Rust out your C (w/FP Bent)

Carol (Nichols || Goulding)
@carols10cents

photo by flickr user celinet, cropped https://flic.kr/p/8K8m9

This is an experience report about what happened when I rewrote a C library in Rust.
Legacy code, rewrite considerations, tradeoffs.
Rust is a multi-paradigm lang, but has FP influences, if you have a C lib that you wish was more functional, Rust *might* be a good candidate.

Agenda

1 **Caveats**
2 **Background**
3 **Techniques**
4 **Benefits?**

2

DO NOT

My first caveat: if you're thinking about rewriting a C library in Rust, do not. This is my cat disapproving of you. A rewrite is not something to take lightly, especially if the thing you want to rewrite is doing the job it needs to do just fine!

Bad reasons to rewrite your C in Rust

- Cool kids
- I feel like it
- I'm bored
- Job security
- Carol said

Please don't leave this talk thinking "oh, Rust is the new hotness, I should tell my boss we have to stop working on everything to go off for a few months and rewrite everything in Rust!" No!!! Also, if you're the only person on your team interested in learning Rust, it may get you some job security, but it's not the most responsible thing to do in that situation.

Good reasons to rewrite your C in Rust	• Performance
	• Safety
	• Lower maintenance costs
	• Expand # of maintainers
	• For fun, not work

5

If you have some code in C or another language, and need to change it, or it's slow, or it crashes a lot, or no one understands it anymore, THEN maybe a rewrite in Rust would be a good fit. I would also posit that more people are *able* to write production Rust than production C, so if your team *is* willing to learn Rust, it might actually expand the number of maintainers.

And then there's how I'm getting away with it— if you're doing a rewrite of something for fun, that's totally fine too.

I FIGHT FOR THE USERS

Basically, keep the users of your software in mind when you're doing a rewrite. What will be the benefits for them?

Any time you rewrite, code will get better.

Another caveat— Rust is not magic. Some of the effects I'm going to show you would have happened if I had chosen a different language to rewrite this library in, because rewriting gives you the benefit of hindsight.

Things I knew before

- Rust
- Legacy code
- Testing

A confessional caveat— when I proposed this talk, these were the relevant things I knew…
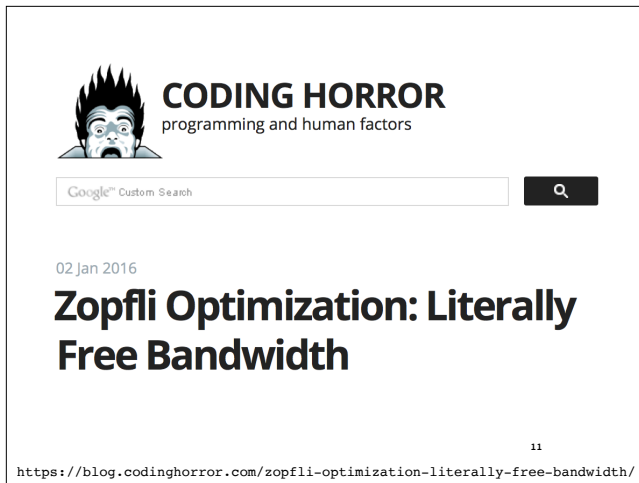
Things I DID NOT know before

- C
- FFI
- zopfli

And these were the things I did NOT know. I haven't written C since college, I've been pretty terrified by it actually. I knew there was a foreign function interface between C and Rust but I hadn't tried it out. And the library I chose— I was not at all familiar with how it worked! But I learned and did everything I'm going to go over, which means YOU CAN TOO!

Background: zopfli

Which brings us to the background info! Zopfli is a compression tool, like gzip or zlib. It was written by people at Google, and from what I can tell it's actually pretty good C code. I didn't have any trouble building it; there aren't any dependencies to set up. It's about 5K lines of C.

