // s? Some("hello")
    // println!("{:?}", s) // error! s has moved...
```

Naming matters - this is called `to_str`, not `as_str`. You can write a method that just borrows that string as an `Option<&String>` (The reference will need the same lifetime as the enum value.) But you would not call it `to_str`.

You can write `to_str` like this - it is completely equivalent:

```
    fn to_str(self) -> Option<String> {
        if let Value::Str(s) = self {
            Some(s)
        } else {
            None
        }
    }
```

More about Matching

Recall that the values of a tuple can be extracted with '()':

```
    let t = (10,"hello".to_string());
    ...
    let (n,s) = t;
    // t has been moved. It is No More
    // n is i32, s is String
```

This is a special case of *destructuring*; we have some data and wish to either pull it apart (like here) or just borrow its values. Either way, we get the parts of a structure.

The syntax is like that used in `match`. Here we are explicitly borrowing the values.

```
let (ref n,ref s) = t;
// n and s are borrowed from t. It still lives!
// n is &i32, s is &String
```

Destructuring works with structs as well:

```
struct Point {
    x: f32,
    y: f32
}

let p = Point{x:1.0,y:2.0};
...
let Point{x,y} = p;
// p still lives, since x and y can and will be copied
// both x and y are f32
```

Time to revisit `match` with some new patterns. The first two patterns are exactly like `let` destructuring - it only matches tuples with first element zero, but *any* string; the second adds an `if` so that it only matches `(1,"hello")`. Finally, just a variable matches *anything*. This is useful if the `match` applies to an expression and you don't want to bind a variable to that expression. `_` works like a variable but is ignored. It's a common way to finish off a `match`.

```
fn match_tuple(t: (i32,String)) {
    let text = match t {
        (0, s) => format!("zero {}", s),
        (1, ref s) if s == "hello" => format!("hello one!"),
        tt => format!("no match {:?}", tt),
        // or say _ => format!("no match") if you're not interested in the value
     };
    println!("{}", text);
}
```

Why not just match against `(1,"hello")`? Matching is an exact business, and the compiler will complain:

```
  = note: expected type `std::string::String`
  = note:    found type `&'static str`
```

Why do we need `ref s`? It's a slightly obscure gotcha (look up the E00008 error) where if you have an *if guard* you need to borrow, since the if guard happens in a different context, a move will take place otherwise. It's a case of the implementation leaking ever so slightly.

If the type *was* `&str` then we match it directly:

```
match (42,"answer") {
    (42,"answer") => println!("yes"),
    _ => println!("no")
};
```

What applies to `match` applies to `if let`. This is a cool example, since if we get a `Some`, we can match inside it and only extract the string from the tuple. So it isn't necessary to have nested `if let` statements here. We use `_` because we aren't interested in the first part of the tuple.

```
let ot = Some((2,"hello".to_string());

if let Some((_,ref s)) = ot {
    assert_eq!(s, "hello");
}
// we just borrowed the string, no 'destructive destructuring'
```

An interesting problem happens when using `parse` (or any function which needs to work out its return type from context)

```
if let Ok(n) = "42".parse() {
    ...
}
```

So what's the type of `n`? You have to give a hint somehow - what kind of integer? Is it even an integer?

```
if let Ok(n) = "42".parse::<i32>() {
    ...
}
```

This somewhat non-elegant syntax is called the 'turbofish operator'.

If you are in a function returning `Result`, then the question-mark operator provides a much more elegant solution:

```
let n: i32 = "42".parse()?;
```

However, the parse error needs to be convertible to the error type of the `Result`, which is a topic we'll take up later when discussing error handling.

Closures

A great deal of Rust's power comes from *closures*. In their simplest form, they act like shortcut functions:

```
let f = |x| x * x;

let res = f(10);

println!("res {}", res);
// res 100
```

There are no explicit types in this example - everything is deduced, starting with the integer literal 10.

We get an error if we call `f` on different types - Rust has already decided that `f` must be called on an integer type:

```
    let res = f(10);

    let resf = f(1.2);
  |
8 |     let resf = f(1.2);
  |                  ^^^ expected integral variable, found floating-point variable
  |
  = note: expected type `{integer}`
  = note:    found type `{float}`
```

So, the first call fixes the type of the argument `x`. It's equivalent to this function:

```
    fn f (x: i32) -> i32 {
        x * x
    }
```

But there's a big difference between functions and closures, *apart* from the need for explicit typing. Here we evaluate a linear function:

```
    let m = 2.0;
    let c = 1.0;

    let lin = |x| m*x + c;

    println!("res {} {}", lin(1.0), lin(2.0));
    // res 3 5
```

You cannot do this with the explicit `fn` form - it does not know about variables in the enclosing scope. The closure has *borrowed* `m` and `c` from its context.

Now, what's the type of `lin`? Only `rustc` knows. Under the hood, a closure is a *struct* that is callable ('implements the call operator'). It behaves as if it was written out like this:

```
struct MyAnonymousClosure1<'a> {
    m: &'a f64,
    c: &'a f64
}

impl <'a>MyAnonymousClosure1<'a> {
    fn call(&self, x: f64) -> f64 {
        self.m * x  + self.c
    }
}
```

The compiler is certainly being helpful, turning simple closure syntax into all that code! You do need to know that a closure is a *struct* and it *borrows* values from its environment. And that therefore it has a *lifetime*.

All closures are unique types, but they have traits in common. So even though we don't know the exact type, we know the generic constraint:

```
fn apply<F>(x: f64, f: F) -> f64
where F: Fn(f64)->f64  {
    f(x)
}
...
    let res1 = apply(3.0,lin);
    let res2 = apply(3.14, |x| x.sin());
```

In English: `apply` works for *any* type `T` such that `T` implements `Fn(f64)->f64` - that is, is a function which takes `f64` and returns `f64`.

After the call to `apply(3.0,lin)`, trying to access `lin` gives an interesting error:

```
    let l = lin;
error[E0382]: use of moved value: `lin`
  --> closure2.rs:22:9
   |
16 |     let res = apply(3.0,lin);
   |                         --- value moved here
...
22 |     let l = lin;
   |         ^ value used here after move
   |
   = note: move occurs because `lin` has type
    `[closure@closure2.rs:12:15: 12:26 m:&f64, c:&f64]`,
     which does not implement the `Copy` trait
```

That's it, `apply` ate our closure. And there's the actual type of the struct that `rustc` made up to implement it. Always thinking of closures as structs is helpful.

Calling a closure is a *method call*: the three kinds of function traits correspond to the three kinds of methods:

- `Fn` struct passed as `&self`
- `FnMut` struct passed as `&mut self`
- `FnOnce` struct passed as `self`

So it's possible for a closure to mutate its *captured* references:

```
    fn mutate<F>(mut f: F)
    where F: FnMut() {
        f()
    }
    let mut s = "world";
    mutate(|| s = "hello");
    assert_eq!(s, "hello");
```

Note that `mut` - `f` needs to be mutable for this to work.

However, you cannot escape the rules for borrowing. Consider this:

```
let mut s = "world";

// closure does a mutable borrow of s
let mut changer = || s = "world";

changer();
// does an immutable borrow of s
assert_eq!(s, "world");
```

Can't be done! The error is that we cannot borrow `s` in the assert statement, because it has been previously borrowed by the closure `changer` as mutable. As long as that closure lives, no other code can access `s`, so the solution is to control that lifetime by putting the closure in a limited scope:

```
let mut s = "world";
{
    let mut changer = || s = "world";
    changer();
}
assert_eq!(s, "world");
```

At this point, if you are used to languages like JavaScript or Lua, you may wonder at the complexity of Rust closures compared with how straightforward they are in those languages. This is the necessary cost of Rust's promise to not sneakily make any allocations. In JavaScript, the equivalent `mutate(function() {s = "hello";})` will always result in a dynamically allocated closure.

Sometimes you don't want a closure to borrow those variables, but instead *move* them.

```
    let name = "dolly".to_string();
    let age = 42;

    let c = move || {
        println!("name {} age {}", name,age);
    };

    c();

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

And the error at the last `println` is: "use of moved value: `name`". So one solution here - if we *did* want to keep `name` alive - is to move a cloned copy into the closure:

```
    let cname = name.to_string();
    let c = move || {
        println!("name {} age {}",cname,age);
    };
```

Why are moved closures needed? Because we might need to call them at a point where the original context no longer exists. A classic case is when creating a *thread*. A moved closure does not borrow, so does not have a lifetime.

A major use of closures is within iterator methods. Recall the `range` iterator we defined to go over a range of floating-point numbers. It's straightforward to operate on this (or any other iterator) using closures:

```
let sine: Vec<f64> = range(0.0,1.0,0.1).map(|x| x.sin()).collect();
```

`map` isn't defined on vectors (although it's easy enough to create a trait that does this), because then *every* map will create a new vector. This way, we have a choice. In this sum, no temporary objects are created:

```
let sum: f64 = range(0.0,1.0,0.1).map(|x| x.sin()).sum();
```

It will (in fact) be as fast as writing it out as an explicit loop! That performance guarantee would be impossible if Rust closures were as 'frictionless' as Javascript closures.

`filter` is another useful iterator method - it only lets through values that match a condition:

```
let tuples = [(10,"ten"),(20,"twenty"),(30,"thirty"),(40,"forty")];
let iter = tuples.iter().filter(|t| t.0 > 20).map(|t| t.1);

for name in iter {
    println!("{} ", name);
}
// thirty
// forty
```

The Three Kinds of Iterators

The three kinds correspond (again) to the three basic argument types. Assume we have a vector of `String` values. Here are the iterator types explicitly, and then *implicitly*, together with the actual type returned by the iterator.

```
for s in vec.iter() {...} // &String
for s in vec.iter_mut() {...} // &mut String
for s in vec.into_iter() {...} // String

// implicit!
for s in &vec {...} // &String
for s in &mut vec {...} // &mut String
for s in vec {...} // String
```

Personally I prefer being explicit, but it's important to understand both forms, and their implications.

`into_iter` *consumes* the vector and extracts its strings, and so afterwards the vector is no longer available - it has been moved. It's a definite gotcha for Pythonistas used to saying `for s in vec`!

So the implicit form `for s in &vec` is usually the one you want, just as `&T` is a good default in passing arguments to functions.

It's important to understand how the three kinds works because Rust relies heavily on type deduction - you won't often see explicit types in closure arguments. And this is a Good Thing, because it would be noisy if all those types were explicitly *typed out*. However, the price of this compact code is that you need to know what the implicit types actually are!

`map` takes whatever value the iterator returns and converts it into something else, but `filter` takes a *reference* to that value. In this case, we're using `iter` so the iterator item type is `&String`. Note that `filter` receives a reference to this type.

```
for n in vec.iter().map(|x: &String| x.len()) {...} // n is usize
....
}

for s in vec.iter().filter(|x: &&String| x.len() > 2) { // s is &String
...
}
```

When calling methods, Rust will derefence automatically, so the problem isn't obvious. But `|x: &&String| x == "one"|` will *not* work, because operators are more strict about type matching. `rustc` will complain that there is no such operator that compares `&&String` and `&str`. So you need an explicit deference to make that `&&String` into a `&String` which *does* match.

```
for s in vec.iter().filter(|x: &&String| *x == "one") {...}
// same as implicit form:
for s in vec.iter().filter(|x| *x == "one") {...}
```

If you leave out the explicit type, you can modify the argument so that the type of `s` is now `&String`:

```
for s in vec.iter().filter(|&x| x == "one")
```

And that's usually how you will see it written.

Structs with Dynamic Data

A most powerful technique is *a struct that contain references to itself*.

Here is the basic building block of a *binary tree*, expressed in C (everyone's favourite old relative with a frightening fondness for using power tools without protection.)

```
struct Node {
    const char *payload;
    struct Node *left;
    struct Node *right;
};
```

You can not do this by *directly* including `Node` fields, because then the size of `Node` depends on the size of `Node`... it just doesn't compute. So we use pointers to `Node` structs, since the size of a pointer is always known.

If `left` isn't `NULL`, the `Node` will have a left pointing to another node, and so moreorless indefinitely.

Rust does not do `NULL` (at least not *safely*) so it's clearly a job for `Option`. But you cannot just put a `Node` in that `Option`, because we don't know the size of `Node` (and so forth.) This is a job for `Box`, since it contains an allocated pointer to the data, and always has a fixed size.

So here's the Rust equivalent, using `type` to create an alias:

```
type NodeBox = Option<Box<Node>>;

#[derive(Debug)]
struct Node {
    payload: String,
    left: NodeBox,
    right: NodeBox
}
```

(Rust is forgiving in this way - no need for forward declarations.)

And a first test program:

```
impl Node {
    fn new(s: &str) -> Node {
        Node{payload: s.to_string(), left: None, right: None}
    }

    fn boxer(node: Node) -> NodeBox {
        Some(Box::new(node))
    }

    fn set_left(&mut self, node: Node) {
        self.left = Self::boxer(node);
    }

    fn set_right(&mut self, node: Node) {
        self.right = Self::boxer(node);
    }

}


fn main() {
    let mut root = Node::new("root");
    root.set_left(Node::new("left"));
    root.set_right(Node::new("right"));

    println!("arr {:#?}", root);
}
```

The output is surprisingly pretty, thanks to "{:#?}" ('#' means 'extended'.)

```
root Node {
    payload: "root",
    left: Some(
        Node {
            payload: "left",
            left: None,
            right: None
        }
    ),
    right: Some(
        Node {
            payload: "right",
            left: None,
            right: None
        }
    )
}
```

Now, what happens when `root` is dropped? All fields are dropped; if the 'branches' of the tree are dropped, they drop *their* fields and so on. `Box::new` may be the closest you will get to a `new` keyword, but we have no need for `delete` or `free`.

We must now work out a use for this tree. Note that strings can be ordered: 'bar' < 'foo', 'abba' > 'aardvark'; so-called 'alphabetical order'. (Strictly speaking, this is *lexical order*, since human languages are very diverse and have strange rules.)

Here is a method which inserts nodes in lexical order of the strings. We compare the new data to the current node - if it's less, then we try to insert on the left, otherwise try to insert on the right. There may be no node on the left, so then `set_left` and so forth.

```
fn insert(&mut self, data: &str) {
    if data < &self.payload {
        match self.left {
            Some(ref mut n) => n.insert(data),
            None => self.set_left(Self::new(data)),
        }
    } else {
        match self.right {
            Some(ref mut n) => n.insert(data),
            None => self.set_right(Self::new(data)),
        }
    }
}

...
fn main() {
    let mut root = Node::new("root");
    root.insert("one");
    root.insert("two");
    root.insert("four");

    println!("root {:#?}", root);
}
```

Note the `match` - we're pulling out a mutable reference to the box, if the `Option` is `Some`, and applying the `insert` method. Otherwise, we need to create a new `Node` for the left side and so forth. `Box` is a *smart* pointer; note that no 'unboxing' was needed to call `Node` methods on it!

And here's the output tree:

```
root Node {
    payload: "root",
    left: Some(
        Node {
            payload: "one",
            left: Some(
                Node {
                    payload: "four",
                    left: None,
                    right: None
                }
            ),
            right: None
        }
    ),
    right: Some(
        Node {
            payload: "two",
            left: None,
            right: None
        }
    )
}
```

The strings that are 'less' than other strings get put down the left side, otherwise the right side.

Time for a visit. This is *in-order traversal* - we visit the left, do something on the node, and then visit the right.

```
    fn visit(&self) {
        if let Some(ref left) = self.left {
            left.visit();
        }
        println!("'{}'", self.payload);
        if let Some(ref right) = self.right {
            right.visit();
        }
    }
    ...
    ...
    root.visit();
    // 'four'
    // 'one'
    // 'root'
    // 'two'
```

So we're visiting the strings in order! Please note the reappearance of `ref` - `if let` uses exactly the same rules as `match`.

Generic Structs

Consider the previous example of a binary tree. It would be *seriously irritating* to have to rewrite it for all possible kinds of payload. So here's our generic `Node` with its type parameter `T`.

```
type NodeBox<T> = Option<Box<Node<T>>>;

#[derive(Debug)]
struct Node<T> {
    payload: T,
    left: NodeBox<T>,
    right: NodeBox<T>
}
```

The implementation shows the difference between the languages. The fundamental operation on the payload is comparison, so T must be comparable with `<`, i.e. implements `PartialOrd`. The type parameter must be declared in the `impl` block with its constraints:

```
impl <T: PartialOrd> Node<T> {
    fn new(s: T) -> Node<T> {
        Node{payload: s, left: None, right: None}
    }

    fn boxer(node: Node<T>) -> NodeBox<T> {
        Some(Box::new(node))
    }

    fn set_left(&mut self, node: Node<T>) {
        self.left = Self::boxer(node);
    }

    fn set_right(&mut self, node: Node<T>) {
        self.right = Self::boxer(node);
    }

    fn insert(&mut self, data: T) {
        if data < self.payload {
            match self.left {
                Some(ref mut n) => n.insert(data),
                None => self.set_left(Self::new(data)),
            }
        } else {
            match self.right {
                Some(ref mut n) => n.insert(data),
                None => self.set_right(Self::new(data)),
            }
        }
    }
}


fn main() {
    let mut root = Node::new("root".to_string());
    root.insert("one".to_string());
    root.insert("two".to_string());
    root.insert("four".to_string());

    println!("root {:#?}", root);
}
```

So generic structs need their type parameter(s) specified in angle brackets, like C++. Rust is usually smart enough to work out that type parameter from context - it knows it has a `Node<T>`, and knows that its `insert` method is passed `T`. The first call of `insert` nails down `T` to be `String`. If any further calls are inconsistent it will complain.

But you do need to constrain that type appropriately!

Filesystem and Processes

Another look at Reading Files

At the end of Part 1, I showed how to read a whole file into a string. Naturally this isn't always such a good idea, so here is how to read a file line-by-line.

`fs::File` implements `io::Read`, which is the trait for anything readable. This trait defines a `read` method which will fill a slice of `u8` with bytes - this is the only *required* method of the trait, and you get some *provided* methods for free, much like with `Iterator`. You can use `read_to_end` to fill a vector of bytes with contents from the readable, and `read_to_string` to fill a string - which must be UTF-8 encoded.

This is a 'raw' read, with no buffering. For buffered reading there is the `io::BufRead` trait which gives us `read_line` and a `lines` iterator. `io::BufReader` will provide an implementation of `io::BufRead` for *any* readable.

`fs::File` *also* implements `io::Write`.

The easiest way to make sure all these traits are visible is `use std::io::prelude::*`.

```rust
use std::fs::File;
use std::io;
use std::io::prelude::*;

fn read_all_lines(filename: &str) -> io::Result<()> {
    let file = File::open(&filename)?;

    let reader = io::BufReader::new(file);

    for line in reader.lines() {
        let line = line?;
        println!("{}", line);
    }
    Ok(())
}
```

The `let line = line?` may look a bit strange. The `line` returned by the iterator is actually an `io::Result<String>` which we unwrap with `?`. Because things *can* go wrong during this iteration - I/O errors, swallowing a chunk of bytes that aren't UTF-8, and so forth.

`lines` being an iterator, it is straightforward to read a file into a vector of strings using `collect`, or print out the line with line numbers using the `enumerate` iterator.

It isn't the most efficient way to read all the lines, however, because a new string is allocated for each line. It is more efficient to use `read_line`, although more awkward. Note that the returned line includes the linefeed, which can be removed using `trim_right`.

```
    let mut reader = io::BufReader::new(file);
    let mut buf = String::new();
    while reader.read_line(&mut buf)? > 0 {
        {
            let line = buf.trim_right();
            println!("{}", line);
        }
        buf.clear();
    }
```

This results in far less allocations, because *clearing* that string does not free its allocated memory; once the string has enough capacity, no more allocations will take place.

This is one of those cases where we use a block to control a borrow. `line` is borrowed from `buf`, and this borrow must finish before we modify `buf`. Again, Rust is trying to stop us doing something stupid, which is to access `line` *after* we've cleared the buffer. (The borrow checker can be restrictive sometimes. Rust is due to get 'non-lexical lifetimes', where it will analyze the code and see that `line` isn't used after `buf.clear()`.)

This isn't very pretty. I cannot give you a proper iterator that returns references to a buffer, but I can give you something that *looks* like an iterator.