CODING HORROR
programming and human factors

Google™ Custom Search

02 Jan 2016

Zopfli Optimization: Literally Free Bandwidth

11

https://blog.codinghorror.com/zopfli-optimization-literally-free-bandwidth/

The reason you might want to use zopfli is that it creates smaller compressed files than gzip or zlib, but still follows the DEFLATE protocol,

This is a DEFLATE file, I know this!

which is what browsers know how to deflate. So using zopfli on your css, js, etc once can save bandwidth on every time those assets are transferred.
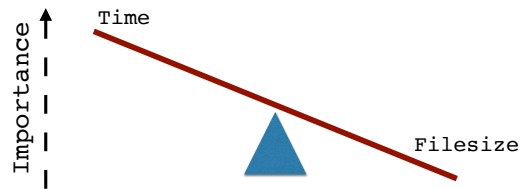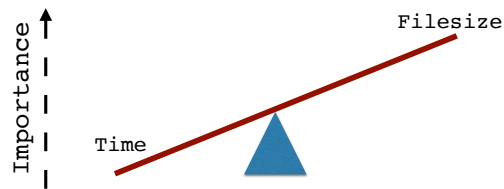
You might NOT want to use zopfli because it does take longer. Compared to gzip, which makes the tradeoff of taking less time to compress but potentially producing files that aren't as small as they could be,

zopfli makes the other tradeoff, with the use case being you'll take the extra time to compress your files once and then save the extra file size every time your files are transferred.

I found out about zopfli in the context of Rails' asset pipeline— gzipping was actually disabled as part of the asset compilation process, and in the discussion around re-enabling it, zopfli was considered. If the filesize of all the Rails assets in all the world was smaller, that would save the internet a lot of bandwidth!

But is it too slow? Would Rails developers complain? Could it be faster? And is the zopfli ruby gem a reasonable dependency to ask people to install? Would it work with JRuby? So now the tradeoffs are looking less like a seesaw and more like…
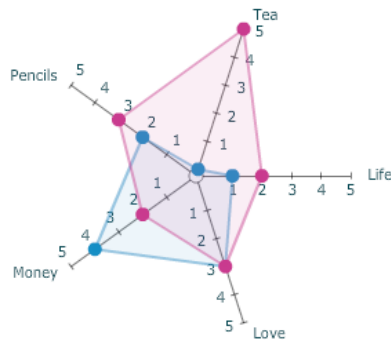
example radar chart demoing MIT Licensed flex component from
http://www.boost.co.nz/blog/2009/05/flex-radar-chart-component/

17

… comparing shapes in multiple dimensions. I thought Rust might be a good fit here, since it is fast and its cross-compilation story is good. I have no idea if it works better with JRuby though! And I had (and have) no idea if what I ended up with is a better choice than the C implementation of zopfli, but I figured this was as good a situation as any to try.

Techniques

18

The first technique I want to emphasize is a general legacy code technique—

IF IT AIN'T BAROQUE

DON'T FIX IT

https://commons.wikimedia.org/wiki/File:Johann_Sebastian_Bach.jpg
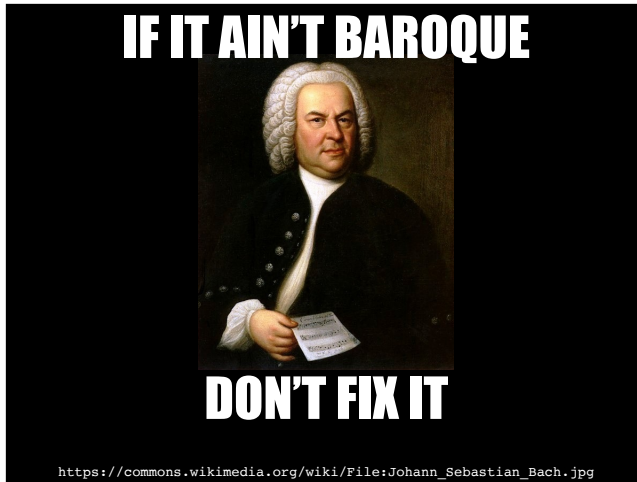
you want to make sure that your rewrite isn't breaking anything and introducing new bugs!! But legacy code often doesn't come with automated tests, and zopfli did not either.

Golden Master Tests

20

So the first thing I did was make a golden master. In the case of zopfli, this meant I collected some files, compressed them and saved the results, and then with every change to the code, I could make sure I got the exact same compressed files. It's not as fine grained as unit tests, so it can be hard to figure out what the problem is when the files don't match, but it's better than not having any tests. An improvement I could have done but didn't would have been to do code coverage checks to make sure the files I picked were executing as much of the code as possible.