First define a generic struct; the type parameter `R` is 'any type that implements Read'. It contains the reader and the buffer which we are going to borrow from.

```
// file5.rs
use std::fs::File;
use std::io;
use std::io::prelude::*;

struct Lines<R> {
    reader: io::BufReader<R>,
    buf: String
}

impl <R: Read> Lines<R> {
    fn new(r: R) -> Lines<R> {
        Lines{reader: io::BufReader::new(r), buf: String::new()}
    }
    ...
}
```

Then the `next` method. It returns an `Option` - just like an iterator, when it returns `None` the iterator finishes. The returned type is a `Result` because `read_line` might fail, and we *never throw errors away*. So if fails, we wrap up its error in a `Some<Result>`. Otherwise, it may have read zero bytes, which is the natural end of the file - not an error, just a `None`.

At this point, the buffer contains the line with a linefeed (`\n') appended. Trim this away, and package up the string slice.

```
    fn next<'a>(&'a mut self) -> Option<io::Result<&'a str>>{
        self.buf.clear();
        match self.reader.read_line(&mut self.buf) {
            Ok(nbytes) => if nbytes == 0 {
                None // no more lines!
            } else {
                let line = self.buf.trim_right();
                Some(Ok(line))
            },
            Err(e) => Some(Err(e))
        }
    }
```

Now, note how the lifetimes work. We need an explicit lifetime because Rust will never allow us to hand out borrowed string slices without knowing their lifetime. And here we say that the lifetime of this borrowed string is within the lifetime of `self`.

And this signature, with the lifetime, is incompatible with the interface of `Iterator`. But it's easy to see problems if it were compatible; consider `collect` trying to make a vector of these string slices. There's no way this could work, since they're all borrowed from the same mutable string! (If you had read *all* the file into a string, then the string's `lines` iterator can return string slices because they are all borrowed from *distinct* parts of the original string.)

The resulting loop is much cleaner, and the file buffering is invisible to the user.

```
fn read_all_lines(filename: &str) -> io::Result<()> {
    let file = File::open(&filename)?;

    let mut lines = Lines::new(file);
    while let Some(line) = lines.next() {
        let line = line?;
        println!("{}", line);
    }

    Ok(())
}
```

You can even write the loop like this, since the explicit match can pull out the string slice:

```
    while let Some(Ok(line)) = lines.next() {
        println!("{}", line)?;
    }
```

It's tempting, but you are throwing away a possible error here; this loop will silently stop whenever an error occurs. In particular, it will stop at the first place where Rust can't convert a line to UTF-8. Fine for casual code, bad for production code!

Writing To Files

We met the `write!` macro when implementing `Debug` - it also works with anything that implements `Write`. So here's a another way of saying `print!`:

```
    let mut stdout = io::stdout();
    ...
    write!(stdout,"answer is {}\n", 42).expect("write failed");
```

If an error is *possible*, you must handle it. It may not be very *likely* but it can happen. It's usually fine, because if you are doing file i/o you should be in a context where `?` works.

But there is a difference: `print!` locks stdout for each write. This is usually what you want for output, because without that locking multithreaded programs can mix up that output in interesting ways. But if you are pumping out a lot of text, then `write!` is going to be faster.

For arbitrary files we need `write!`. The file is closed when `out` is dropped at the end of `write_out`, which is both convenient and important.

```
// file6.rs
use std::fs::File;
use std::io;
use std::io::prelude::*;

fn write_out(f: &str) -> io::Result<()> {
    let mut out = File::create(f)?;
    write!(out,"answer is {}\n", 42)?;
    Ok(())
}

fn main() {
  write_out("test.txt").expect("write failed");
}
```

If you care about performance, you need to know that Rust files are unbuffered by default. So each little write request goes straight to the OS, and this is going to be significantly slower. I mention this because this default is different from other programming languages, and could lead to the shocking discovery that Rust can be left in the dust by scripting languages! Just as with `Read` and `io::BufReader`, there is `io::BufWriter` for buffering any `Write`.

Files, Paths and Directories

Here is a little program for printing out the Cargo directory on a machine. The simplest case is that it's '~/.cargo'. This is a Unix shell expansion, so we use `env::home_dir` because it's cross-platform. (It might fail, but a computer without a home directory isn't going to be hosting Rust tools anyway.)

We then create a PathBuf and use its `push` method to build up the full file path from its *components*. (This is much easier than fooling around with '/','' or whatever, depending on the system.)

```rust
// file7.rs
use std::env;
use std::path::PathBuf;

fn main() {
    let home = env::home_dir().expect("no home!");
    let mut path = PathBuf::new();
    path.push(home);
    path.push(".cargo");

    if path.is_dir() {
        println!("{}", path.display());
    }
}
```

A `PathBuf` is like `String` - it owns a growable set of characters, but with methods specialized to building up paths. Most of its functionality however comes from the borrowed version `Path`, which is like `&str`. So, for instance, `is_dir` is a `Path` method.

This might sound suspiciously like a form of inheritance, but the magic Deref trait works differently. It works just like it does with `String/&str` - a reference to `PathBuf` can be *coerced* into a reference to `Path`. ('Coerce' is a strong word, but this really is one of the few places where Rust does conversions for you.)

```rust
fn foo(p: &Path) {...}
...
let path = PathBuf::from(home);
foo(&path);
```

`PathBuf` has an intimate relationship with `OsString`, which represents strings we get directly from the system. (There is a corresponding `OsString/&OsStr` relationship.)

Such strings are not *guaranteed* to be representable as UTF-8! Real life is a complicated matter, particularly see the answer to 'Why are they so hard?'. To summarize, first there are years of ASCII legacy coding, and multiple special encodings for other languages. Second, human languages are complicated. For instance 'noël' is *five* Unicode code points!

It's true that most of the time with modern operating systems file names will be Unicode (UTF-8 on the Unix side, UTF-16 for Windows), except when they're not. And Rust must handle that possibility rigorously. For instance, `Path` has a method `as_os_str` which returns a `&OsStr`, but the `to_str` method returns an `Option<&str>`. Not always possible!

People have trouble at this point because they have become too attached to 'string' and 'character' as the only necessary abstractions. As Einstein could have said, a programming language has to be as simple as possible, but no simpler. A systems language *needs* a `String/&str` distinction (owned versus borrowed: this is also very convenient) and if it wishes to standardize on Unicode strings then it needs another type to handle text which isn't valid Unicode - hence `OsString/&OsStr`. Notice that there aren't any interesting string-like methods for these types, precisely because we don't know the encoding.

But, people are used to processing filenames as if they were strings, which is why Rust makes it easier to manipulate file paths using `PathBuf` methods.

You can `pop` to successively remove path components. Here we start with the current directory of the program:

```
// file8.rs
use std::env;

fn main() {
    let mut path = env::current_dir().expect("can't access current dir");
    loop {
        println!("{}", path.display());
        if ! path.pop() {
            break;
        }
    }
}
// /home/steve/rust/gentle-intro/code
// /home/steve/rust/gentle-intro
// /home/steve/rust
// /home/steve
// /home
// /
```

Here's a useful variation. I have a program which searches for a configuration file, and the rule is that it may appear in any subdirectory of the current directory. So I create `/home/steve/rust/config.txt` and start this program up in `/home/steve/rust/gentle-intro/code`:

```
// file9.rs
use std::env;

fn main() {
    let mut path = env::current_dir().expect("can't access current dir");
    loop {
        path.push("config.txt");
        if path.is_file() {
            println!("gotcha {}", path.display());
            break;
        } else {
            path.pop();
        }
        if ! path.pop() {
            break;
        }
    }
}
// gotcha /home/steve/rust/config.txt
```

This is pretty much how **git** works when it wants to know what the current repo is.

The details about a file (its size, type, etc) are called its *metadata*. As always, there may be an error - not just 'not found' but also if we don't have permission to read this file.

```
// file10.rs
use std::env;
use std::path::Path;

fn main() {
    let file = env::args().skip(1).next().unwrap_or("file10.rs".to_string());
    let path = Path::new(&file);
    match path.metadata() {
        Ok(data) => {
            println!("type {:?}", data.file_type());
            println!("len {}", data.len());
            println!("perm {:?}", data.permissions());
            println!("modified {:?}", data.modified());
        },
        Err(e) => println!("error {:?}", e)
    }
}
// type FileType(FileType { mode: 33204 })
// len 488
// perm Permissions(FilePermissions { mode: 436 })
// modified Ok(SystemTime { tv_sec: 1483866529, tv_nsec: 600495644 })
```

The length of the file (in bytes) and modified time are straightforward to interpret. (Note we may not be able to get this time!) The file type has methods `is_dir`, `is_file` and `is_symlink`.

`permissions` is an interesting one. Rust strives to be cross-platform, and so it's a case of the 'lowest common denominator'. In general, all you can query is whether the file is read-only - the 'permissions' concept is extended in Unix and encodes read/write/executable for user/group/others.

But, if you are not interested in Windows, then bringing in a platform-specific trait will give us at least the permission mode bits. (As usual, a trait only kicks in when it is visible.) Then, applying the program to its own executable gives:

```
use std::os::unix::fs::PermissionsExt;
...
println!("perm {:o}",data.permissions().mode());
// perm 755
```

(Note '{:o}' for printing out in *octal*)

(Whether a file is executable on Windows is determined by its extension. The executable extensions are found in the `PATHEXT` environment variable - '.exe','.bat' and so forth).

`std::fs` contains a number of useful functions for working with files, such as copying or moving files, making symbolic links and creating directories.

To find the contents of a directory, `std::fs::read_dir` provides an iterator. Here are all files with extension '.rs' and size greater than 1024 bytes:

```
fn dump_dir(dir: &str) -> io::Result<()> {
    for entry in fs::read_dir(dir)? {
        let entry = entry?;
        let data = entry.metadata()?;
        let path = entry.path();
        if data.is_file() {
            if let Some(ex) = path.extension() {
                if ex == "rs" && data.len() > 1024 {
                    println!("{} length {}", path.display(),data.len());
                }
            }
        }
    }
    Ok(())
}
// ./enum4.rs length 2401
// ./struct7.rs length 1151
// ./sexpr.rs length 7483
// ./struct6.rs length 1359
// ./new-sexpr.rs length 7719
```

Obviously `read_dir` might fail (usually 'not found' or 'no permission'), but also getting each new entry might fail (it's like the `lines` iterator over a buffered reader's contents). Plus, we might not be able to get the metadata corresponding to the entry. A file might have no extension, so we have to check for that as well.

Why not just an iterator over paths? On Unix this is the way the `opendir` system call works, but on Windows you cannot iterate over a directory's contents without getting the metadata. So this is a reasonably elegant compromise that allows cross-platform code to be as efficient as possible.

You can be forgiven for feeling 'error fatigue' at this point. But please note that the *errors always existed* - it's not that Rust is inventing new ones. It's just trying hard to make it impossible for you to ignore them. Any operating system call may fail.

Languages like Java and Python throw exceptions; languages like Go and Lua return two values, where the first is the result and the second is the error: like Rust it is considered bad manners for library functions to raise errors. So there is a lot of error checking and early-returns from functions.

Rust uses `Result` because it's either-or: you cannot get both a result and an error. And the question-mark operator makes handling errors much cleaner.

Processes

A fundamental need is for programs to run programs, or to *launch processes*. Your program can *spawn* as many child processes it likes, and as the name suggests they have a special relationship with their parent.

To run a program is straightforward using the `Command` struct, which builds up arguments to pass to the program:

```
use std::process::Command;

fn main() {
    let status = Command::new("rustc")
        .arg("-V")
        .status()
        .expect("no rustc?");

    println!("cool {} code {}", status.success(), status.code().unwrap());
}
// rustc 1.15.0-nightly (8f02c429a 2016-12-15)
// cool true code 0
```

So `new` receives the name of the program (it will be looked up on `PATH` if not an absolute filename), `arg` adds a new argument, and `status` causes it to be run. This returns a `Result`, which is `Ok` if the program actually run, containing an `ExitStatus`. In this case, the program succeeded, and returned an exit code 0. (The `unwrap` is because we can't always get the code if the program was killed by a signal).

If we change the `-V` to `-v` (an easy mistake) then `rustc` fails:

```
error: no input filename given

cool false code 101
```

So there are three possibilities:

- program didn't exist, was bad, or we were not allowed to run it
- program ran, but was not successful - non-zero exit code
- program ran, with zero exit code. Success!

By default, the program's standard output and standard error streams go to the terminal.

Often we are very interested in capturing that output, so there's the `output` method.

```
// process2.rs
use std::process::Command;

fn main() {
    let output = Command::new("rustc")
        .arg("-V")
        .output()
        .expect("no rustc?");

    if output.status.success() {
        println!("ok!");
    }
    println!("len stdout {} stderr {}", output.stdout.len(), output.stderr.len());
}
// ok!
// len stdout 44 stderr 0
```

As with `status` our program blocks until the child process is finished, and we get back three things - the status (as before), the contents of stdout and the contents of stderr.

The captured output is simply `Vec<u8>` - just bytes. Recall we have no guarantee that data we receive from the operating system is a properly encoded UTF-8 string. In fact, we have no guarantee that it *even* is a string - programs may return arbitrary binary data.

If we are pretty sure the output is UTF-8, then `String::from_utf8` will convert those vectors or bytes - it returns a `Result` because this conversion may not succeed. A more sloppy function is `String::from_utf8_lossy` which will make a good attempt at conversion and insert the invalid Unicode mark � where it failed.

Here is a useful function which runs a program using the shell. This uses the usual shell mechanism for joining stderr to stdout. The name of the shell is different on Windows, but otherwise things work as expected.

```
fn shell(cmd: &str) -> (String,bool) {
    let cmd = format!("{} 2>&1",cmd);
    let shell = if cfg!(windows) {"cmd.exe"} else {"/bin/sh"};
    let flag = if cfg!(windows) {"/c"} else {"-c"};
    let output = Command::new(shell)
        .arg(flag)
        .arg(&cmd)
        .output()
        .expect("no shell?");
    (
        String::from_utf8_lossy(&output.stdout).trim_right().to_string(),
        output.status.success()
    )
}


fn shell_success(cmd: &str) -> Option<String> {
    let (output,success) = shell(cmd);
    if success {Some(output)} else {None}
}
```

I'm trimming any whitespace from the right so that if you said `shell("which rustc")` you will get the path without any extra linefeed.

You can control the execution of a program launched by `Process` by specifying the directory it will run in using the `current_dir` method and the environment variables it sees using `env`.

Up to now, our program simply waits for the child process to finish. If you use the `spawn` method then we return immediately, and must explicitly wait for it to finish - or go off and do something else in the meantime! This example also shows how to suppress both standard out and standard error:

```
// process5.rs
use std::process::{Command,Stdio};

fn main() {
    let mut child = Command::new("rustc")
        .stdout(Stdio::null())
        .stderr(Stdio::null())
        .spawn()
        .expect("no rustc?");

    let res = child.wait();
    println!("res {:?}", res);
}
```

By default, the child 'inherits' the standard input and output of the parent. In this case, we redirect the child's output handles into 'nowhere'. It's equivalent to saying `> /dev/null 2> /dev/null` in the Unix shell.

Now, it's possible to do these things using the shell (`sh` or `cmd`) in Rust. But this way you get full programmatic control of process creation.

For example, if we just had `.stdout(Stdio::piped())` then the child's standard output is redirected to a pipe. Then `child.stdout` is something you can use to directly read the output (i.e. implements `Read`). Likewise, you can use the `.stdout(Stdio::piped())` method so you can write to `child.stdin` .

But if we used `wait_with_output` instead of `wait` then it returns a `Result<Output>` and the child's output is captured into the `stdout` field of that `Output` as a `Vec<u8>` just as before.

The `Child` struct also gives you an explicit `kill` method.

Modules and Cargo

Modules

As programs get larger, it's necessary to spread them over more than one file and put functions and types in different *namespaces*. The Rust solution for both of these is *modules*.

C does the first, and not the second, so you end up with awful names like `primitive_display_set_width` and so forth. The actual filenames can be named arbitrarily.

In Rust the full name would look like `primitive::display::set_width` , and after saying `use primitive::display` you can then refer to it as `display::set_width` . You can even say `use primitive::display::set_width` and then just say `set_width` , but it's not a good idea to get carried away with this. `rustc` will not be confused, but *you* may get confused later. But for this to work, filenames must follow some simple rules.

A new keyword `mod` is used to define a module as a block where Rust types or functions can be written:

```rust
mod foo {
    #[derive(Debug)]
    struct Foo {
        s: &'static str
    }
}

fn main() {
    let f = foo::Foo{s: "hello"};
    println!("{:?}", f);
}
```

But it's still not quite right - we get 'struct Foo is private'. To solve this, we need the `pub` keyword to export `Foo`. The error then changes to 'field s of struct foo::Foo is private', so put `pub` before the field `s` to export `Foo::s`. Then things will work.

```rust
    pub struct Foo {
        pub s: &'static str
    }
```

Needing an explicit `pub` means that you must *choose* what items to make public from a module. The set of functions and types exported from a module is called its *interface*.

It is usually better to hide the insides of a struct, and only allow access through methods:

```rust
mod foo {
    #[derive(Debug)]
    pub struct Foo {
        s: &'static str
    }

    impl Foo {
        pub fn new(s: &'static str) -> Foo {
            Foo{s: s}
        }
    }
}

fn main() {
    let f = foo::Foo::new("hello");
    println!("{:?}", f);
}
```

Why is hiding the implementation a good thing? Because it means you may change it later without breaking the interface, without consumers of a module getting too dependent on its details. The great enemy of large-scale programing is a tendency for code to get entangled, so that understanding a piece of code is impossible in isolation.

In a perfect world a module does one thing, does it well, and keeps its own secrets.

When not to hide? As Stroustrup says, when the interface *is* the implementation, like `struct Point{x: f32, y: f32}`.

Within a module, all items are visible to all other items. It's a cozy place where everyone can be friends and know intimate details about each other.

Everyone gets to a point where they want to break a program up into separate files, depending on taste. I start getting uncomfortable around 500 lines, but we all agree that more than 2000 lines is pushing it.

So how to break this program into separate files?

We put the `foo` code into `foo.rs`:

```rust
// foo.rs
#[derive(Debug)]
pub struct Foo {
    s: &'static str
}

impl Foo {
    pub fn new(s: &'static str) -> Foo {
        Foo{s: s}
    }
}
```

And use a `mod foo` statement *without* a block in the main program:

```rust
// mod3.rs
mod foo;

fn main() {
    let f = foo::Foo::new("hello");
    println!("{:?}", f);
}
```

Now `rustc mod3.rs` will cause `foo.rs` to be compiled as well. There is no need to fool around with makefiles!

The compiler will also look at `MODNAME/mod.rs`, so this will work if I create a directory `boo` containing a file `mod.rs`:

```rust
// boo/mod.rs
pub fn answer()->u32 {
    42
}
```

And now the main program can use both modules as separate files:

```
// mod3.rs
mod foo;
mod boo;

fn main() {
    let f = foo::Foo::new("hello");
    let res = boo::answer();
    println!("{:?} {}", f,res);
}
```

So far, there's `mod3.rs`, containing `main`, a module `foo.rs` and a directory `boo` containing `mod.rs`. The usual convention is that the file containing `main` is just called `main.rs`.

Why two ways to do the same thing? Because `boo/mod.rs` can refer to other modules defined in `boo`, Update `boo/mod.rs` and add a new module - note that this is explicitly exported. (Without the `pub`, `bar` can only be seen inside the `boo` module.)

```
// boo/mod.rs
pub fn answer()->u32 {
    42
}

pub mod bar {
    pub fn question() -> &'static str {
        "the meaning of everything"
    }
}
```

and then we have the question corresponding to the answer (the `bar` module is inside `boo`):

```
let q = boo::bar::question();
```

That module block can be pulled out as `boo/bar.rs`:

```
// boo/bar.rs
pub fn question() -> &'static str {
    "the meaning of everything"
}
```

And `boo/mod.rs` becomes:

```
// boo/mod.rs
pub fn answer()->u32 {
    42
}

pub mod bar;
```

In summary, modules are about organization and visibility, and this may or may not involve separate files.

Please note that `use` has nothing to do with importing, and simply specifies visibility of module names. For example:

```
{
    use boo::bar;
    let q = bar::question();
    ...
}
{
    use boo::bar::question();
    let q = question();
    ...
}
```

An important point to note is there is no *separate compilation* here. The main program and its module files will be recompiled each time. Larger programs will take a fair amount of time to build, although `rustc` is getting better at incremental compilation.

Crates

The 'compilation unit' for Rust is the *crate*, which is either an executable or a library.