Slide 21

```
Remove a function from C

size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned* d_lengths) {
  size_t result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                        i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

Now that we've got an empty Rust project being built and linked with C, let's say we want to move a function's implementation from C to Rust. This is a function from zopfli that I moved over. First, copy the whole C function.

Slide 22

```
Remove a function from C

extern size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned*
d_lengths);
```

Then remove the body, leaving only the signature. Add "extern" to the beginning and a semicolon at the end. This will let the remaining C code call this function.

Add a function to Rust

```
size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned* d_lengths) {
  size_t result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                        i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

23

Now paste the C code into a Rust code file.

Add a function to Rust

```
#[no_mangle]
#[allow(non_snake_case)]
size_t CalculateTreeSize(const unsigned* ll_lengths, const unsigned* d_lengths) {
  size_t result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                        i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

24

We have to tell Rust not to mangle the function name so that the C code can link to the symbol. Rust also isn't going to be happy about the non snake case name, but we're going to leave it this way for now, so turn off the warning about the naming convention for this function.

Slide 25

Add a function to Rust

```
#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(const unsigned* ll_lengths, const unsigned*
d_lengths) -> size_t {
  size_t result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                        i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

25

We'll need a pub extern fn at the beginning of the signature, and we'll move the return type that was there to the end after an arrow.

Slide 26

Add a function to Rust

```
#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: const unsigned*, d_lengths: const
unsigned*) -> size_t {
  size_t result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                        i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

26

At this point you could start trying to compile it and just let the Rust compiler tell you what needs fixed! Here's what it would say:
Argument names go before the types in Rust, so switch those around.

Slide 27

Add a function to Rust

```rust
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
  size_t result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    size_t size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                          i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

27

We need to change the types a little bit— luckily there's this Rust library called libc that defines aliases to map C types to Rust types, but we still need to change "unsigned" to "c_uint".

Slide 28

Add a function to Rust

```rust
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
  let mut result = 0;
  int i;

  for(i = 0; i < 8; i++) {
    let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                          i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

28

Next change the variable declarations to use "let" and make ones that need to be mutable

Add a function to Rust

```rust
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
  let mut result = 0;

  for i in 0..8 {
    let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                     i & 1, i & 2, i & 4);
    if (result == 0 || size < result) result = size;
  }

  return result;
}
```

29

Change the for loop to the Rust way of doing for loops, with a range

---

Add a function to Rust

```rust
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
  let mut result = 0;

  for i in 0..8 {
    let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                     i & 1, i & 2, i & 4);
    if result == 0 || size < result {
      result = size;
    }
  }

  return result;
}
```

30

Remove parens from the if condition and add braces

Add a function to Rust

```rust
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
  let mut result = 0;

  for i in 0..8 {
    let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                          i & 1, i & 2, i & 4);
    if result == 0 || size < result {
      result = size;
    }
  }

  result
}
```

31

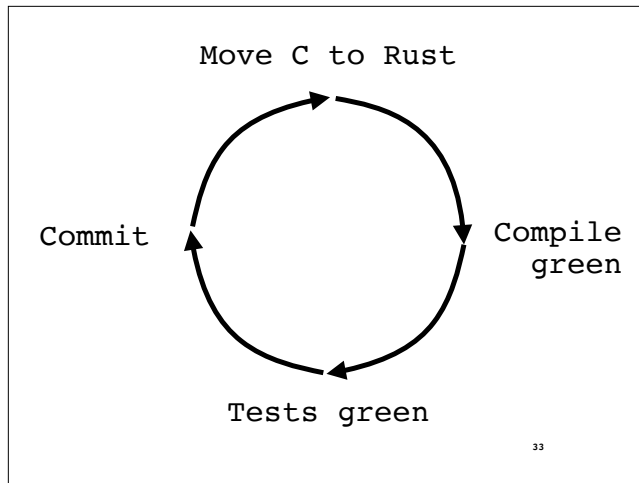and in Rust we don't need to return if we're returning the last expression, we just need to not have a semicolon.

At this point, golden master tests should pass again. Do the least amount possible.

Incremental

32

And then you repeat! Doing this incrementally, function by function, struct by struct is important. Not only do you only have to learn one little piece at a time, it's like a red-green-refactor TDD cycle,