To separately compile the files from the last section, first build `foo.rs` as a Rust *static library* crate:

```
src$ rustc foo.rs --crate-type=lib
src$ ls -l libfoo.rlib
-rw-rw-r-- 1 steve steve 7888 Jan  5 13:35 libfoo.rlib
```

We can now *link* this into our main program:

```
src$ rustc mod4.rs --extern foo=libfoo.rlib
```

But the main program must now look like this, where the `extern` name is the same as the one used when linking. There is an implicit top-level module `foo` associated with the library crate:

```
// mod4.rs
extern crate foo;

fn main() {
    let f = foo::Foo::new("hello");
    println!("{:?}", f);
}
```

Before people start chanting 'Cargo! Cargo!' let me justify this lower-level look at building Rust. I'm a great believer in 'Know Thy Toolchain', and this will reduce the amount of new magic you need to learn when we look at managing projects with Cargo. Modules are basic language features and can be used outside Cargo projects.

It's time to understand why Rust binaries are so large:

```
src$ ls -lh mod4
-rwxrwxr-x 1 steve steve 3,4M Jan  5 13:39 mod4
```

That's rather fat! There is a *lot* of debug information in that executable. This is not a Bad Thing, if you want to use a debugger and actually want meaningful backtraces when your program panics.

So let's strip that debug information and see:

```
src$ strip mod4
src$ ls -lh mod4
-rwxrwxr-x 1 steve steve 300K Jan  5 13:49 mod4
```

Still feels a little large for something so simple, but this program links *statically* against the Rust standard library. This is a Good Thing, since you can hand this executable to anyone with the right operating system - they will not need a 'Rust runtime'. (And `rustup` will even let you cross-compile for other operating systems and platforms as well.)

We can link dynamically against the Rust runtime and get truly tiny exes:

```
src$ rustc -C prefer-dynamic mod4.rs --extern foo=libfoo.rlib
src$ ls -lh mod4
-rwxrwxr-x 1 steve steve 14K Jan  5 13:53 mod4
src$ ldd mod4
    linux-vdso.so.1 =>  (0x00007fffa8746000)
    libstd-b4054fae3db32020.so => not found
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3cd47aa000)
    /lib64/ld-linux-x86-64.so.2 (0x00007f3cd4d72000)
```

That 'not found' is because `rustup` doesn't install the dynamic libraries globally. We can hack our way to happiness, at least on Unix (yes, I know the best solution is a symlink.)

```
src$ export LD_LIBRARY_PATH=~/.rustup/toolchains/nightly-x86_64-unknown-linux-gnu/lib
src$ ./mod4
Foo { s: "hello" }
```

Rust does not have a *philosophical* problem with dynamic linking, in the same way as Go does. It's just that when there's a stable release every 6 weeks it becomes inconvenient to have to recompile everything. If you have a stable version that Works For You, then cool. As stable versions of Rust get increasingly delivered by the OS package manager, dynamic linking will become more popular.

Cargo

The Rust standard library is not very large, compared to Java or Python; although much more fully featured than C or C++, which lean heavily on operating system provided libraries.

But it is straightforward to access community-provided libraries in crates.io using **Cargo**. Cargo will look up the correct version and download the source for you, and ensures that any other needed crates are downloaded as well.

Let's create a simple program which needs to read JSON. This data format is very widely used, but is too specialized for inclusion in the standard library. So we initialize a Cargo project, using '--bin' because the default is to create a library project.

```
test$ cargo init --bin test-json
     Created binary (application) project
test$ cd test-json
test$ cat Cargo.toml
[package]
name = "test-json"
version = "0.1.0"
authors = ["Your Name <you@example.org>"]

[dependencies]
```

To make the project depend on the JSON crate, edit the 'Cargo.toml' file so:

```
[dependencies]
json="0.11.4"
```

Then do a first build with Cargo:

```
test-json$ cargo build
    Updating registry `https://github.com/rust-lang/crates.io-index`
 Downloading json v0.11.4
   Compiling json v0.11.4
   Compiling test-json v0.1.0 (file:///home/steve/c/rust/test/test-json)
    Finished debug [unoptimized + debuginfo] target(s) in 1.75 secs
```

The main file of this project has already been created - it's 'main.rs' in the 'src' directory. It starts out just as a 'hello world' app, so let's edit it to be a proper test program.

Note the very convenient 'raw' string literal - otherwise we would need to escape those double quotes and end up with ugliness:

```
// test-json/src/main.rs
extern crate json;

fn main() {
    let doc = json::parse(r#"
    {
        "code": 200,
        "success": true,
        "payload": {
            "features": [
                "awesome",
                "easyAPI",
                "lowLearningCurve"
            ]
        }
    }
    "#).expect("parse failed");

    println!("debug {:?}", doc);
    println!("display {}", doc);
}
```

You can now build and run this project - only `main.rs` has changed.

```
test-json$ cargo run
   Compiling test-json v0.1.0 (file:///home/steve/c/rust/test/test-json)
    Finished debug [unoptimized + debuginfo] target(s) in 0.21 secs
     Running `target/debug/test-json`
debug Object(Object { store: [("code", Number(Number { category: 1, exponent: 0, mantissa: 200
}),
 0, 1), ("success", Boolean(true), 0, 2), ("payload", Object(Object { store: [("features",
 Array([Short("awesome"), Short("easyAPI"), Short("lowLearningCurve")]), 0, 0)] }), 0, 0)] })
display {"code":200,"success":true,"payload":{"features":
["awesome","easyAPI","lowLearningCurve"]}}
```

The debug output shows some internal details of the JSON document, but a plain '{}', using the `Display` trait, regenerates JSON from the parsed document.

Let's explore the JSON API. It would not be useful if we could not extract values. The `as_TYPE` methods return `Option<TYPE>` since we cannot be sure that the field exists or is of the correct type. (see the docs for JsonValue)

```
    let code = doc["code"].as_u32().unwrap_or(0);
    let success = doc["success"].as_bool().unwrap_or(false);

    assert_eq!(code, 200);
    assert_eq!(success, true);

    let features = &doc["payload"]["features"];
    for v in features.members() {
        println!("{}", v.as_str().unwrap()); // MIGHT explode
    }
    // awesome
    // easyAPI
    // lowLearningCurve
```

`features` here is a reference to `JsonValue` - it has to be a reference because otherwise we would be trying to move a *value* out of the JSON document. Here we know it's an array, so `members()` will return a non-empty iterator over `&JsonValue`.

What if the 'payload' object didn't have a 'features' key? Then `features` would be set to `Null`. There will be no explosion. This convenience expresses the free-form, anything-goes nature of JSON very well. It is up to you to examine the structure of any document you receive and create your own errors if the structure does not match.

You can modify these structures. If we had `let mut doc` then this would do what you expect:

```
let features = &mut doc["payload"]["features"];
features.push("cargo!").expect("couldn't push");
```

The `push` will fail if `features` wasn't an array, hence it returns `Result<()>`.

Here's a truly beautiful use of macros to generate *JSON literals*:

```
let data = object!{
    "name"    => "John Doe",
    "age"     => 30,
    "numbers" => array![10,53,553]
};
assert_eq!(
    data.dump(),
    r#"{"name":"John Doe","age":30,"numbers":[10,53,553]}"#
);
```

For this to work, you need to explicitly import macros from the JSON crate thus:

```
#[macro_use]
extern crate json;
```

There is a downside to using this crate, because of the mismatch between the amorphous, dynamically-typed nature of JSON and the structured, static nature of Rust. (The readme explicitly speaks of 'friction') So if you *did* want to map JSON to Rust data structures, you would end up doing a lot of checking, because you can not assume that the received structure matches your structs! For that, a better solution is serde_json where you *serialize* Rust data structures into JSON and *deserialize* JSON into Rust.

For this, create a another Cargo binary project with `cargo new --bin test-serde-json`, go into the `test-serde-json` directory and edit `Cargo.toml`. Edit it like so:

```
[dependencies]
serde="0.9"
serde_derive="0.9"
serde_json="0.9"
```

And edit `src/main.rs` to be this:

```
#[macro_use]
extern crate serde_derive;
extern crate serde_json;

#[derive(Serialize, Deserialize, Debug)]
struct Person {
    name: String,
    age: u8,
    address: Address,
    phones: Vec<String>,
}

#[derive(Serialize, Deserialize, Debug)]
struct Address {
    street: String,
    city: String,
}

fn main() {
    let data = r#" {
     "name": "John Doe", "age": 43,
     "address": {"street": "main", "city":"Downtown"},
     "phones":["27726550023"]
    } "#;
    let p: Person = serde_json::from_str(data).expect("deserialize error");
    println!("Please call {} at the number {}", p.name, p.phones[0]);

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

You have seen the `derive` attribute before, but the `serde_derive` crate defines *custom derives* for the special `Serialize` and `Deserialize` traits. And the result shows the generated Rust struct:

```
Please call John Doe at the number 27726550023
Person {
    name: "John Doe",
    age: 43,
    address: Address {
        street: "main",
        city: "Downtown"
    },
    phones: [
        "27726550023"
    ]
}
```

Now, if you did this using the `json` crate, you would need a few hundred lines of custom conversion code, mostly error handling. Tedious, easy to mess up, and not where you want to put effort anyway.

This is clearly the best solution if you are processing well-structured JSON from outside sources (it's possible to remap field names if needed) and provides a robust way for Rust programs to share data with other programs over the network (since everything understands JSON these days.) The cool thing about `serde` (for SERialization DEserialization) is that other file formats are also supported, such as `toml`, which is the popular configuration-friendly format used in Cargo. So your program can read .toml files into structs, and write those structs out as .json.

Serialization is an important technique and similar solutions exist for Java and Go - but with a big difference. In those languages the structure of the data is found at *run-time* using *reflection*, but in this case the serialization code is generated at *compile-time* - altogether more efficient!

Cargo is considered to be one of the great strengths of the Rust ecosystem, because it does a lot of work for us. Otherwise we would have had to download these libraries from Github, build as static library crates, and link them against the program. It's painful to do this for C++ projects, and would be nearly as painful for Rust projects if Cargo did not exist. C++ is somewhat unique in its painfullness here, so we should compare this with other languages' package managers. npm (for JavaScript) and pip (for Python) manage dependencies and downloads for you, but the distribution story is harder, since the user of your program needs NodeJS or Python installed. But Rust programs are statically linked against their dependencies, so again they can be handed out to your buddies without external dependencies.

More Gems

When processing anything except simple text, regular expressions make your life much easier. These are commonly available for most languages and I'll here assume a basic familiarity with regex notation. To use the regex crate, put 'regex = "0.2.1"' after "[dependencies]" in your Cargo.toml.

We'll use 'raw strings' again so that the backslashes don't have to be escaped. In English, this regular expression is "match exactly two digits, the character ':', and then any number of digits. Capture both sets of digits":

```
extern crate regex;
use regex::Regex;

let re = Regex::new(r"(\d{2}):(\d+)").unwrap();
println!("{:?}", re.captures("  10:230"));
println!("{:?}", re.captures("[22:2]"));
println!("{:?}", re.captures("10:x23"));
// Some(Captures({0: Some("10:230"), 1: Some("10"), 2: Some("230")}))
// Some(Captures({0: Some("22:2"), 1: Some("22"), 2: Some("2")}))
// None
```

The successful output actually has three *captures* - the whole match, and the two sets of digits. These regular expressions are not *anchored* by default, so **regex** will hunt for the first occurring match, skipping anything that doesn't match. (If you left out the '()' it would just give us the whole match.)

It's possible to *name* those captures, and spread the regular expression over several lines, even including comments! Compiling the regex might fail (the first *expect*) or the match might fail (the second *expect*). Here we can use the result as an associative array and look up captures by name.

```
let re = Regex::new(r"(?x)
(?P<year>\d{4})  # the year
-
(?P<month>\d{2}) # the month
-
(?P<day>\d{2})   # the day
").expect("bad regex");
let caps = re.captures("2010-03-14").expect("match failed");

assert_eq!("2010", &caps["year"]);
assert_eq!("03", &caps["month"]);
assert_eq!("14", &caps["day"]);
```

Regular expressions can break up strings that match a pattern, but won't check whether they make sense. That is, you can specify and match the *syntax* of ISO-style dates, but *semantically* they may be nonsense, like "2014-24-52".

For this, you need dedicated date-time processing, which is provided by chrono. You need to decide on a time zone when doing dates:

```
extern crate chrono;
use chrono::*;

fn main() {
    let date = Local.ymd(2010,3,14);
    println!("date was {}", date);
}
// date was 2010-03-14+02:00
```

However, this isn't recommended because feeding it bad dates will cause a panic! (try a bogus date to see this.) The method you need here is `ymd_opt` which returns `LocalResult<Date>`

```
    let date = Local.ymd_opt(2010,3,14);
    println!("date was {:?}", date);
    // date was Single(2010-03-14+02:00)

    let date = Local.ymd_opt(2014,24,52);
    println!("date was {:?}", date);
    // date was None
```

You can also directly parse date-times, either in standard UTC format or using custom formats. These self-same formats allow you to print out dates in exactly the format you want.

I specifically highlighted these two useful crates because they would be part of the standard library in most other languages. And, in fact, the embryonic form of these crates was once part of the Rust stdlib, but were cut loose. This was a deliberate decision: the Rust team takes stdlib stability very seriously so features only arrive in stable once they have gone through incubation in unstable nightly versions, and only then beta and stable. For libraries that need experimentation and refinement, it's much better that they remain independent and get tracked with Cargo. For all practical purposes, these two crates *are* standard - they are not going away and may be folded back into the stdlib at some point.

Standard Library Containers

Reading the Documentation

In this section I'll briefly introduce some common parts of the Rust standard library. The documentation is excellent but a little context and a few examples is always useful.

Initially, reading the Rust documentation can be challenging, so I'll go through `Vec` as an example. A useful tip is to tick the '[-]' box to collapse the docs. (If you download the standard library source using `rustup component add rust-src` a '[src]' link will appear next to this.) This gives you a bird's eye view of all the available methods.

The first point to notice is that *not all possible methods* are defined on `Vec` itself. They are (mostly) mutable methods that change the vector, e.g. `push`. Some methods are only implemented for vectors where the type matches some constraint. For example, you can only call `dedup` (remove duplicates) if the type is indeed something that can be compared for equality. There are multiple `impl` blocks that define `Vec` for different type constraints.

Then there's the very special relationship between `Vec<T>` and `&[T]`. Any method that works on slices will also directly work on vectors, without explicitly having to use the `as_slice` method. This relationship is expressed by `Deref<Target=[T]>`. This also kicks in when you pass a vector by reference to something that expects a slice - this is one of the few places where a conversion between types happens automatically. So slice methods like `first`, which maybe-returns a reference to the first element, or `last`, work for vectors as well. Many of the methods are similar to the corresponding string methods, so there's `split_at` for getting a pair of slices split at an index, `starts_with` to check whether a vector starts with sequence of values, and `contains` to check whether a vector contains a particular value.

There's no `search` method for finding the index of a particular value, but here's a rule of thumb: if you can't find a method on the container, look for a method on the iterator:

```rust
let v = vec![10,20,30,40,50];
assert_eq!(v.iter().position(|&i| i == 30).unwrap(), 2);
```

(The `&` is because this is an iterator over *references* - alternatively you could say `*i == 30` for the comparison.)

Likewise, there's no `map` method on vectors, because `iter().map(...).collect()` will do the job just as well. Rust does not like to allocate unnecessarily - often you don't need the result of that `map` as an actual allocated vector.

So I'd suggest you become familiar with all the iterator methods, because they are crucial to writing good Rust code without having to write loops out all the time. As always, write little programs to explore iterator methods, rather than wrestling with them in the context of a more complicated program.

The `Vec<T>` and `&[T]` methods are followed by the common traits: vectors know how to do a debug display of themselves (but only if the elements implement `Debug`). Likewise, they are clonable if their elements are clonable. They implement `Drop`, which happens when vectors get to finally die; memory is released, and all the elements are dropped as well.

The `Extend` trait means values from iterators can be added to a vector without a loop:

```
v.extend([60,70,80].iter());
let mut strings = vec!["hello".to_string(), "dolly".to_string()];
strings.extend(["you","are","fine"].iter().map(|s| s.to_string()));
```

There's also `FromIterator`, which lets vectors be *constructed* from iterators. (The iterator `collect` method leans on this.)

Any container needs to be iterable as well. Recall that there are three kinds of iterators

```
for x in v {...} // returns T, consumes v
for x in &v {...} // returns &T
for x in &mut v {...} // returns &mut T
```

The `for` statement relies on the `IntoIterator` trait, and there's indeed three implementations.

Then there is indexing, controlled by `Index` (reading from a vector) and `IndexMut` (modifying a vector.) There are many possibilities, because there's slice indexing as well, like `v[0..2]`, returning these return slices, as well as plain `v[0]` which returns a reference to the first element.

There's a few implementations of the `From` trait. For instance `Vec::from("hello".to_string())` will give you a vector of the underlying bytes of the string `Vec<u8>`. Now, there's already a method `into_bytes` on `String`, so why the redundancy? It seems confusing to have multiple ways of doing the same thing. But it's needed because explicit traits make generic methods possible.

Sometimes limitations of the Rust type system make things clumsy. An example here is how `PartialEq` is *separately* defined for arrays up to size 32! (This will get better.) This allows the convenience of directly comparing vectors with arrays, but beware the size limit.

And there are Hidden Gems buried deep in the documentation. As Karol Kuczmarski says "Because let's be honest: no one scrolls that far". How does one handle errors in an iterator? Say you map over some operation that might fail and so returns `Result`, and then want to collect the results:

```
fn main() {
    let nums = ["5","52","65"];
    let iter = nums.iter().map(|s| s.parse::<i32>());
    let converted: Vec<_> = iter.collect();
    println!("{:?}",converted);
}
//[Ok(5), Ok(52), Ok(65)]
```

Fair enough, but now you have to unwrap these errors - carefully!. But Rust already knows how to do the Right Thing, if you ask for the vector to be *contained* in a `Result` - that is, either is a vector or an error:

```
    let converted: Result<Vec<_>,_> = iter.collect();
//Ok([5, 52, 65])
```

And if there was a bad conversion? Then you would just get `Err` with the first error encountered. It's a good example of how extremely flexible `collect` is. (The notation here can be intimidating - `Vec<_>` means "this is a vector, work out the actual type for me `and` Result<Vec<>,>` is furthermore asking Rust to work out the error type as well.)

So there's a *lot* of detail in the documentation. But it's certainly clearer than what the C++ docs say about `std::vector`

> The requirements that are imposed on the elements depend on the actual operations performed on the container. Generally, it is required that element type is a complete type and meets the requirements of Erasable, but many member functions impose stricter requirements.

With C++, you're on your own. The explicitness of Rust is daunting at first, but as you learn to read the constraints you will know exactly what any particular method of `Vec` requires.

I would suggest that you get the source using `rustup component add rust-src`, since the standard library source is very readable and the method implementations are usually less scary than the method declarations.

Maps

Maps (sometimes called *associative arrays* or *dicts*) let you look up values associated with a *key*. It's not really a fancy concept, and can be done with an array of tuples:

```
    let entries = [("one","eins"),("two","zwei"),("three","drei")];

    if let Some(val) = entries.iter().find(|t| t.0 == "two") {
        assert_eq!(val.1,"zwei");
    }
```

This is fine for small maps and just requires equality to be defined for the keys, but the search takes linear time - proportional to the size of the map.

A `HashMap` does much better when there are a *lot* of key/value pairs to be searched:

```
use std::collections::HashMap;

let mut map = HashMap::new();
map.insert("one","eins");
map.insert("two","zwei");
map.insert("three","drei");

assert_eq! (map.contains_key("two"), true);
assert_eq! (map.get("two"), Some(&"zwei"));
```

`&"zwei"`? This is because `get` returns a *reference* to the value, not the value itself. Here the value type is `&str`, so we get a `&&str`. In general it *has* to be a reference, because we can't just *move* a value out of its owning type.

`get_mut` is like `get` but returns a possible mutable reference. Here we have a map from strings to integers, and wish to update the value for the key 'two':

```
let mut map = HashMap::new();
map.insert("one",1);
map.insert("two",2);
map.insert("three",3);

println!("before {}", map.get("two").unwrap());

{
    let mut mref = map.get_mut("two").unwrap();
    *mref = 20;
}

println!("after {}", map.get("two").unwrap());
// before 2
// after 20
```

Note that getting that writable reference takes place in its own block - otherwise, we would have a mutable borrow lasting until the end, and then Rust won't allow you to borrow from `map` again with `map.get("two")`; it cannot allow any readable references while there's already a writable reference in scope. (If it did, it could not guarantee that those readable references would remain valid.) So the solution is to make sure that mutable borrow doesn't last very long.

It is not the most elegant API possible, but we can't throw away any possible errors. Python would bail out with an exception, and C++ would just create a default value. (This is convenient but sneaky; easy to forget that the price of `a_map["two"]` always returning an integer is that we can't tell the difference between zero and 'not found', *plus* an extra entry is created!)

And no-one just calls `unwrap`, except in examples. However, most Rust code you see consists of little standalone examples! Much more likely for a match to take place:

```
if let Some(v) = map.get("two") {
    let res = v + 1;
    assert_eq!(res, 3);
}
...
match map.get_mut("two") {
    Some(mref) => *mref = 20,
    None => panic!("_now_ we can panic!")
}
```

We can iterate over the key/value pairs, but not in any particular order.

```
for (k,v) in map.iter() {
    println!("key {} value {}", k,v);
}
// key one value eins
// key three value drei
// key two value zwei
```

There are also `keys` and `values` methods returning iterators over the keys and values respectively, which makes creating vectors of values easy.

Example: Counting Words

An entertaining thing to do with text is count word frequency. It is straightforward to break text into words with `split_whitespace`, but really we must respect punctuation. So the words should be defined as consisting only of alphabetic characters. And the words need to be compared as lower-case as well.

Doing a mutable lookup on a map is straightforward, but also handling the case where the lookup fails is a little awkward. Fortunately there's an elegant way to update the values of a map:

```
let mut map = HashMap::new();

for s in text.split(|c: char| ! c.is_alphabetic()) {
    let word = s.to_lowercase();
    let mut count = map.entry(word).or_insert(0);
    *count += 1;
}
```

If there's no existing count corresponding to a word, then let's create a new entry containing zero for that word and *insert* it into the map. Its exactly what a C++ map does, except it's done explicitly and not sneakily.

There is exactly one explicit type in this snippet, and that's the `char` needed because of a quirk of the string `Pattern` trait used by `split`. But we can deduce that the key type is `String` and the value type is `i32`.

Using The Adventures of Sherlock Holmes from Project Gutenberg, we can test this out more thoroughly. The total number of unique words (`map.len()`) is 8071.

How to find the twenty most common words? First, convert the map into a vector of (key,value) tuples. (This consumes the map, since we used `into_iter`.)

```
let mut entries: Vec<_> = map.into_iter().collect();
```

Next we can sort in descending order. `sort_by` expects the result of the `cmp` method that comes from the `Ord` trait, which is implemented by the integer value type:

```
    entries.sort_by(|a,b| b.1.cmp(&a.1));
```

And finally print out the first twenty entries:

```
    for e in entries.iter().take(20) {
        println!("{} {}", e.0, e.1);
    }
```

(Well, you *could* just loop over `0..20` and index the vector here - it isn't wrong, just a little un-idiomatic - and potentially more expensive for big iterations.)

```
 38765
the 5810
and 3088
i 3038
to 2823
of 2778
a 2701
in 1823
that 1767
it 1749
you 1572
he 1486
was 1411
his 1159
is 1150
my 1007
have 929
with 877
as 863
had 830
```

A little surprise - what's that empty word? It is because `split` works on single-character delimiters, so any punctuation or extra spaces causes a new split.

Sets

Sets are maps where you care only about the keys, not any associated values. So `insert` only takes one value, and you use `contains` for testing whether a value is in a set.

Like all containers, you can create a `HashSet` from an iterator. And this is exactly what `collect` does, once you have given it the necessary type hint.

```
// set1.rs
use std::collections::HashSet;

fn make_set(words: &str) -> HashSet<&str> {
    words.split_whitespace().collect()
}

fn main() {
    let fruit = make_set("apple orange pear orange");

    println!("{:?}", fruit);
}
// {"orange", "pear", "apple"}
```

Note (as expected) that repeated insertions of the same key have no effect, and the order of values in a set are not important.

They would not be sets without the usual operations:

```
let fruit = make_set("apple orange pear");
let colours = make_set("brown purple orange yellow");

for c in fruit.intersection(&colours) {
    println!("{:?}",c);
}
// "orange"
```

They all create iterators, and you can use `collect` to make these into sets.

Here's a shortcut, just as we defined for vectors:

```
use std::hash::Hash;

trait ToSet<T> {
    fn to_set(self) -> HashSet<T>;
}

impl <T,I> ToSet<T> for I
where T: Eq + Hash, I: Iterator<Item=T> {

    fn to_set(self) -> HashSet<T> {
       self.collect()
    }
}

...

let intersect = fruit.intersection(&colours).to_set();
```

As with all Rust generics, you do need to constrain types - this can only be implemented for types that understand equality (`Eq`) and for which a 'hash function' exists (`Hash`). Remember that there is no *type* called `Iterator` , so `I` represents any type that *implements* `Iterator` .

This technique for implementing our own methods on standard library types may appear to be a little too powerful, but again, there are Rules. We can only do this for our own traits. If both the struct and the trait came from the same crate (particularly, the stdlib) then such implemention would not be allowed. In this way, you are saved from creating confusion.

Before congratulating ourselves on such a clever, convenient shortcut, you should be aware of the consequences. If `make_set` was written so, so that these are sets of owned strings, then the actual type of `intersect` could come as a surprise:

```
fn make_set(words: &str) -> HashSet<String> {
    words.split_whitespace().map(|s| s.to_string()).collect()
}
...
// intersect is HashSet<&String>!
let intersect = fruit.intersection(&colours).to_set();
```

And it cannot be otherwise, since Rust will not suddenly start making copies of owned strings. `intersect` contains a single `&String` borrowed from `fruit` . I can promise you that this will cause you trouble later, when you start patching up lifetimes! A better solution is to use the iterator's `cloned` method to make owned string copies of the intersection.

```
// intersect is HashSet<String> - much better
let intersect = fruit.intersection(&colours).cloned().to_set();
```

A more robust definition of `to_set` might be `self.cloned().collect()` , which I invite you to try out.

Example: Interactive command processing

It's often useful to have an interactive session with a program. Each line is read in and split into words; the command is looked up on the first word, and the rest of the words are passed as an argument to that command.

A natural implementation is a map from command names to closures. But how do we store closures, given that they will all have different sizes? Boxing them will copy them onto the heap:

Here's a first try:

```
let mut v = Vec::new();
v.push(Box::new(|x| x * x));
v.push(Box::new(|x| x / 2.0));

for f in v.iter() {
    let res = f(1.0);
    println!("res {}", res);
}
```

We get a very definite error on the second push:

```
  = note: expected type `[closure@closure4.rs:4:21: 4:28]`
  = note:    found type `[closure@closure4.rs:5:21: 5:28]`
note: no two closures, even if identical, have the same type
```

`rustc` has deduced a type which is too specific, so it's necessary to force that vector to have the *boxed trait type* before things just work:

```
let mut v: Vec<Box<Fn(f64)->f64>> = Vec::new();
```

We can now use the same trick and keep these boxed closures in a `HashMap`. We still have to watch out for lifetimes, since closures can borrow from their environment.

It's tempting as first to make them `FnMut` - that is, they can modify any captured variables. But we will have more than one command, each with its own closure, and you cannot then mutably borrow the same variables.

So the closures are passed a *mutable reference* as an argument, plus a slice of string slices (`&[&str]`) representing the command arguments. They will return some `Result` - We'll use `String` errors at first.

`D` is the data type, which can be anything with a size.

```rust
type CliResult = Result<String,String>;

struct Cli<'a,D> {
    data: D,
    callbacks: HashMap<String, Box<Fn(&mut D,&[&str])->CliResult + 'a>>
}

impl <'a,D: Sized> Cli<'a,D> {
    fn new(data: D) -> Cli<'a,D> {
        Cli{data: data, callbacks: HashMap::new()}
    }

    fn cmd<F>(&mut self, name: &str, callback: F)
    where F: Fn(&mut D, &[&str])->CliResult + 'a {
        self.callbacks.insert(name.to_string(),Box::new(callback));
    }
```

`cmd` is passed a name and any closure that matches our signature, which is boxed and entered into the map. `Fn` means that our closures borrow their environment but can't modify it. It's one of those generic methods where the declaration is scarier than the actual implementation! Forgetting the explicit lifetime is a common error here - Rust won't let us forget that these closures have a lifetime limited to their environment!

Now for reading and running commands:

```rust
    fn process(&mut self,line: &str) -> CliResult {
        let parts: Vec<_> = line.split_whitespace().collect();
        if parts.len() == 0 {
            return Ok("".to_string());
        }
        match self.callbacks.get(parts[0]) {
            Some(callback) => callback(&mut self.data,&parts[1..]),
            None => Err("no such command".to_string())
        }
    }

    fn go(&mut self) {
        let mut buff = String::new();
        while io::stdin().read_line(&mut buff).expect("error") > 0 {
            {
                let line = buff.trim_left();
                let res = self.process(line);
                println!("{:?}", res);

            }
            buff.clear();
        }
    }
```

This is all reasonably straightforward - split the line into words as a vector, look up the first word in the map and call the closure with our stored mutable data and the rest of the words. An empty line is ignored and not considered an error.

Next, let's define some helper functions to make it easier for our closures to return correct and incorrect results. There's a *little* bit of cleverness going on; they are generic functions that work for any type that can be converted to a `String`.

```
fn ok<T: ToString>(s: T) -> CliResult {
    Ok(s.to_string())
}

fn err<T: ToString>(s: T) -> CliResult {
    Err(s.to_string())
}
```

So finally, the Main Program. Look at how `ok(answer)` works - because integers know how to convert themselves to strings!

```
use std::error::Error;

fn main() {
    println!("Welcome to the Interactive Prompt! ");

    struct Data {
        answer: i32
    }

    let mut cli = Cli::new(Data{answer: 42});

    cli.cmd("go",|data,args| {
        if args.len() == 0 { return err("need 1 argument"); }
        data.answer = match args[0].parse::<i32>() {
            Ok(n) => n,
            Err(e) => return err(e.description())
        };
        println!("got {:?}", args);
        ok(data.answer)
    });

    cli.cmd("show",|data,_| {
        ok(data.answer)
    });

    cli.go();
}
```

The error handling is a bit clunky here, and we'll later see how to use the question mark operator in cases like this. Basically, the particular error `std::num::ParseIntError` implements the trait `std::error::Error`, which we must bring into scope to use the `description` method - Rust doesn't let traits operate unless they're visible.

And in action:

```
Welcome to the Interactive Prompt!
go 32
got ["32"]
Ok("32")
show
Ok("32")
goop one two three
Err("no such command")
go 42 one two three
got ["42", "one", "two", "three"]
Ok("42")
go boo!
Err("invalid digit found in string")
```

Here are some obvious improvements for you to try. First, if we give `cmd` three arguments with the second being a help line, then we can store these help lines and automatically implement a 'help' command. Second, having some command editing and history is *very* convenient, so use the rustyline crate from Cargo.

Error Handling

Basic Error Handling

Error handling in Rust can be clumsy if you can't use the question-mark operator. To achieve happiness, we need to return a `Result` which can accept any error. All errors implement the trait `std::error::Error`, and so *any* error can convert into a `Box<Error>`.

Say we needed to handle *both* i/o errors and errors from converting strings into numbers:

```
// box-error.rs
use std::fs::File;
use std::io::prelude::*;
use std::error::Error;

fn run(file: &str) -> Result<i32,Box<Error>> {
    let mut file = File::open(file)?;
    let mut contents = String::new();
    file.read_to_string(&mut contents)?;
    Ok(contents.trim().parse()?)
}
```

So that's two question-marks for the i/o errors (can't open file, or can't read as string) and one question-mark for the conversion error. Finally, we wrap the result in `Ok`. Rust can work out from the return type that `parse` should convert to `i32`.

It's easy to create a shortcut for this `Result` type:

```
type BoxResult<T> = Result<T,Box<Error>>;
```

However, our programs will have application-specific error conditions, and so we need to create our own error type. The basic requirements are straightforward:

- May implement `Debug`
- Must implement `Display`
- Must implement `Error`

Otherwise, your error can do pretty much what it likes.

```rust
// error1.rs
use std::error::Error;
use std::fmt;

#[derive(Debug)]
struct MyError {
    details: String
}

impl MyError {
    fn new(msg: &str) -> MyError {
        MyError{details: msg.to_string()}
    }
}

impl fmt::Display for MyError {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f,"{}",self.details)
    }
}

impl Error for MyError {
    fn description(&self) -> &str {
        &self.details
    }
}

// a test function that returns our error result
fn raises_my_error(yes: bool) -> Result<(),MyError> {
    if yes {
        Err(MyError::new("borked"))
    } else {
        Ok(())
    }
}
```

Typing `Result<T,MyError>` gets tedious and many Rust modules define their own `Result` - e.g. `io::Result<T>` is short for `Result<T,io::Error>`.

In this next example we need to handle the specific error when a string can't be parsed as a floating-point number.

Now the way that `?` works is to look for a conversion from the error of the *expression* to the error that must be *returned*. And this conversion is expressed by the `From` trait. `Box<Error>` works as it does because it implements `From` for all types implementing `Error`.

At this point you can continue to use the convenient alias `BoxResult` and catch everything as before; there will be a conversion from our error into `Box<Error>`. This is a good option for smaller applications. But I want to show other errors can be explicitly made to cooperate with our error type.

`ParseFloatError` implements `Error` so `description()` is defined.

```
use std::num::ParseFloatError;

impl From<ParseFloatError> for MyError {
    fn from(err: ParseFloatError) -> Self {
        MyError::new(err.description())
    }
}

// and test!
fn parse_f64(s: &str, yes: bool) -> Result<f64,MyError> {
    raises_my_error(yes)?;
    let x: f64 = s.parse()?;
    Ok(x)
}
```

The first `?` is fine (a type always converts to itself with `From`) and the second `?` will convert the `ParseFloatError` to `MyError`.

And the results:

```
fn main() {
    println!(" {:?}", parse_f64("42",false));
    println!(" {:?}", parse_f64("42",true));
    println!(" {:?}", parse_f64("?42",false));
}
//  Ok(42)
//  Err(MyError { details: "borked" })
//  Err(MyError { details: "invalid float literal" })
```

Not too complicated, although a little long-winded. The tedious bit is having to write `From` conversions for all the other error types that need to play nice with `MyError` - or you simply lean on `Box<Error>`. Newcomers get confused by the multitude of ways to do the same thing in Rust; there is always another way to peel the avocado (or skin the cat, if you're feeling bloodthirsty). The price of flexibility is having many options. Error-handling for a 200 line program can afford to be simpler than for a large application. And if you ever want to package your precious droppings as a Cargo crate, then error handling becomes crucial.

Currently, the question-mark operator only works for `Result`, not `Option`, and this is a feature, not a limitation. `Option` has a `ok_or_else` which converts itself into a `Result`. For example, say we had a `HashMap` and must fail if a key isn't defined:

```
    let val = map.get("my_key").ok_or_else(|| MyError::new("my_key not defined"))?;
```

Now here the error returned is completely clear! (This form uses a closure, so the error value is only created if the lookup fails.)

simple-error for Simple Errors

The simple-error crate provides you with a basic error type based on a string, as we have defined it here, and a few convenient macros. Like any error, it works fine with `Box<Error>` :

```
#[macro_use]
extern crate simple_error;

use std::error::Error;

type BoxResult<T> = Result<T,Box<Error>>;

fn run(s: &str) -> BoxResult<i32> {
    if s.len() == 0 {
        bail!("empty string");
    }
    Ok(s.trim().parse()?)
}

fn main() {
    println!("{:?}", run("23"));
    println!("{:?}", run("2x"));
    println!("{:?}", run(""));
}
// Ok(23)
// Err(ParseIntError { kind: InvalidDigit })
// Err(StringError("empty string"))
```

`bail!(s)` expands to `return SimpleError::new(s).into();` - return early with a conversion *into* the receiving type.

You need to use `BoxResult` for mixing the `SimpleError` type with other errors, since we can't implement `From` for it, since both the trait and the type come from other crates.

error-chain for Serious Errors

For non-trivial applications have a look at the error_chain crate. A little macro magic can go a long way in Rust...

Create a binary crate with `cargo new --bin test-error-chain` and change to this directory. Edit `Cargo.toml` and add `error-chain="0.8.1"` to the end.

What **error-chain** does for you is create all the definitions we needed for manually implementing an error type; creating a struct, and implementing the necessary traits: `Display` , `Debug` and `Error` . It also by default implements `From` so strings can be converted into errors.

Our first `src/main.rs` file looks like this. All the main program does is call `run` , print out any errors, and end the program with a non-zero exit code. The macro `error_chain` generates all the definitions needed, within an `error` module - in a larger program you would put this in its own file. We need to bring everything in `error` back into global scope because our code will need to see the generated traits. By default, there will be an `Error` struct and a `Result` defined with that error.

Here we also ask for `From` to be implemented so that `std::io::Error` will convert into our error type using `foreign_links`:

```rust
#[macro_use]
extern crate error_chain;

mod errors {
    error_chain!{
        foreign_links {
            Io(::std::io::Error);
        }
    }
}
use errors::*;

fn run() -> Result<()> {
    use std::fs::File;

    File::open("file")?;

    Ok(())
}


fn main() {
    if let Err(e) = run() {
        println!("error: {}", e);

        std::process::exit(1);
    }
}
// error: No such file or directory (os error 2)
```

The 'foreign_links' has made our life easier, since the question mark operator now knows how to convert `std::io::Error` into our `error::Error`. (Under the hood, the macro is creating a `From<std::io::Error>` conversion, exactly as spelt out earlier.)

All the action happens in `run`; let's make it print out the first 10 lines of a file given as the first program argument. There may or may not be such an argument, which isn't necessarily an error. Here we want to convert an `Option<String>` into a `Result<String>`. There are two `Option` methods for doing this conversion, and I've picked the simplest one. Our `Error` type implements `From` for `&str`, so it's straightforward to make an error with a simple text message.

```
fn run() -> Result<()> {
    use std::env::args;
    use std::fs::File;
    use std::io::BufReader;
    use std::io::prelude::*;

    let file = args().skip(1).next()
        .ok_or(Error::from("provide a file"))?;

    let f = File::open(&file)?;
    let mut l = 0;
    for line in BufReader::new(f).lines() {
        let line = line?;
        println!("{}", line);
        l += 1;
        if l == 10 {
            break;
        }
    }

    Ok(())
}
```

There is (again) a useful little macro `bail!` for 'throwing' errors. An alternative to the `ok_or` method here could be:

```
    let file = match args().skip(1).next() {
        Some(s) => s,
        None => bail!("provide a file")
    };
```

Like `?` it does an *early return*.

The returned error contains an enum `ErrorKind`, which allows us to distinguish between various kinds of errors. There's always a variant `Msg` (when you say `Error::from(str)`) and the `foreign_links` has declared `Io` which wraps I/O errors:

```
fn main() {
    if let Err(e) = run() {
        match e.kind() {
            &ErrorKind::Msg(ref s) => println!("msg {}",s),
            &ErrorKind::Io(ref s) => println!("io {}",s),
        }
        std::process::exit(1);
    }
}
// $ cargo run
// msg provide a file
// $ cargo run foo
// io No such file or directory (os error 2)
```

It's straightforward to add new kinds of errors. Add an `errors` section to the `error_chain!` macro:

```
error_chain!{
    foreign_links {
        Io(::std::io::Error);
    }

    errors {
        NoArgument(t: String) {
            display("no argument provided: '{}'", t)
        }
    }

}
```

This defines how `Display` works for this new kind of error. And now we can handle 'no argument' errors more specifically, feeding `ErrorKind::NoArgument` a `String` value:

```
let file = args().skip(1).next()
    .ok_or(ErrorKind::NoArgument("filename needed".to_string()))?;
```

There's now an extra `ErrorKind` variant that you must match:

```
fn main() {
    if let Err(e) = run() {
        println!("error {}",e);
        match e.kind() {
            &ErrorKind::Msg(ref s) => println!("msg {}", s),
            &ErrorKind::Io(ref s) => println!("io {}", s),
            &ErrorKind::NoArgument(ref s) => println!("no argument {:?}", s),
        }
        std::process::exit(1);
    }
}
// cargo run
// error no argument provided: 'filename needed'
// no argument "filename needed"
```

Generally, it's useful to make errors as specific as possible, *particularly* if this is a library function! This match-on-kind technique is pretty much the equivalent of traditional exception handling, where you match on exception types in a `catch` or `except` block.

In summary, **error-chain** creates a type `Error` for you, and defines `Result<T>` to be `std::result::Result<T,Error>`. `Error` contains an enum `ErrorKind` and by default there is one variant `Msg` for errors created from strings. You define external errors with `foreign_links` which does two things. First, it creates a new `ErrorKind` variant. Second, it defines `From` on these external errors so they can be converted to our error. New error variants can be easily added. A lot of irritating boilerplate code is eliminated.

Chaining Errors

But the really cool thing that this crate provides is *error chaining*.

As a *library user*, it's irritating when a method simply just 'throws' a generic I/O error. OK, it could not open a file, fine, but what file? Basically, what use is this information to me?

`error_chain` does *error chaining* which helps solve this problem of over-generic errors. When we try to open the file, we can lazily lean on the conversion to `io::Error` using `?`, or *chain* the error.

```
// non-specific error
let f = File::open(&file)?;

// a specific chained error
let f = File::open(&file).chain_err(|| "unable to read the damn file")?;
```

Here's a new version of the program, with *no* imported 'foreign' errors, just the defaults:

```
#[macro_use]
extern crate error_chain;

mod errors {
    error_chain!{
    }

}
use errors::*;

fn run() -> Result<()> {
    use std::env::args;
    use std::fs::File;
    use std::io::BufReader;
    use std::io::prelude::*;

    let file = args().skip(1).next()
        .ok_or(Error::from("filename needed"))?;

    ///////// chain explicitly! ///////////
    let f = File::open(&file).chain_err(|| "unable to read the damn file")?;

    let mut l = 0;
    for line in BufReader::new(f).lines() {
        let line = line.chain_err(|| "cannot read a line")?;
        println!("{}", line);
        l += 1;
        if l == 10 {
            break;
        }
    }

    Ok(())
}


fn main() {
    if let Err(e) = run() {
        println!("error {}", e);

        /////// look at the chain of errors... ///////
        for e in e.iter().skip(1) {
            println!("caused by: {}", e);
        }

        std::process::exit(1);
    }
}
// $ cargo run foo
// error unable to read the damn file
// caused by: No such file or directory (os error 2)
```

So the `chain_err` method takes the original error, and creates a new error which contains the original error - this can be continued indefinitely. The closure is expected to return any value which can be *converted* into an error.

Rust macros can clearly save you a lot of typing. `error-chain` even provides a shortcut that replaces the whole main program:

```
quick_main!(run);
```

(`run` is where all the action takes place, anyway.)

Threads, Networking and Sharing

Changing the Unchangeable

If you're feeling pig-headed (as I get) you wonder if it's *ever* possible to get around the restrictions of the borrow checker.

Consider the following little program, which compiles and runs without problems.

```
// cell.rs
use std::cell::Cell;

fn main() {
    let answer = Cell::new(42);

    assert_eq!(answer.get(), 42);

    answer.set(77);

    assert_eq!(answer.get(), 77);
}
```

The answer was changed - and yet the *variable* `answer` was not mutable!

This is obviously perfectly safe, since the value inside the cell is only accessed through `set` and `get`. This goes by the grand name of *interior mutability*. The usual is called *inherited mutability*: if I have a struct value `v`, then I can only write to a field `v.a` if `v` itself is writeable. `Cell` values relax this rule, since we can change the value contained within them with `set` even if the cell itself is not mutable.

However, `Cell` only works with `Copy` types (e.g primitive types and user types deriving the `Copy` trait).

For other values, we have to get a reference we can work on, either mutable or immutable. This is what `RefCell` provides - you ask it explicitly for a reference to the contained value:

```
// refcell.rs
use std::cell::RefCell;

fn main() {
    let greeting = RefCell::new("hello".to_string());

    assert_eq!(*greeting.borrow(), "hello");
    assert_eq!(greeting.borrow().len(), 5);

    *greeting.borrow_mut() = "hola".to_string();

    assert_eq!(*greeting.borrow(), "hola");
}
```

Again, `greeting` was not declared as mutable!

The explicit dereference operator `*` can be a bit confusing in Rust, because often you don't need it - for instance `greeting.borrow().len()` is fine since method calls will dereference implicitly. But you *do* need `*` to pull out the underlying `&String` from `greeting.borrow()` or the `&mut String` from `greeting.borrow_mut()`.

Using a `RefCell` isn't always safe, because any references returned from these methods must follow the usual rules.

```
    let mut gr = greeting.borrow_mut(); // gr is a mutable borrow
    *gr = "hola".to_string();

    assert_eq!(*greeting.borrow(), "hola"); // <== we blow up here!
....
thread 'main' panicked at 'already mutably borrowed: BorrowError'
```

You cannot borrow immutably if you have already borrowed mutably! Except - and this is important - the violation of the rules happens at *runtime*. The solution (as always) is to keep the scope of mutable borrows as limited as possible - in this case, you could put a block around the first two lines here so that the mutable reference `gr` gets dropped before we borrow again.

So, this is not a feature you use without good reason, since you will *not* get a compile-time error. These types provide *dynamic borrowing* in cases where the usual rules make some things impossible.

Shared References

Up to now, the relationship between a value and its borrowed references has been clear and known at compile time. The value is the owner, and the references cannot outlive it. But many cases simply don't fit into this neat pattern. For example, say we have a `Player` struct and a `Role` struct. A `Player` keeps a vector of references to `Role` objects. There isn't a neat one-to-one relationship between these values, and persuading `rustc` to cooperate becomes nasty.

`Rc` works like `Box` - heap memory is allocated and the value is moved to it. If you clone a `Box`, it allocates a full cloned copy of the value. But cloning an `Rc` is cheap, because each time you clone it just updates a *reference count* to the *same data*. This is an old and very popular strategy for memory management, for example it's used in the Objective C runtime on iOS/MacOS. (In modern C++, it is implemented with `std::shared_ptr`.)

When a `Rc` is dropped, the reference count is decremented. When that count goes to zero the owned value is dropped and the memory freed.

```
// rc1.rs
use std::rc::Rc;

fn main() {
    let s = "hello dolly".to_string();
    let rs1 = Rc::new(s); // s moves to heap; ref count 1
    let rs2 = rs1.clone(); // ref count 2

    println!("len {}, {}", rs1.len(), rs2.len());
} // both rs1 and rs2 drop, string dies.
```

You may make as many references as you like to the original value - it's *dynamic borrowing* again. You do not have to carefully track the relationship between the value `T` and its references `&T`. There is some runtime cost involved, so it isn't the *first* solution you choose, but it makes patterns of sharing possible which would fall foul of the borrow checker. Note that `Rc` gives you immutable shared references, since otherwise that would break one of the very basic rules of borrowing. A leopard can't change its spots without ceasing to be a leopard.

In the case of a `Player`, it can now keep its roles as a `Vec<Rc<Role>>` and things work out fine - we can add or remove roles but not *change* them after their creation.

However, what if each `Player` needs to keep references to a *team* as a vector of `Player` references? Then everything becomes immutable, because all the `Player` values need to be stored as `Rc`! This is the place where `RefCell` becomes necessary. The team may be then defined as `Vec<Rc<RefCell<Player>>>`. It is now possible to change a `Player` value using `borrow_mut`, *provided* no-one has 'checked out' a reference to a `Player` at the same time. For example, say we have a rule that if something special happens to a player, then all of their team gets stronger:

```
    for p in &self.team {
        p.borrow_mut().make_stronger();
    }
```

So the application code isn't too bad, but the type signatures get a bit scary. You can always simplify them with a `type` alias:

```
type PlayerRef = Rc<RefCell<Player>>;
```

Multithreading

Over the last twenty years, there has been a shift away from raw processing speed to CPUs having multiple cores. So the only way to get the most out of a modern computer is to keep all of those cores busy. It's certainly possible to spawn child processes in the background as we saw with `Command` but there's still a synchronization problem: we don't know exactly when those children are finished without waiting on them.

There are other reasons for needing separate *threads of execution*, of course. You cannot afford to lock up your whole process just to wait on blocking i/o, for instance.

Spawning threads is straightforward in Rust - you feed `spawn` a closure which is executed in the background.

```
// thread1.rs
use std::thread;
use std::time;

fn main() {
    thread::spawn(|| println!("hello"));
    thread::spawn(|| println!("dolly"));

    println!("so fine");
    // wait a little bit
    thread::sleep(time::Duration::from_millis(100));
}
// so fine
// hello
// dolly
```

Well obviously just 'wait a little bit' is not a very rigorous solution! It's better to call `join` on the returned object - then the main thread waits for the spawned thread to finish.

```
// thread2.rs
use std::thread;

fn main() {
    let t = thread::spawn(|| {
        println!("hello");
    });
    println!("wait {:?}", t.join());
}
// hello
// wait Ok(())
```

Here's an interesting variation: force the new thread to panic.

```
    let t = thread::spawn(|| {
        println!("hello");
        panic!("I give up!");
    });
    println!("wait {:?}", t.join());
```

We get a panic as expected, but only the panicking thread dies! We still manage to print out the error message from the `join`. So yes, panics are not always fatal, but threads are relatively expensive, so this should not be seen as a routine way of handling panics.

```
hello
thread '<unnamed>' panicked at 'I give up!', thread2.rs:7
note: Run with `RUST_BACKTRACE=1` for a backtrace.
wait Err(Any)
```

The returned objects can be used to keep track of multiple threads:

```
// thread4.rs
use std::thread;

fn main() {
    let mut threads = Vec::new();

    for i in 0..5 {
        let t = thread::spawn(move || {
            println!("hello {}", i);
        });
        threads.push(t);
    }

    for t in threads {
        t.join().expect("thread failed");
    }
}
// hello 0
// hello 2
// hello 4
// hello 3
// hello 1
```

Rust insists that we handle the case where the join failed - i.e. that thread panicked. (You would typically not bail out of the main program when this happens, just note the error, retry etc)

There is no particular order to thread execution (this program gives different orders for different runs), and this is key - they really are *independent threads of execution*. Multithreading is easy; what's hard is *concurrency* - managing and synchronizing multiple threads of execution.

Threads Don't Borrow

It's possible for the thread closure to capture values, but by *moving*, not by *borrowing*!

```
// thread3.rs
use std::thread;

fn main() {
    let name = "dolly".to_string();
    let t = thread::spawn(|| {
        println!("hello {}", name);
    });
    println!("wait {:?}", t.join());
}
```

And here's the helpful error message:

```
error[E0373]: closure may outlive the current function, but it borrows `name`, which is owned by
the current function
 --> thread3.rs:6:27
  |
6 |     let t = thread::spawn(|| {
  |                           ^^ may outlive borrowed value `name`
7 |         println!("hello {}", name);
  |                              ---- `name` is borrowed here
  |
help: to force the closure to take ownership of `name` (and any other referenced variables), use
the `move` keyword, as shown:
  |     let t = thread::spawn(move || {
```

That's fair enough! Imagine spawning this thread from a function - it will exist after the function call has finished and `name` gets dropped. So adding `move` solves our problem.

But this is a *move*, so `name` may only appear in one thread! I'd like to emphasize that it *is* possible to share references, but they need to have `static` lifetime:

```
let name = "dolly";
let t1 = thread::spawn(move || {
    println!("hello {}", name);
});
let t2 = thread::spawn(move || {
    println!("goodbye {}", name);
});
```

`name` exists for the whole duration of the program (`static`), so `rustc` is satisfied that the closure will never outlive `name` . However, most interesting references do not have `static` lifetimes!

Threads can't share the same environment - by *design* in Rust. In particular, they cannot share regular references because the closures move their captured variables.

shared references are fine however, because their lifetime is 'as long as needed' - but you cannot use `Rc` for this. This is because `Rc` is not *thread safe* - it's optimized to be fast for the non-threaded case. Fortunately it is a compile error to use `Rc` here; the compiler is watching your back as always.

For threads, you need `std::sync::Arc` - 'Arc' stands for 'Atomic Reference Counting'. That is, it guarantees that the reference count will be modified in one logical operation. To make this guarantee, it must ensure that the operation is locked so that only the current thread has access. `clone` is still much cheaper than actually making a copy however.

```
// thread5.rs
use std::thread;
use std::sync::Arc;

struct MyString(String);

impl MyString {
    fn new(s: &str) -> MyString {
        MyString(s.to_string())
    }
}

fn main() {
    let mut threads = Vec::new();
    let name = Arc::new(MyString::new("dolly"));

    for i in 0..5 {
        let tname = name.clone();
        let t = thread::spawn(move || {
            println!("hello {} count {}", tname.0, i);
        });
        threads.push(t);
    }

    for t in threads {
        t.join().expect("thread failed");
    }
}
```

I"ve deliberately created a wrapper type for `String` here (a 'newtype') since our `MyString` does not implement `Clone`. But the *shared reference* can be cloned!

The shared reference `name` is passed to each new thread by making a new reference with `clone` and moving it into the closure. It's a little verbose, but this is a safe pattern. Safety is important in concurrency precisely because the problems are so unpredictable. A program may run fine on your machine, but occasionally crash on the server, usually on the weekend. Worse still, the symptoms of such problems are not easy to diagnose.

Channels

There are ways to send data between threads. This is done in Rust using *channels*. `std::sync::mpsc::channel()` returns a tuple consisting of the *receiver* channel and the *sender* channel. Each thread is passed a copy of the sender with `clone`, and calls `send`. Meanwhile the main thread calls `recv` on the receiver.

'MPSC' stands for 'Multiple Producer Single Consumer'. We create multiple threads which attempt to send to the channel, and the main thread 'consumes' the channel.

```
// thread9.rs
use std::thread;
use std::sync::mpsc;

fn main() {
    let nthreads = 5;
    let (tx, rx) = mpsc::channel();

    for i in 0..nthreads {
        let tx = tx.clone();
        thread::spawn(move || {
            let response = format!("hello {}", i);
            tx.send(response).unwrap();
        });
    }

    for _ in 0..nthreads {
        println!("got {:?}", rx.recv());
    }
}
// got Ok("hello 0")
// got Ok("hello 1")
// got Ok("hello 3")
// got Ok("hello 4")
// got Ok("hello 2")
```

There's no need to join here since the threads send their response just before they end execution, but obviously this can happen at any time. `recv` will block, and will return an error if the sender channel is disconnected. `recv_timeout` will only block for a given time period, and may return a timeout error as well.

`send` never blocks, which is useful because threads can push out data without waiting for the receiver to process. In addition, the channel is buffered so multiple send operations can take place, which will be received in order.

However, not blocking means that `Ok` does not automatically mean 'successfully delivered message'!

A `sync_channel` *does* block on send. With an argument of zero, the send blocks until the recv happens. The threads must meet up or *rendezvous* (on the sound principle that most things sound better in French.)

```
let (tx, rx) = mpsc::sync_channel(0);

let t1 = thread::spawn(move || {
    for i in 0..5 {
        tx.send(i).unwrap();
    }
});

for _ in 0..5 {
    let res = rx.recv().unwrap();
    println!("{}",res);
}
t1.join().unwrap();
```

We can easily cause an error here by calling `recv` when there has been no corresponding `send` , e.g by looping `for i in 0..4` . The thread ends, and `tx` drops, and then `recv` will fail. This will also happen if the thread panics, which causes its stack to be unwound, dropping any values.

If the `sync_channel` was created with a non-zero argument `n` , then it acts like a queue with a maximum size of `n` - `send` will only block when it tries to add more than `n` values to the queue.

Channels are strongly typed - here the channel had type `i32` - but type inference makes this implicit. If you need to pass different kinds of data, then enums are a good way to express this.

Synchronization

Let's look at *synchronization*. `join` is very basic, and merely waits until a particular thread has finished. A `sync_channel` synchronizes two threads - in the last example, the spawned thread and the main thread are completely locked together.

Barrier synchronization is a checkpoint where the threads must wait until *all* of them have reached that point. Then they can keep going as before. The barrier is created with the number of threads that we want to wait for. As before we use use `Arc` to share the barrier with all the threads.

```
// thread7.rs
use std::thread;
use std::sync::Arc;
use std::sync::Barrier;

fn main() {
    let nthreads = 5;
    let mut threads = Vec::new();
    let barrier = Arc::new(Barrier::new(nthreads));

    for i in 0..nthreads {
        let barrier = barrier.clone();
        let t = thread::spawn(move || {
            println!("before wait {}", i);
            barrier.wait();
            println!("after wait {}", i);
        });
        threads.push(t);
    }

    for t in threads {
        t.join().unwrap();
    }
}
// before wait 2
// before wait 0
// before wait 1
// before wait 3
// before wait 4
// after wait 4
// after wait 2
// after wait 3
// after wait 0
// after wait 1
```

The threads do their semi-random thing, all meet up, and then continue. It's like a kind of resumable `join` and useful when you need to farm off pieces of a job to different threads and want to take some action when all the pieces are finished.

Shared State

How can threads *modify* shared state?

Recall the `Rc<RefCell<T>>` strategy for *dynamically* doing a mutable borrow on shared references. The threading equivalent to `RefCell` is `Mutex` - you may get your mutable reference by calling `lock`. While this reference exists, no other thread can access it. `mutex` stands for 'Mutual Exclusion' - we lock a section of code so that only one thread can access it, and then unlock it. You get the lock with the `lock` method, and it is unlocked when the reference is dropped.

```
// thread9.rs
use std::thread;
use std::sync::Arc;
use std::sync::Mutex;

fn main() {
    let answer = Arc::new(Mutex::new(42));

    let answer_ref = answer.clone();
    let t = thread::spawn(move || {
        let mut answer = answer_ref.lock().unwrap();
        *answer = 55;
    });

    t.join().unwrap();

    let ar = answer.lock().unwrap();
    assert_eq!(*ar, 55);

}
```

This isn't so straightforward as using `RefCell` because asking for the lock on the mutex might fail, if another thread has panicked while holding the lock. (In this case, the documentation actually recommends just exiting the thread with `unwrap` because things have gone seriously wrong!)

It's even more important to keep this mutable borrow as short as possible, because as long as the mutex is locked, other threads are *blocked*. This is not the place for expensive calculations! So typically such code would be used like this:

```
// ... do something in the thread
// get a locked reference and use it briefly!
{
    let mut data = data_ref.lock().unwrap();
    // modify data
}
//... continue with the thread
```

Higher-Level Operations

It's better to find higher-level ways of doing threading, rather than managing the synchronization yourself. An example is when you need to do things in parallel and collect the results. One very cool crate is pipeliner which has a very straightforward API. Here's the 'Hello, World!' - an iterator feeds us inputs and we execute up to `n` of the operations on the values in parallel.

```
extern crate pipeliner;
use pipeliner::Pipeline;

fn main() {
    for result in (0..10).with_threads(4).map(|x| x + 1) {
        println!("result: {}", result);
    }
}
// result: 1
// result: 2
// result: 5
// result: 3
// result: 6
// result: 7
// result: 8
// result: 9
// result: 10
// result: 4
```

It's a silly example of course, because the operation is so cheap to calculate, but shows how easy it is to run code in parallel.

Here's something more useful. Doing network operations in parallel is very useful, because they can take time, and you don't want to wait for them *all* to finish before starting to do work.

This example is pretty crude (believe me, there are better ways of doing it) but here we want to focus on the principle. We reuse the `shell` function defined in section 4 to call `ping` on a range of IP4 addresses.

```
extern crate pipeliner;
use pipeliner::Pipeline;

use std::process::Command;

fn shell(cmd: &str) -> (String,bool) {
    let cmd = format!("{} 2>&1",cmd);
    let output = Command::new("/bin/sh")
        .arg("-c")
        .arg(&cmd)
        .output()
        .expect("no shell?");
    (
        String::from_utf8_lossy(&output.stdout).trim_right().to_string(),
        output.status.success()
    )
}

fn main() {
    let addresses: Vec<_> = (1..40).map(|n| format!("ping -c1 192.168.0.{}",n)).collect();
    let n = addresses.len();

    for result in addresses.with_threads(n).map(|s| shell(&s)) {
        if result.1 {
            println!("got: {}", result.0);
        }
    }

}
```

And the result on my home network looks like this:

```
got: PING 192.168.0.1 (192.168.0.1) 56(84) bytes of data.
64 bytes from 192.168.0.1: icmp_seq=1 ttl=64 time=43.2 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 43.284/43.284/43.284/0.000 ms
got: PING 192.168.0.18 (192.168.0.18) 56(84) bytes of data.
64 bytes from 192.168.0.18: icmp_seq=1 ttl=64 time=0.029 ms

--- 192.168.0.18 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 0.029/0.029/0.029/0.000 ms
got: PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=110 ms

--- 192.168.0.3 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 110.008/110.008/110.008/0.000 ms
got: PING 192.168.0.5 (192.168.0.5) 56(84) bytes of data.
64 bytes from 192.168.0.5: icmp_seq=1 ttl=64 time=207 ms
...
```

The active addresses come through pretty fast within the first half-second, and we then wait for the negative results to come in. Otherwise, we would wait for the better part of a minute! You can now proceed to scrape things like ping times from the output, although this would only work on Linux. `ping` is universal, but the exact output format is different for each platform. To do better we need to use the cross-platform Rust networking API, and so let's move onto Networking.

A Better Way to Resolve Addresses

If you *just* want availability and not detailed ping statistics, the `std::net::ToSocketAddrs` trait will do any DNS resolution for you:

```rust
use std::net::*;

fn main() {
    for res in "google.com:80".to_socket_addrs().expect("bad") {
        println!("got {:?}", res);
    }
}
// got V4(216.58.223.14:80)
// got V6([2c0f:fb50:4002:803::200e]:80)
```

It's an iterator because there is often more than one interface associated with a domain - there are both IPV4 and IPV6 interfaces to Google.

So, let's naively use this method to rewrite the pipeliner example. Most networking protocols use both an address and a port:

```
extern crate pipeliner;
use pipeliner::Pipeline;

use std::net::*;

fn main() {
    let addresses: Vec<_> = (1..40).map(|n| format!("192.168.0.{}:0",n)).collect();
    let n = addresses.len();

    for result in addresses.with_threads(n).map(|s| s.to_socket_addrs()) {
        println!("got: {:?}", result);
    }
}
// got: Ok(IntoIter([V4(192.168.0.1:0)]))
// got: Ok(IntoIter([V4(192.168.0.39:0)]))
// got: Ok(IntoIter([V4(192.168.0.2:0)]))
// got: Ok(IntoIter([V4(192.168.0.3:0)]))
// got: Ok(IntoIter([V4(192.168.0.5:0)]))
// ....
```

This is much faster than the ping example because it's just checking that the IP address is valid - if we fed it a list of actual domain names the DNS lookup could take some time, hence the importance of parallelism.

Suprisingly, it sort-of Just Works. The fact that everything in the standard library implements `Debug` is great for exploration as well as debugging. The iterator is returning `Result` (hence `Ok`) and in that `Result` is an `IntoIter` into a `SocketAddr` which is an enum with either a ipv4 or a ipv6 address. Why `IntoIter` ? Because a socket may have multiple addresses (e.g. both ipv4 and ipv6).

```
    for result in addresses.with_threads(n)
        .map(|s| s.to_socket_addrs().unwrap().next().unwrap())
    {
        println!("got: {:?}", result);
    }
// got: V4(192.168.0.1:0)
// got: V4(192.168.0.39:0)
// got: V4(192.168.0.3:0)
```

This also works, surprisingly enough, at least for our simple example. The first `unwrap` gets rid of the `Result` , and then we explicitly pull the first value out of the iterator. The `Result` will get bad typically when we give a nonsense address (like an address name without a port.)

TCP Client Server

Rust provides a straightforward interface to the most commonly used network protocol, TCP. It is very fault-resistant and is the base on which our networked world is built - *packets* of data are sent and received, with acknowledgement. By contrast, UDP sends packets out into the wild without acknowledgement - there's a joke that goes "I could tell you a joke about UDP but you might not get it." (Jokes about networking are only funny for a specialized meaning of the word 'funny')

However, error handling is *very* important with networking, because anything can happen, and will, eventually.

TCP works as a client/server model; the server listens on a address and a particular *network port*, and the client connects to that server. A connection is established and thereafter the client and server can communicate with a socket.

`TcpStream::connect` takes anything that can convert into a `SocketAddr`, in particular the plain strings we have been using.

A simple TCP client in Rust is easy - a `TcpStream` struct is both readable and writeable. As usual, we have to bring the `Read`, `Write` and other `std::io` traits into scope:

```rust
// client.rs
use std::net::TcpStream;
use std::io::prelude::*;

fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8000").expect("connection failed");

    write!(stream,"hello from the client!\n").expect("write failed");
 }
```

The server is not much more complicated; we set up a listener and wait for connections. When a client connects, we get a `TcpStream` on the server side. In this case, we read everything that the client has written into a string.

```rust
// server.rs
use std::net::TcpListener;
use std::io::prelude::*;

fn main() {

    let listener = TcpListener::bind("127.0.0.1:8000").expect("could not start server");

    // accept connections and get a TcpStream
    for connection in listener.incoming() {
        match connection {
            Ok(mut stream) => {
                let mut text = String::new();
                stream.read_to_string(&mut text).expect("read failed");
                println!("got '{}'", text);
            }
            Err(e) => { println!("connection failed {}", e); }
        }
    }
}
```

Here I've chosen a port number moreorless at random, but most ports are assigned some meaning.

Note that both parties have to agree on a protocol - the client expects it can write text to the stream, and the server expects to read text from the stream. If they don't play the same game, then situations can occur where one party is blocked, waiting for bytes that never come.

Error checking is important - network I/O can fail for many reasons, and errors that might appear once in a blue moon on a local filesystem can happen on a regular basis. Someone can trip over the network cable, the other party could crash, and so forth. This little server isn't very robust, because it will fall over on the first read error.

Here is a more solid server that handles the error without failing. It also specifically reads a *line* from the stream, which is done using `io::BufReader` to create an `io::BufRead` on which we can call `read_line`.

```rust
// server2.rs
use std::net::{TcpListener, TcpStream};
use std::io::prelude::*;
use std::io;

fn handle_connection(stream: TcpStream) -> io::Result<()>{
    let mut rdr = io::BufReader::new(stream);
    let mut text = String::new();
    rdr.read_line(&mut text)?;
    println!("got '{}'", text.trim_right());
    Ok(())
}

fn main() {

    let listener = TcpListener::bind("127.0.0.1:8000").expect("could not start server");

    // accept connections and get a TcpStream
    for connection in listener.incoming() {
        match connection {
            Ok(stream) => {
                if let Err(e) = handle_connection(stream) {
                    println!("error {:?}", e);
                }
            }
            Err(e) => { print!("connection failed {}\n", e); }
        }
    }
}
```

`read_line` might fail in `handle_connection`, but the resulting error is safely handled.

One-way communications like this are certainly useful - for instance. a set of services across a network which want to collect their status reports together in one central place. But it's reasonable to expect a polite reply, even if just 'ok'!

A simple example is a basic 'echo' server. The client writes some text ending in a newline to the server, and receives the same text back with a newline - the stream is readable and writeable.

```rust
// client_echo.rs
use std::io::prelude::*;
use std::net::TcpStream;

fn main() {
    let mut stream = TcpStream::connect("127.0.0.1:8000").expect("connection failed");
    let msg = "hello from the client!";

    write!(stream,"{}\n", msg).expect("write failed");

    let mut resp = String::new();
    stream.read_to_string(&mut resp).expect("read failed");
    let text = resp.trim_right();
    assert_eq!(msg,text);
}
```

The server has an interesting twist. Only `handle_connection` changes:

```rust
fn handle_connection(stream: TcpStream) -> io::Result<()>{
    let mut ostream = stream.try_clone()?;
    let mut rdr = io::BufReader::new(stream);
    let mut text = String::new();
    rdr.read_line(&mut text)?;
    ostream.write_all(text.as_bytes())?;
    Ok(())
}
```

This is a common gotcha with simple two-way socket communication; we want to read a line, so need to feed the readable stream to `BufReader` - but it *consumes* the stream! So we have to clone the stream, creating a new struct which refers to the same underlying socket. Then we have happiness.

Object-Orientation in Rust

Everyone comes from somewhere, and the chances are good that your previous programming language implemented Object-Oriented Programming (OOP) in a particular way:

- 'classes' act as factories for generating *objects* (often called *instances*) and define unique types.
- Classes may *inherit* from other classes (their *parents*), inheriting both data (*fields*) and behaviour (*methods*)
- If B inherits from A, then an instance of B can be passed to something expecting A (*subtyping*)
- An object should hide its data (*encapsulation*), which can only be operated on with methods.

Object-oriented *design* is then about identifying the classes (the 'nouns') and the methods (the 'verbs') and then establishing relationships between them, *is-a* and *has-a*.

There was a point in the old Star Trek series where the doctor would say to the captain, "It's Life, Jim, just not Life as we know it". And this applies very much to Rust-flavoured object-orientation: it comes as a shock, because Rust data aggregates (structs, enums and tuples) are dumb. You can define methods on them, and make the data itself private, all the usual tactics of encapsulation, but they are all *unrelated types*. There is no subtyping and no inheritance of data (apart from the specialized case of `Deref` coercions.)

The relationships between various data types in Rust are established using *traits*. A large part of learning Rust is understanding how the standard library traits operate, because that's the web of meaning that glues all the data types together.

Traits are interesting because there's no one-to-one correspondence between them and concepts from mainstream languages. It depends if you're thinking dynamically or statically. In the dynamic case, they're rather like Java or Go interfaces.

Trait Objects

Consider the example first used to introduce traits:

```
trait Show {
    fn show(&self) -> String;
}

impl Show for i32 {
    fn show(&self) -> String {
        format!("four-byte signed {}", self)
    }
}

impl Show for f64 {
    fn show(&self) -> String {
        format!("eight-byte float {}", self)
    }
}
```

Here's a little program with big implications:

```
fn main() {
    let answer = 42;
    let maybe_pi = 3.14;
    let v: Vec<&Show> = vec![&answer,&maybe_pi];
    for d in v.iter() {
        println!("show {}",d.show());
    }
}
// show four-byte signed 42
// show eight-byte float 3.14
```

This is a case where Rust needs some type guidance - I specifically want a vector of references to anything that implements `Show`. Now note that `i32` and `f64` have no relationship to each other, but they both understand the `show` method because they both implement the same trait. This method is *virtual*, because the actual method has different code for different types, and yet the correct method is invoked based on *runtime* information. These references are called trait objects.

And *that* is how you can put objects of different types in the same vector. If you come from a Java or Go background, you can think of `Show` as acting like an interface.

A little refinement of this example - we *box* the values. A box contains a reference to data allocated on the heap, and acts very much like a reference - it's a *smart pointer*. When boxes go out of scope and `Drop` kicks in, then that memory is released.

```
let answer = Box::new(42);
let maybe_pi = Box::new(3.14);

let show_list: Vec<Box<Show>> = vec![question,answer];
for d in &show_list {
    println!("show {}",d.show());
}
```

The difference is that you can now take this vector, pass it as a reference or give it away without having to track any borrowed references. When the vector is dropped, the boxes will be dropped, and all memory is reclaimed.

Animals

For some reason, any discussion of OOP and inheritance seems to end up talking about animals. It makes for a nice story: "See, a Cat is a Carnivore. And a Carnivore is an Animal". But I'll start with a classic slogan from the Ruby universe: "if it quacks, it's a duck". All your objects have to do is define `quack` and they can be considered to be ducks, albeit in a very narrow way.

```
trait Quack {
    fn quack(&self);
}

struct Duck ();

impl Quack for Duck {
    fn quack(&self) {
        println!("quack!");
    }
}

struct RandomBird {
    is_a_parrot: bool
}

impl Quack for RandomBird {
    fn quack(&self) {
        if ! self.is_a_parrot {
            println!("quack!");
        } else {
            println!("squawk!");
        }
    }
}

let duck1 = Duck();
let duck2 = RandomBird{is_a_parrot: false};
let parrot = RandomBird{is_a_parrot: true};

let ducks: Vec<&Quack> = vec![&duck1,&duck2,&parrot];

for d in &ducks {
    d.quack();
}
// quack!
// quack!
// squawk!
```

Here we have two completely different types (one is so dumb it doesn't even have data), and yes, they all `quack()` . One is behaving a little odd (for a duck) but they share the same method name and Rust can keep a collection of such objects in a type-safe way.

Type safety is a fantastic thing. Without static typing, you could insert a *cat* into that collection of Quackers, resulting in run-time chaos.

Here's a funny one:

```
// and why the hell not!
impl Quack for i32 {
    fn quack(&self) {
        for i in 0..*self {
            print!("quack {} ",i);
        }
        println!("");
    }
}

let int = 4;

let ducks: Vec<&Quack> = vec![&duck1,&duck2,&parrot,&int];
...
// quack!
// quack!
// squawk!
// quack 0 quack 1 quack 2 quack 3
```

What can I say? It quacks, it must be a duck. What's interesting is that you can apply your traits to any Rust value, not just 'objects'. (Since `quack` is passed a reference, there's an explicit dereference `*` to get the integer.)

However, you can only do this with a trait and a type from the same crate, so the standard library cannot be 'monkey patched', which is another piece of Ruby folk practice (and not the most wildly admired either.)

Up to this point, the trait `Quack` was behaving very much like a Java interface, and like modern Java interfaces you can have *provided* methods which supply a default implementation if you have implemented the *required* methods. (The `Iterator` trait is a good example.)

But, note that traits are not part of the *definition* of a type and you can define and implement new traits on any type, subject to the same-crate restriction.

It's possible to pass a reference to any `Quack` implementor:

```
fn quack_ref (q: &Quack) {
    q.quack();
}

quack_ref(&d);
```

And that's subtyping, Rust-style.

Since we're doing Programming Language Comparisons 101 here, I'll mention that Go has an interesting take on the quacking business - if there's a Go interface `Quack`, and a type has a `quack` method, then that type satisfies `Quack` without any need for explicit definition. This also breaks the baked-into-definition Java model, and allows compile-time duck-typing, at the cost of some clarity and type-safety.

But there is a problem with duck-typing. One of the signs of bad OOP is too many methods which have some generic name like `run`. "If it has run(), it must be Runnable" doesn't sound so catchy as the original! So it is possible for a Go interface to be *accidentally* valid. In Rust, both the `Debug` and `Display` traits define `fmt` methods, but they really mean different things.

So Rust traits allow traditional *polymorphic* OOP. But what about inheritance? People usually mean *implementation inheritance* whereas Rust does *interface inheritance*. It's as if a Java programmer never used `extend` and instead used `implements`. And this is actually recommended practice by Alan Holub. He says:

> I once attended a Java user group meeting where James Gosling (Java's inventor) was the featured speaker. During the memorable Q&A session, someone asked him: "If you could do Java over again, what would you change?" "I'd leave out classes," he replied. After the laughter died down, he explained that the real problem wasn't classes per se, but rather implementation inheritance (the extends relationship). Interface inheritance (the implements relationship) is preferable. You should avoid implementation inheritance whenever possible

---

So even in Java, you've probably been overdoing classes!

Implementation inheritance has some serious problems. But it does feel so very *convenient*. There's this fat base class called `Animal` and it has loads of useful functionality (it may even expose its innards!) which our derived class `Cat` can use. That is, it is a form of code reuse. But code reuse is a separate concern.

Getting the distinction between implementation and interface inheritance is important when understanding Rust.

Note that traits may have *provided* methods. Consider `Iterator` - you only *have* to override `next`, but get a whole host of methods free. This is similar to 'default' methods of modern Java interfaces. Here we only define `name` and `upper_case` is defined for us. We *could* override `upper_case` as well, but it isn't *required*.

```
trait Named {
    fn name(&self) -> String;

    fn upper_case(&self) -> String {
        self.name().to_uppercase()
    }
}

struct Boo();

impl Named for Boo {
    fn name(&self) -> String {
        "boo".to_string()
    }
}

let f = Boo();

assert_eq!(f.name(),"boo".to_string());
assert_eq!(f.upper_case(),"BOO".to_string());
```

This is a *kind* of code reuse, true, but note that it does not apply to the data, only the interface!

Ducks and Generics

An example of generic-friendly duck function in Rust would be this trivial one:

```
fn quack<Q> (q: &Q)
where Q: Quack {
    q.quack();
}

let d = Duck();
quack(&d);
```

The type parameter is *any* type which implements `Quack` . There's an important difference between `quack` and the `quack_ref` defined in the last section. The body of this function is compiled for *each* of the calling types and no virtual method is needed; such functions can be completely inlined. It uses the trait `Quack` in a different way, as a *constraint* on generic types.

This is the C++ equivalent to the generic `quack` (note the `const`):

```
template <class Q>
void quack(const Q& q) {
    q.quack();
}
```

Note that the type parameter is not constrained in any way.

This is very much compile-time duck-typing - if we pass a reference to a non-quackable type, then the compiler will complain bitterly about no `quack` method. At least the error is found at compile-time, but it's worse when a type is accidentally Quackable, as happens with Go. More involved template functions and classes lead to terrible error messages, because there are *no* constraints on the generic types.

You could define a function which could handle an iteration over Quacker pointers:

```
template <class It>
void quack_everyone (It start, It finish) {
    for (It i = start; i != finish; i++) {
        (*i)->quack();
    }
}
```

This would then be implemented for *each* iterator type `It`. The Rust equivalent is a little more challenging:

```
fn quack_everyone <I> (iter: I)
where I: Iterator<Item=Box<Quack>> {
    for d in iter {
        d.quack();
    }
}

let ducks: Vec<Box<Quack>> =
vec![Box::new(duck1),Box::new(duck2),Box::new(parrot),Box::new(int)];

quack_everyone(ducks.into_iter());
```

Iterators in Rust aren't duck-typed but are types that must implement `Iterator`, and in this case the iterator provides boxes of `Quack`. There's no ambiguity about the types involved, and the values must satisfy `Quack`. Often the function signature is the most challenging thing about a generic Rust function, which is why I recommend reading the source of the standard library - the implementation is often much simpler than the declaration! Here the only type parameter is the actual iterator type, which means that this will work with anything that can deliver a sequence of `Box<Duck>`, not just a vector iterator.

Inheritance

A common problem with object-oriented design is trying to force things into a *is-a* relationship, and neglecting *has-a* relationships. The GoF said "Prefer Composition to Inheritance" in their Design Patterns book, twenty-two years ago.

Here's an example: you want to model the employees of some company, and `Employee` seems a good name for a class. Then, Manager is-a Employee (this is true) so we start building our hierarchy with a `Manager` subclass of `Employee`. This isn't as smart as it seems. Maybe we got carried away with identifying important Nouns, maybe we (unconsciously) think that managers and employees are different kinds of animals? It's better for `Employee` to has-a `Roles` collection, and then a manager is just an `Employee` with more responsibilities and capabilities.

Or consider Vehicles - ranging from bicycles to 300t ore trucks. There are multiple ways to think about vehicles, road-worthiness (all-terrain, city, rail-bound, etc), power-source (electric, diesel, diesel-electric, etc), cargo-or-people, and so forth. Any fixed hierarchy of classes you create based on one aspect ignores all other aspects. That is, there are multiple possible classifications of vehicles!

Composition is more important in Rust for the obvious reason that you can't inherit functionality in a lazy way from a base class.

Composition is also important because the borrow checker is smart enough to know that borrowing different struct fields are separate borrows. You can have a mutable borrow of one field while having an immutable borrow of another field, and so forth. Rust cannot tell that a method only accesses one field, so the fields should be structs with their own methods for implementation convenience. (The *external* interface of the struct can be anything you like using suitable traits.)

A concrete example of 'split borrrowing' will make this clearer. We have a struct that owns some strings, with a method for borrowing the first string mutably.

```
struct Foo {
    one: String,
    two: String
}

impl Foo {
    fn borrow_one_mut(&mut self) -> &mut String {
        &mut self.one
    }
    ....
}
```

(This is an example of a Rust naming convention - such methods should end in `_mut`)

Now, a method for borrowing both strings, reusing the first method:

```
    fn borrow_both(&self) -> (&str,&str) {
        (self.borrow_one_mut(), &self.two)
    }
```

Which can't work! We've borrrowed mutably from `self` and *also* borrowed immmutably from `self`. If Rust allowed situations like this, then that immmutable reference can't be guaranteed not to change.

The solution is simple:

```
    fn borrow_both(&self) -> (&str,&str) {
        (&self.one, &self.two)
    }
```

And this is fine, because the borrow checker considers these to be independent borrows. So imagine that the fields were some arbitrary types, and you can see that methods called on these fields will not cause borrowing problems.

There is a restricted but very important kind of 'inheritance' with Deref, which is the trait for the 'dereference' operator `*`. `String` implements `Deref<Target=str>` and so all the methods defined on `&str` are automatically available for `String` as well! In a similar way, the methods of `Foo` can be directly called on `Box<Foo>`. Some find this a little ... magical, but it is tremendously convenient. There is a simpler language inside modern Rust, but it would not be half as pleasant to use. It really should be used for cases where there is an owned, mutable type and a simpler borrowed type.

Generally in Rust there is *trait inheritance*:

```rust
trait Show {
    fn show(&self) -> String;
}

trait Location {
    fn location(&self) -> String;
}

trait ShowTell: Show + Location {}
```

The last trait simply combines our two distinct traits into one, although it could specify other methods.

Things now proceed as before:

```rust
#[derive(Debug)]
struct Foo {
    name: String,
    location: String
}

impl Foo {
    fn new(name: &str, location: &str) -> Foo {
        Foo{
            name: name.to_string(),
            location: location.to_string()
        }
    }
}

impl Show for Foo {
    fn show(&self) -> String {
        self.name.clone()
    }
}

impl Location for Foo {
    fn location(&self) -> String {
        self.location.clone()
    }
}

impl ShowTell for Foo {}
```

Now, if I have a value `foo` of type `Foo`, then a reference to that value will satisfy `&Show`, `&Location` or `&ShowTell` (which implies both.)

Here's a useful little macro:

```rust
macro_rules! dbg {
    ($x:expr) => {
        println!("{} = {:?}",stringify!($x),$x);
    }
}
```

It takes one argument (represented by `$x`) which must be an 'expression'. We print out its value, and a *stringified* version of the value. C programmers can be a *little* smug at this point, but this means that if I passed `1+2` (an expression) then `stringify!(1+2)` is the literal string "1+2". This will save us some typing when playing with code:

```
let foo = Foo::new("Pete","bathroom");
dbg!(foo.show());
dbg!(foo.location());

let st: &ShowTell = &foo;

dbg!(st.show());
dbg!(st.location());

fn show_it_all(r: &ShowTell) {
    dbg!(r.show());
    dbg!(r.location());
}

let boo = Foo::new("Alice","cupboard");
show_it_all(&boo);

fn show(s: &Show) {
    dbg!(s.show());
}

show(&boo);

// foo.show() = "Pete"
// foo.location() = "bathroom"
// st.show() = "Pete"
// st.location() = "bathroom"
// r.show() = "Alice"
// r.location() = "cupboard"
// s.show() = "Alice"
```

This *is* object-orientation, just not the kind you may be used to.

Please note that the `Show` reference passed to `show` can not be *dynamically* upgraded to a `ShowTell`! Languages with more dynamic class systems allow you to check whether a given object is an instance of a class and then to do a dynamic cast to that type. It isn't really a good idea in general, and specifically cannot work in Rust because that `Show` reference has 'forgotten' that it was originally a `ShowTell` reference.

You always have a choice: polymorphic, via trait objects, or monomorphic, via generics constrainted by traits. Modern C++ and the Rust standard library tends to take the generic route, but the polymorphic route is not obsolete. You do have to understand the different trade-offs - generics generate the fastest code, which can be inlined. This may lead to code bloat. But not everything needs to be as *fast as possible* - it may only happen a 'few' times in the lifetime of a typical program run.

So, here's a summary:

- the role played by `class` is shared between data and traits
- structs and enums are dumb, although you can define methods and do data hiding
- a *limited* form of subtyping is possible on data using the `Deref` trait
- traits don't have any data, but can be implemented for any type (not just structs)
- traits can inherit from other traits

- traits can have provided methods, allowing interface code re-use
- traits give you both virtual methods (polymorphism) and generic constraints (monomorphism)

Example: Windows API

One of the areas where traditional OOP is used extensively is GUI toolkits. An `EditControl` or a `ListWindow` is-a `Window`, and so forth. This makes writing Rust bindings to GUI toolkits more difficult than it needs to be.

Win32 programming can be done directly in Rust, and it's a little less awkward than the original C. As soon as I graduated from C to C++ I wanted something cleaner and did my own OOP wrapper.

A typical Win32 API function is ShowWindow which is used to control the visibility of a window. Now, an `EditControl` has some specialized functionality, but it's all done with a Win32 `HWND` ('handle to window') opaque value. You would like `EditControl` to also have a `show` method, which traditionally would be done by implementation inheritance. You *not* want to have to type out all these inherited methods for each type! But Rust traits provide a solution. There would be a `Window` trait:

```rust
trait Window {
    // you need to define this!
    fn get_hwnd(&self) -> HWND;

    // and all these will be provided
    fn show(&self, visible: bool) {
        unsafe {
         user32_sys::ShowWindow(self.get_hwnd(), if visible {1} else {0})
        }
    }

    // ..... oodles of methods operating on Windows

}
```

So, the implementation struct for `EditControl` can just contain a `HWND` and implement `Window` by defining one method; `EditControl` is a trait that inherits from `Window` and defines the extended interface. Something like `ComboxBox` - which behaves like an `EditControl` *and* a `ListWindow` can be easily implemented with trait inheritance.

The Win32 API ('32' no longer means '32-bit' anymore) is in fact object-oriented, but an older style, influenced by Alan Kay's definition: objects contain hidden data, and are operated on by *messages*. So at the heart of any Windows application there's a message loop, and the various kinds of windows (called 'window classes') implement these methods with their own switch statements. There is a message called `WM_SETTEXT` but the implementation can be different: a label's text changes, but a top-level window's caption changes.

Here is a rather promising minimal Windows GUI framework. But to my taste, there are too many `unwrap` instances going on - and some of them aren't even errors. This is because NWG is exploiting the loose dynamic nature of messaging. With a proper type-safe interface, more errors are caught at compile-time.

The next edition of The Rust Programming Language book has a very good discussion on what 'object-oriented' means in Rust.

Parsing Text with Nom

Nom, (documented here) is a parser library for Rust which is well worth the initial time investment.

If you have to parse a known data format, like CSV or JSON, then it's best to use a specialized library like Rust CSV or the JSON libraries discussed in Section 4.

Likewise, for configuration files use dedicated parsers like ini or toml. (The last one is particularly cool since it integrates with the Serde framework, just as we saw with serde_json.

But if the text is not regular, or some made-up format, then you need to scan that text without writing a lot of tedious string-processing code. The suggested go-to is often regex, but regexes can be frustratingly opaque when sufficiently involved. Nom provides a way to parse text which is just as powerful and can be built up by combining simpler parsers. And regexes have their limits, for instance, don't use regexes for parsing HTML but you *could* use Nom to parse HTML. If you ever had the itch to write your own programming language, Nom is a good place for you start on that hard road to obscurity.

There are some excellent tutorials for learning Nom, but I want to start at the hello-world level to build some initial familiarity. The basic things you need to know - first, Nom is macros all the way down, and second, Nom prefers to work with byte slices, not strings. The first means that you have to be especially careful to get Nom expressions right, because the error messages are not going to be friendly. And the second means that Nom can be used for *any* data format, not just text. People have used Nom to decode binary protocols and file headers. It can also work with 'text' in encodings other than UTF-8.

Recent versions of Nom work fine with string slices, although you need to use the macros that end with `_s`.

```
#[macro_use]
extern crate nom;

named!(get_greeting<&str,&str>,
    tag_s!("hi")
);

fn main() {
    let res = get_greeting("hi there");
    println!("{:?}",res);
}
// Done(" there", "hi")
```

The `named!` macro creates functions which take some input type (`&[u8]` by default) and return the second type in angle brackets. `tag_s!` matches a literal string in the stream of characters, and its value is a string slice representing that literal. (If you wanted to work with `&[u8]` then use the `tag!` macro.)

We call the defined parser `get_greeting` with a `&str` and get back an IResult. And indeed we get back the matching value.

Look at " there" - This is the string slice left over after matching..

We want to ignore whitespace. By just wrapping the `tag!` in `ws!` we can match "hi" anywhere among spaces, tabs or newlines:

```
named!(get_greeting<&str,&str>,
    ws!(tag_s!("hi"))
);

fn main() {
    let res = get_greeting("hi there");
    println!("{:?}",res);
}
// Done("there", "hi")
```

The result is "hi" as before, and the remaining string is "there"! The spaces have been skipped.

"hi" is matching nicely, although this isn't very useful yet. Let's match *either* "hi" or "bye". The `alt!` macro ("alternate") takes parser expressions separated by `|` and matches *any* of them. Note that you can use whitespace here to make the parser function easier to read:

```
named!(get_greeting<&str>,
    ws!(alt!(tag_s!("hi") | tag_s!("bye")))
);
println!("{:?}", get_greeting(" hi "));
println!("{:?}", get_greeting(" bye "));
println!("{:?}", get_greeting("  hola "));
// Done("", "hi")
// Done("", "bye")
// Error(Alt)
```

The last match failed because there is no alternative that matches "hola".

Clearly we need to understand this `IResult` type to go further, but first let's compare this with the regex solution:

```
    let greetings = Regex::new(r"\s*(hi|bye)\s*").expect("bad regex");
    let caps = greetings.captures(" hi ").expect("match failed");
    println!("{:?}",caps);
// Captures({0: Some(" hi "), 1: Some("hi")})
```

Regular expressions are certainly more *compact*!. We needed to put '()' around the two possibilities separated by '|' so that we will *capture* the greeting and nothing else. The first result is the whole string, the second is the matched capture. ('|' is the so-called 'alternation' operator in regexes, which is the motivation for the `alt!` macro syntax.)

But this is a very simple regex, and they get complicated very quickly. Being a text mini-language, you have to escape significant characters like `*` and `(`. If I wanted to match "(hi)" or "(bye)" the regex becomes "\s*((hi|bye))\s*" but the Nom parser simply becomes `alt!(tag_s!("(hi)") | tag_s!("(bye)"))`.

It's also a heavy-weight dependency. On this fairly feeble i5 laptop, Nom examples take about 0.55 seconds to compile, which is not much more than "Hello world". But the regex examples take about 0.90s. And the stripped release build executable of the Nom example is about 0.3Mb (which is about as small as statically linked Rust programs go) versus 0.8Mb for the regex example.

What a Nom Parser returns

IResult has an interesting difference from the standard `Result` type - there are three possibilities:

- `Done` - success - you get both the result and the remaining bytes
- `Error` - failed to parse - you get an error
- `Imcomplete` - more data needed

We can write a generic `dump` function that handles any return value that can be debug-printed. This also demonstrates the `to_result` method which returns a regular `Result` - this is probably the method you will use for most cases since it returns either the returned value or an error.

```
#[macro_use]
extern crate nom;
use nom::IResult;
use std::str::from_utf8;
use std::fmt::Debug;

fn dump<T: Debug>(res: IResult<&str,T>) {
    match res {
      IResult::Done(rest, value) => {println!("Done {:?} {:?}",rest,value)},
      IResult::Error(err) => {println!("Err {:?}",err)},
      IResult::Incomplete(needed) => {println!("Needed {:?}",needed)}
    }
}


fn main() {
    named!(get_greeting<&str,&str>,
        ws!(
            alt!( tag_s!("hi") | tag_s!("bye"))
        )
    );

    dump(get_greeting(" hi "));
    dump(get_greeting(" bye hi"));
    dump(get_greeting("  hola "));

    println!("result {:?}", get_greeting(" bye  ").to_result());
}
// Done Ok("") "hi"
// Done Ok("hi") "bye"
// Err Alt
// result Ok("bye")
```

Parsers returning any unparsed text, and being able to indicate that they don't have enough input characters to decide, is very useful for stream parsing. But usually `to_result` is your friend.

Combining Parsers

Let's continue the greeting example and imagine that a greeting consists of "hi" or "bye", plus a name. `nom::alpha` matches a series of alphabetical characters. The `pair!` macro will collect the result of matching two parsers as a tuple:

```
    named!(full_greeting<&str,(&str,&str)>,
        pair!(
            get_greeting,
            nom::alpha
        )
    );

    println!("result {:?}", full_greeting(" hi Bob  ").to_result());
// result Ok(("hi", "Bob"))
```

Now, further imagine that the greeter is perhaps a little shy or doesn't know anybody's name:

let us make the name optional. Naturally, the second value of the tuple becomes an `Option`.

```
    named!(full_greeting<&str, (&str,Option<&str>)>,
        pair!(
            get_greeting,
            opt!(nom::alpha)
        )
    );

    println!("result {:?}", full_greeting(" hi Bob  ").to_result());
    println!("result {:?}", full_greeting(" bye ?").to_result());
// result Ok(("hi", Some("Bob")))
// result Ok(("bye", None))
```

Notice that it was straightforward to combine an existing parser for greetings with a parser that picks up names, and then it was easy to make that name optional. This is the great power of Nom, and it's why it's called a "parser combinator library". You can build up your complicated parsers from simpler parsers, which you can test individually. (At this point, the equivalent regex is starting to look like a Perl program: regexes do not combine well.)

However, we are not yet home and dry! `full_greeting(" bye ")` will fail with an `Imcomplete` error. Nom knows that "bye" may be followed by a name and wants us to give it more data. This is how a *streaming* parser needs to work, so you can feed it a file chunk by chunk, but here we need to tell Nom that the input is complete.

```
    named!(full_greeting<&str,(&str,Option<&str>)>,
        pair!(
            get_greeting,
            opt!(complete!(nom::alpha))
        )
    );

    println!("result {:?}", full_greeting(" bye ").to_result());
// result Ok(("bye", None))
```

Parsing Numbers

Nom provides a function `digit` which matches a series of numerical digits. So we use `map!`, to convert the string into an integer, and return the full `Result` type.

```rust
use nom::digit;
use std::str::FromStr;
use std::num::ParseIntError;

named!(int8 <&str, Result<i8,ParseIntError>>,
    map!(digit, FromStr::from_str)
);

named!(int32 <&str, Result<i32,ParseIntError>>,
    map!(digit, FromStr::from_str)
);

println!("{:?}", int8("120"));
println!("{:?}", int8("1200"));
println!("{:?}", int8("x120"));
println!("{:?}", int32("1202"));

// Done("", Ok(120))
// Done("", Err(ParseIntError { kind: Overflow }))
// Error(Digit)
// Done("", Ok(1202))
```

So what we get is a parser `IResult` containing a conversion `Result` - and sure enough, there is more than one way to fail here. Note that the body of our converting function has exactly the same code; the actual conversion depends on the return type of the function.

Integers may have a sign. We can capture integers as a pair, where the first value may be a sign, and the second value would be any digits following.

Consider:

```rust
named!(signed_digits<&str, (Option<&str>,&str)>,
    pair!(
        opt!(alt!(tag_s!("+") | tag_s!("-"))),  // maybe sign?
        digit
    )
);

println!("signed {:?}", signed_digits("4"));
println!("signed {:?}", signed_digits("+12"));
// signed Done("", (None, "4"))
// signed Done("", (Some("+"), "12"))
```

When we aren't interested in the intermediate results, but just want all the matching input, then `recognize!` is what you need.

```rust
named!(maybe_signed_digits<&str,&str>,
    recognize!(signed_digits)
);

println!("signed {:?}", maybe_signed_digits("+12"));
// signed Done("", "+12")
```

With this technique, we can recognize floating-point numbers. Again we map to string slice from the byte slice over all these matches. `tuple!` is the generalization of `pair!`, although we aren't interested in the generated tuple here. `complete!` is needed to resolve the same problem we had with incomplete greetings - "12" is a valid number without the optional floating-point part.

```
named!(floating_point<&str,&str>,
    recognize!(
        tuple!(
            maybe_signed_digits,
            opt!(complete!(pair!(
                tag_s!("."),
                digit
            ))),
            opt!(complete!(pair!(
                alt!(tag_s!("e") | tag_s!("E")),
                maybe_signed_digits
            )))
        )
    )
);
```

By defining a little helper macro, we get some passing tests. The test passes if `floating_point` matches all of the string that it is given.

```
macro_rules! nom_eq {
    ($p:expr,$e:expr) => (
        assert_eq!($p($e).to_result().unwrap(), $e)
    )
}

nom_eq!(floating_point, "+2343");
nom_eq!(floating_point, "-2343");
nom_eq!(floating_point, "2343");
nom_eq!(floating_point, "2343.23");
nom_eq!(floating_point, "2e20");
nom_eq!(floating_point, "2.0e-6");
```

(Although sometimes macros feel a *little* dirty, making your tests pretty is a fine thing.)

And then we can parse and convert floating point numbers. Here I'll throw caution to the winds and throw away the error:

```
    named!(float64<f64>,
        map_res!(floating_point, FromStr::from_str)
    );
```

Please note how it's possible to build up complicated parsers step by step, testing each part in isolation first. That's a strong advantage of parser combinators over regexes. It is very much the classic programming tactic of divide-and-rule.

Operations over Multiple Matches

We've met `pairs!` and `tuple!` which capture a fixed number of matches as Rust tuples.

There is also `many0` and `many1` - they both capture indefinite numbers of matches as vectors. The difference is that the first may capture 'zero or many' and the second 'one or many' (like the difference between the regex `*` versus `+` modifiers.) So `many1!(ws!(float64))` would parse "1 2 3" into `vec![1.0,2.0,3.0]`, but will fail on the empty string.

`fold_many0` is a *reducing* operation. The match values are combined into a single value, using a binary operator. For instance, this is how Rust people did sums over iterators before `sum` was added; this fold starts with an initial value (here zero) for the *accumulator* and keeps adding values to that accumulator using `+`.

```
let res = [1,2,3].iter().fold(0,|acc,v| acc + v);
println!("{}",res);
// 6
```

Here's the Nom equivalent:

```
named!(fold_sum<&str,f64>,
    fold_many1!(
        ws!(float64),
        0.0,
        |acc, v| acc + v
    )
);

println!("fold {}", fold_sum("1 2 3").to_result().unwrap());
//fold 6
```

Up to now, we've had to capture every expression, or just grab all matching bytes with `recognize!`:

```
named!(pointf<(f64,&[u8],f64)>,
    tuple!(
        float64,
        tag_s!(","),
        float64
    )
);

println!("got {:?}", nom_res!(pointf,"20,52.2").unwrap());
//got (20, ",", 52.2)
```

For more complicated expressions, capturing the results of all the parsers leads to rather untidy types! We can do better.

`do_parse!` lets you extract only the values you're interested in. The matches are separated with `>>` - the matches of interest are of the form `name: parser`. Finally, there's a code block in parentheses.

```
    #[derive(Debug)]
    struct Point {
        x: f64,
        y: f64
    }

    named!(pointf<Point>,
        do_parse!(
            first: float64 >>
            tag_s!(",") >>
            second: float64
            >>
            (Point{x: first, y: second})
        )
    );

    println!("got {:?}", nom_res!(pointf,"20,52.2").unwrap());
// got Point { x: 20, y: 52.2 }
```

We're not interested in that tag's value (it can only be a comma) but we assign the two float values to temporary values which are used to build a struct. The code at the end can be any Rust expression.

Parsing Arithmetic Expressions

With the necessary background established, we can do simple arithmetic expressions. This is a good example of something that really can't be done with regexes.

The idea is to build up expressions from the bottom up. Expressions consist of *terms*, which are added or subtracted. Terms consist of *factors*, which are multiplied or divided. And (for now) factors are just floating-point numbers:

```
    named!(factor<f64>,
        ws!(float64)
    );

    named!(term<&str,f64>, do_parse!(
        init: factor >>
        res: fold_many0!(
            tuple!(
                alt!(tag_s!("*") | tag_s!("/")),
                factor
            ),
            init,
            |acc, v:(_,f64)| {
                if v.0 == "*" {acc * v.1} else {acc / v.1}
            }
        )
        >> (res)
    ));

    named!(expr<&str,f64>, do_parse!(
        init: term >>
        res: fold_many0!(
            tuple!(
                alt!(tag_s!("+") | tag_s!("-")),
                term
            ),
            init,
            |acc, v:(_,f64)| {
                if v.0 == "+" {acc + v.1} else {acc - v.1}
            }
        )
        >> (res)
    ));
```

This expresses our definitions more precisely - an expression consists of at least one term, and then zero or many plus-or-minus terms. We don't collect them, but *fold* them using the appropriate operator. (It's one of those cases where Rust can't quite work out the type of the expression, so we need a type hint.) Doing it like this establishes the correct *operator precedence* - `*` always wins over `+` and so forth.

We're going to need floating-point asserts here, and there's a crate for that.

Add the line 'approx="0.1.1" to your Cargo.toml, and away we go:

```
#[macro_use]
extern crate approx;
...
    assert_relative_eq!(fold_sum("1 2 3").to_result().unwrap(), 6.0);
```

Let's define a convenient little testing macro. `stringify!` turns the expression into a string literal which we can feed into `expr` and then compare the result with how Rust would evaluate it.

```
macro_rules! expr_eq {
    ($e:expr) => (assert_relative_eq!(
        expr(stringify!($e).to_result().unwrap(),
        $e)
    )
}

expr_eq!(2.3);
expr_eq!(2.0 + 3.0 - 4.0);
expr_eq!(2.0*3.0 - 4.0);
```

This is very cool - a few lines to get an expression evaluator! But it gets better. We add an alternative to numbers in the `factor` parser - expressions contained inside parentheses:

```
named!(factor<&str,f64>,
    alt!(
        ws!(float64) |
        ws!(delimited!( tag_s!("("), expr, tag_s!(")") ))
    )
);

expr_eq!(2.2*(1.1 + 4.5)/3.4);
expr_eq!((1.0 + 2.0)*(3.0 + 4.0*(5.0 + 6.0)));
```

The coolness is that expressions are now defined *recursively* in terms of expressions!

The particular magic of `delimited!` is that parentheses may be nested - Nom makes sure the brackets match up.

We are now way past the capabilities of regular expressions, and the stripped executable at 0.5Mb is still half the size of a "hello world" regex program.

Pain Points

It is true to say that Rust is a harder language to learn than most 'mainstream' languages. There are exceptional people who don't find it so difficult, but note the strict meaning of 'exceptional' - they are *exceptions*. Many struggle at first, and then succeed. Initial difficulties aren't predictive of later competency!

We all come from somewhere, and in the case of programming languages this means previous exposure to mainstream languages like one of the 'dynamic' languages like Python or one of the 'static' languages like C++. Either way, Rust is sufficiently different to require mental retooling. Clever people with experience jump in and are disappointed that their cleverness is not immediately rewarded; people with less self-worth think they aren't 'clever' enough.

For those with dynamic language experience (in which I would include Java) everything is a reference, and all references are mutable by default. And garbage collection *does* make it easier to write memory-safe programs. A lot has gone into making the JVM pretty fast, at the cost of memory use and predicability. Often that cost is considered worth it - the old new idea that programmer productivity is more important than computer performance.

But most computers in the world - the ones that handle really important things like throttle control on cars - don't have the massive resources that even a cheap laptop has, and they need to respond to events in *real time*. Likewise, basic software infrastructure needs to be correct, robust, and fast (the old engineering trinity). Much of this is done in C and C++ which are inherently unsafe - the *total cost* of this unsafety is the thing to look at here. Maybe you knock the program together quicker, but *then* the real development starts.

System languages can't afford garbage collection, because they are the bedrock on which everything rests. They allow you to be free to waste resources as you see fit.

If there is no garbage collection, then memory must be managed in other ways. Manual memory management - I grab memory, use it, and explicitly give it back - is hard to get right. You can learn enough C to be productive and dangerous in a few weeks - but it takes years to become a good safe C programmer, checking every possible error condition.

Rust manages memory like modern C++ - as objects are destroyed, their memory is reclaimed. You can allocate memory on the heap with `Box`, but as soon as that box 'goes out of scope' at the end of the function, the memory is reclaimed. So there is something like `new` but nothing like `delete`. You create a `File` and at the end, the file handle (a precious resource) is closed. In Rust this is called *dropping*.

You need to share resources - it's very inefficient to make copies of everything - and that's where things get interesting. C++ also has references, although Rust references are rather more like C pointers - you need to say `*r` to refer to the value, you need to say `&` to pass a value as a reference.

Rust's *borrow checker* makes sure that is impossible for a reference to exist after the original value is destroyed.

Type Inference

The distinction between 'static' and 'dynamic' isn't everything. Like with most things, there are more dimensions in play. C is statically-typed (every variable has a type at compile-time) but weakly-typed (e.g. `void*` can point to *anything*); Python is dynamically-typed (the type is in the value, not the variable) but strongly-typed. Java is static/sorta strong (with reflection as convenient/dangerous escape valve) and Rust is static/strong, with no runtime reflection.

Java is famous for needing all thoses types *typed out* in numbing detail, Rust likes to *infer* types. This is generally a good idea, but it does mean that you sometimes need to work out what the actual types are. You will see `let n = 100` and wonder - what kind of integer is this? By default, it would be `i32` - a four-byte signed integer. Everyone agrees by now that C's unspecified integer types (like `int` and `long`) are a bad idea; better to be explicit. You can always spell out the type, as in `let n: u32 = 100` or let the literal force the type, as in `let n = 100u32`. But type inference goes much further than that! If you declare `let n = 100` then all `rustc` knows that `n` must be *some* integer type. If you then passed `n` to a function expecting a `u64` then that must be the type of `n`!

After that, you try to pass `n` to a function expecting `u32`. `rustc` will not let you do this, because `n` has been tied down to `u64` and it *will not* take the easy way out and convert that integer for you. This is strong typing in action - there are none of those little conversions and promotions which make your life smoother until integer overflow bites your ass suddenly. You would have to explicitly pass `n` as `n as u32` - a Rust typecast. Fortunately, `rustc` is good at breaking the bad news in an 'actionable' way - that is, you can follow the compiler's advice about fixing the problem.

So, Rust code can be very free of explicit types:

```
let mut v = Vec::new();
// v is deduced to have type Vec<i32>
v.push(10);
v.push(20);
v.push("hello") <--- just can't do this, man!
```

Not being able to put strings into a vector of integers is a feature, not a bug. The flexibility of dynamic typing is also a curse.

(If you *do* need to put integers and strings into the same vector, then Rust `enum` types are the way to do it safely.)

Sometimes you need to at least give a type *hint*. `collect` is a fantastic iterator method, but it needs a hint. Say I have a iterator returning `char`. Then `collect` can swing two ways:

```
// a vector of char ['h','e','l','l','o']
let v: Vec<_> = "hello".chars().collect();
// a string "doy"
let m: String = "dolly".chars().filter(|&c| c != 'l').collect();
```

When feeling uncertain about the type of a variable, there's always this trick, which forces `rustc` to reveal the actual type name in an error message:

```
let x: () = var;
```

`rustc` may pick an over-specific type. Here we want to put different references into a vector as `&Debug` but need to declare the type explicitly.

```
use std::fmt::Debug;

let answer = 42;
let message = "hello";
let float = 2.7212;

let display: Vec<&Debug> = vec![&message, &answer, &float];

for d in display {
    println!("got {:?}", d);
}
```

Mutable References

The rule is: only one mutable reference at a time. The reason is that tracking mutability is hard when it can happen *all over the place*. Not obvious in dinky little programs, but things can get bad in big codebases.

The further constraint is that you can't have immutable references while there's a mutable reference out. Otherwise, anybody who has those references doesn't have a guarantee that they won't change. C++ also has immutable references (e.g. `const string&`) but does *not* give you this guarantee that someone can't keep a `string&` reference and modify it behind your back.

This is a challenge if you are used to languages where every reference is mutable! Unsafe, 'relaxed' languages depend on people understanding their own programs and nobly deciding not to do Bad Things. But big programs are written by more than one person and are beyond the power of a single individual to understand in detail.

The *irritating* thing is that the borrow checker is not as smart as it could be.

```
let mut m = HashMap::new();
m.insert("one", 1);
m.insert("two", 2);

if let Some(r) = m.get_mut("one") { // <-- mutable borrow of m
    *r = 10;
} else {
    m.insert("one", 1); // can't borrow mutably again!
}
```

Clearly this does not *really* violate the Rules since if we got `None` we haven't actually borrowed anything from the map.

There are various ugly workarounds:

```
let mut found = false;
if let Some(r) = m.get_mut("one") {
    *r = 10;
    found = true;
}
if ! found {
    m.insert("one", 1);
}
```

Which is yucky, but it works because the bothersome borrow is kept to the first if-statement.

The better way here is to use `HashMap` 's entry API.

```
use std::collections::hash_map::Entry;

match m.entry("one") {
    Entry::Occupied(e) => {
        *e.into_mut() = 10;
    },
    Entry::Vacant(e) => {
        e.insert(1);
    }
};
```

The borrow checker will get less frustrating when *non-lexical lifetimes* arrive sometime this year.

The borrow checker *does* understand some important cases, however. If you have a struct, fields can be independently borrowed. So composition is your friend; a big struct should contain smaller structs, which have their own methods. Defining all the mutable methods on the big struct will lead to a situation where you can't modify things, even though the methods might only refer to one field.

With mutable data, there are special methods for treating parts of the data independently. For instance, if you have a mutable slice, then `split_at_mut` will split this into two mutable slices. This is perfectly safe, since Rust knows that the slices do not overlap.

References and Lifetimes

Rust cannot allow a situation where a reference outlives the value. Otherwise we would have a 'dangling reference' where it refers to a dead value - a segfault is inevitable.

`rustc` can often make sensible assumptions about lifetimes in functions:

```
fn pair(s: &str, ch: char) -> (&str, &str) {
    if let Some(idx) = s.find(ch) {
        (&s[0..idx], &s[idx+1..])
    } else {
        (s, "")
    }
}
fn main() {
    let p = pair("hello:dolly", ':');
    println!("{:?}", p);
}
// ("hello", "dolly")
```

This is quite safe because we cope with the case where the delimiter isn't found. `rustc` is here assuming that both strings in the tuple are borrowed from the string passed as an argument to the function.

Explicitly, the function definition looks like this:

```
fn pair<'a>(s: &'a str, ch: char) -> (&'a str, &'a str) {...}
```

What the notation says is that the output strings live *at least as long* as the input string. It's not saying that the lifetimes are the same, we could drop them at any time, just that they cannot outlive `s`.

So, `rustc` makes common cases prettier with *lifetime ellision*.

Now, if that function received *two* strings, then you would need to explicitly do lifetime annotation to tell Rust what output string is borrowed from what input string.

You always need an explicit lifetime when a struct borrows a reference:

```
struct Container<'a> {
    s: &'a str
}
```

Which is again insisting that the struct cannot outlive the reference. For both structs and functions, the lifetime needs to be declared in `<>` like a type parameter.

Closures are very convenient and a powerful feature - a lot of the power of Rust iterators comes from them. But if you store them, you have to specify a lifetime. This is because basically a closure is a generated struct that can be called, and that by default borrows its environment. Here the `linear` closure has immutable references to `m` and `c`.

```
let m = 2.0;
let c = 0.5;

let linear = |x| m*x + c;
let sc = |x| m*x.cos()
...
```

Both `linear` and `sc` implement `Fn(x: f64)->f64` but they are *not* the same animal - they have different types and sizes! So to store them you have to make a `Box<Fn(x: f64)->f64 + 'a>`.

Very irritating if you're used to how fluent closures are in Javascript or Lua, but C++ does a similar thing to Rust and needs `std::function` to store different closures, taking a little penalty for the virtual call.

Strings

It is common to feel irritated with Rust strings in the beginning. There are different ways to create them, and they all feel verbose:

```
let s1 = "hello".to_string();
let s2 = String::from("dolly");
```

Isn't "hello" *already* a string? Well, in a way. `String` is an *owned* string, allocated on the heap; a string literal "hello" is of type `&str` ("string slice") and might be either baked into the executable ("static") or borrowed from a `String`. System languages need this distinction - consider a tiny microcontroller, which has a little bit of RAM and rather more ROM. Literal strings will get stored in ROM ("read-only") which is both cheaper and consumes much less power.

But (you may say) it's so simple in C++:

```
std::string s = "hello";
```

Which is shorter yes, but hides the implicit creation of a string object. Rust likes to be explicit about memory allocations, hence `to_string`. On the other hand, to borrow from a C++ string requires `c_str`, and C strings are stupid.

Fortunately, things are better in Rust - *once* you accept that both `String` and `&str` are necessary. The methods of `String` are mostly for changing the string, like `push` adding a char (under the hood it's very much like a `Vec<u8>`). But all the methods of `&str` are also available. By the same `Deref` mechanism, a `String` can be passed as `&str` to a function - which is why you rarely see `&String` in function definitions.

There are a number of ways to convert `&str` to `String`, corresponding to various traits. Rust needs these traits to work with types generically. As a rule of thumb, anything that implements `Display` also knows `to_string`, like `42.to_string()`.

Some operators may not behave according to intuition:

```
    let s1 = "hello".to_string();
    let s2 = s1.clone();
    assert!(s1 == s2);  // cool
    assert!(s1 == "hello"); // fine
    assert!(s1 == &s2); // WTF?
```

Remember, `String` and `&String` are different types, and `==` isn't defined for that combination. This might puzzle a C++ person who is used to references being almost interchangeable with values. Furthermore, `&s2` doesn't *magically* become a `&str`, that's a *deref coercion* which only happens when assigning to a `&str` variable or argument. (The explicit `s2.as_str()` would work.)

However, this more genuinely deserves a WTF:

```
let s3 = s1 + s2;  // <--- no can do
```

You cannot concatenate two `String` values, but you can concatenate a `String` with a `&str`. You furthermore cannot concatenate a `&str` with a `String`. So mostly people don't use `+` and use the `format!` macro, which is convenient but not so efficient.

Some string operations are available but work differently. For instance, languages often have a `split` method for breaking up a string into an array of strings. This method for Rust strings returns an *iterator*, which you can *then* collect into a vector.

```
let parts: Vec<_> = s.split(',').collect();
```

This is a bit clumsy if you are in a hurry to get a vector. But you can do operations on the parts *without* allocating a vector! For instance, length of largest string in the split?

```
let max = s.split(',').map(|s| s.len()).max().unwrap();
```

(The `unwrap` is because an empty iterator has no maximum and we must cover this case.)

The `collect` method returns a `Vec<&str>`, where the parts are borrowed from the original string - we only need allocate space for the references. There is no method like this in C++, but until recently it would have to individually allocate each substring. (C++ 17 has `std::string_view` which behaves like a Rust string slice.)

A Note on Semicolons

Semicolons are *not* optional, but usually left out in the same places as in C, e.g. after `{}` blocks. They also aren't needed after `enum` or `struct` (that's a C peculiarity.) However, if the block must have a *value*, then the semi-colons are dropped:

```
    let msg = if ok {"ok"} else {"error"};
```

Note that there must be a semi-colon after this `let` statement!

If there were semicolons after these string literals then the returned value would be `()` (like `Nothing` or `void`). It's common error when defining functions:

```
fn sqr(x: f64) -> f64 {
    x * x;
}
```

`rustc` will give you a clear error in this case.

C++-specific Issues

Rust value semantics are Different

In C++, it's possible to define types which behave exactly like primitives and copy themselves. In addition, a move constructor can be defined to specify how a value can be moved out of a temporary context.

In Rust, primitives behave as expected, but the `Copy` trait can only be defined if the aggregate type (struct, tuple or enum) itself contains only copyable types. Arbitrary types may have `Clone` , but you have to call the `clone` method on values. Rust requires any allocation to be explicit and not hide in copy constructors or assignment operators.

So, copying and moving is always defined as just moving bits around and is not overrideable.

If `s1` is a non `Copy` value type, then `s2 = s1;` causes a move to happen, and this *consumes* `s1`! So, when you really want a copy, use `clone` .

Borrowing is often better than copying, but then you must follow the rules of borrowing. Fortunately, borrowing *is* an overridable behaviour. For instance, `String` can be borrowed as `&str` , and shares all the immutable methods of `&str` . *String slices* are very powerful compared to the analogous C++ 'borrowing' operation, which is to extract a `const char*` using `c_str` . `&str` consists of a pointer to some owned bytes (or a string literal) and a *size*. This leads to some very memory-efficient patterns. You can have a `Vec<&str>` where all the strings have been borrowed from some underlying string - only space for the vector needs to be allocated:

For example, splitting by whitespace:

```
fn split_whitespace(s: &str) -> Vec<&str> {
    s.split_whitespace().collect()
}
```

Likewise, a C++ `s.substr(0,2)` call will always copy the string, but a slice will just borrow: `&s[0..2]`.

There is an equivalent relationship between `Vec<T>` and `&[T]`.

Shared References

Rust has *smart pointers* like C++ - for instance, the equivalent of `std::unique_ptr` is `Box`. There's no need for `delete`, since any memory or other resources will be reclaimed when the box goes out of scope (Rust very much embraces RAII).

```
let mut answer = Box::new("hello".to_string());
*answer = "world".to_string();
answer.push('!');
println!("{} {}", answer, answer.len());
```

People find `to_string` irritating at first, but it is *explicit*.

Note the explicit dererefence `*`, but methods on smart pointers don't need any special notation (we do not say `(*answer).push('!')`)

Obviously, borrowing only works if there is a clearly defined owner of the original content. In many designs this isn't possible.

In C++, this is where `std::shared_ptr` is used; copying just involves modifying a reference count on the common data. This is not without cost, however:

- even if the data is read-only, constantly modifying the reference count can cause cache invalidation
- `std::shared_ptr` is designed to be thread-safe and carries locking overhead as well

In Rust, `std::rc::Rc` also acts like a shared smart pointer using reference-counting. However, it is for immutable references only! If you want a thread-safe variant, use `std::sync::Arc` (for 'Atomic Rc'). So Rust is being a little awkward here in providing two variants, but you get to avoid the locking overhead for non-threaded operations.

These must be immutable references because that is fundamental to Rust's memory model. However, there's a get-out card: `std::cell::RefCell`. If you have a shared reference defined as `Rc<RefCell<T>>` then you can mutably borrow using its `borrow_mut` method. This applies the Rust borrowing rules *dynamically* - so e.g. any attempt to call `borrow_mut` when a borrow was already happening will cause a panic.

This is still *safe*. Panics will happen *before* any memory has been touched inappropriately! Like exceptions, they unroll the call stack. So it's an unfortunate word for such a structured process - it's an ordered withdrawal rather than a panicked retreat.

The full `Rc<RefCell<T>>` type is clumsy, but the application code isn't unpleasant. Here Rust (again) is prefering to be explicit.

If you wanted thread-safe access to shared state, then `Arc<T>` is the only *safe* way to go. If you need mutable access, then `Arc<Mutex<T>>` is the equivalent of `Rc<RefCell<T>>`. `Mutex` works a little differently than how it's usually defined: it is a container for a value. You get a *lock* on the value and can then modify it.

```
let answer = Arc::new(Mutex::new(10));

// in another thread
..
{
  let mut answer_ref = answer.lock().unwrap();
  *answer_ref = 42;
}
```

Why the `unwrap`? If the previous holding thread panicked, then this `lock` fails. (It's one place in the documentation where `unwrap` is considered a reasonable thing to do, since clearly things have gone seriously wrong. Panics can always be caught on threads.)

It's important (as always with mutexes) that this exclusive lock is held for as little time as possible. So it's common for them to happen in a limited scope - then the lock ends when the mutable reference goes out of scope.

Compared with the apparently simpler situation in C++ ("use shared_ptr dude") this seems awkward. But now any *modifications* of shared state become obvious, and the `Mutex` lock pattern forces thread safety.

Like everything, use shared references with caution.

Iterators

Iterators in C++ are defined fairly informally; they involve smart pointers, usually starting with `c.begin()` and ending with `c.end()`. Operations on iterators are then implemented as stand-alone template functions, like `std::find_if`.

Rust iterators are defined by the `Iterator` trait; `next` returns an `Option` and when the `Option` is `None` we are finished.

The most common operations are now methods. Here is the equivalent of `find_if`. It returns an `Option` (case of not finding is `None`) and here the `if let` statement is convenient for extracting the non-`None` case:

```
let arr = [10, 2, 30, 5];
if let Some(res) = arr.find(|x| x == 2) {
    // res is 2
}
```

Unsafety and Linked Lists

It's no secret that parts of the Rust stdlib are implemented using `unsafe`. This does not invalidate the conservative approach of the borrow checker. Remember that "unsafe" has a particular meaning - operations which Rust cannot fully verify at compile time. From Rust's perspective, C++ operates in unsafe mode all the time! So if a large application needs a few dozen lines of unsafe code, then that's fine, since these few lines can be carefully checked by a human. Humans are not good at checking 100Kloc+ of code.

I mention this, because there appears to be a pattern: an experienced C++ person tries to implement a linked list or a tree structure, and gets frustrated. Well, a double-linked list *is* possible in safe Rust, with `Rc` references going forward, and `Weak` references going back. But the standard library gets more performance out of using... pointers.