```
          Move C to Rust


Commit                    Compile
                           green


          Tests green
                              33
```

except it's more like moving a function to rust, getting it to compile, getting the tests to pass, then committing. Taking lots of little baby steps and committing after each is great because when someone asks that ever-present question during rewrites,

"Are you done yet?"

34
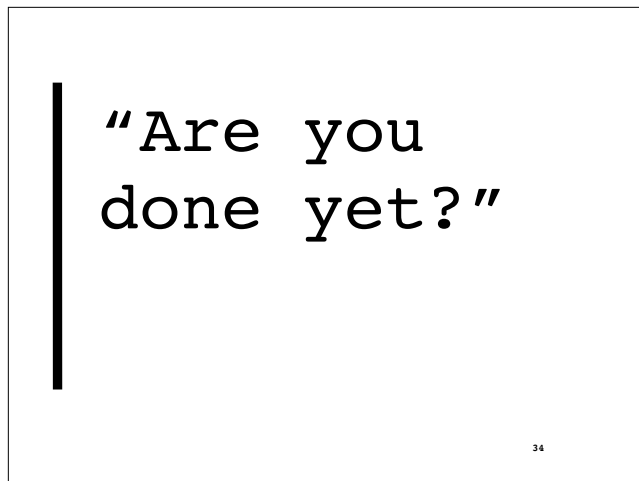
as long as you only commit and push the code in a state where it's compiling and passing tests, you'll be in this sort of

superposition of done and not-done. You're done because you could ship what you have right now and switch to something else without having unrealized benefits from a sunk cost. But you're not done because the whole library isn't rewritten. So it gives you more flexibility in how you prioritize the rewrite if other more important tasks come up.

What if it doesn't pass the tests?

The other benefit of committing often after small changes is that, since you don't have fine-grained unit tests to tell you where the problem is, if your tests fail and you don't know why,

git checkout! take a smaller step.

it's not that big a deal to go back to a state where everything WAS working and try again. Probably try again with an even smaller step— many times when I broke everything, I was overconfident and was changing too much at once.

smaller

- Extract functions
- Make the C more like Rust first
- Don't make the Rust idiomatic at all, even when it seems easy

In this context, by smaller step I mean extracting functions from large functions so that you can move smaller pieces from C to rust, maybe making the C more like Rust first and getting that to pass the tests so that the actual change is more mechanical (ex: change complex C `for` loops to `while` loops more easily translated), and leave the Rust exactly like C even when it looks really simple to change pointers to slices— don't do it until after you have the tests passing!

Slide 39

Move C to Rust

Commit ← → Compile green

Make idiomatic ← → Tests green

Commit

39
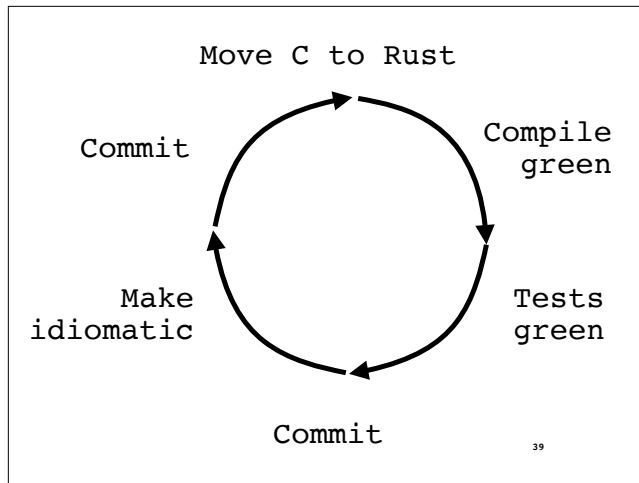
Add another step to our cycle— making it idiomatic Rust, can do more when C doesn't call it anymore

---

Slide 40

```
Idiomaticize
use libc::{size_t, c_uint};

#[no_mangle]
#[allow(non_snake_case)]
pub extern fn CalculateTreeSize(ll_lengths: *const c_uint, d_lengths: *const
c_uint) -> size_t {
  let mut result = 0;

  for i in 0..8 {
    let size = EncodeTreeNoOutput(ll_lengths, d_lengths,
                    i & 1, i & 2, i & 4);
    if result == 0 || size < result {
      result = size;
    }
  }

  result
}
```

40

Once we've moved all the places that call this function to Rust, we can get rid of the parts that let us call it from C

Idiomaticize

```rust
pub fn calculate_tree_size(ll_lengths: &[u32], d_lengths: &[u32]) -> usize {
  let mut result = 0;

  for i in 0..8 {
    let size = encode_tree_no_output(ll_lengths, d_lengths,
                        i & 1, i & 2, i & 4);
    if result == 0 || size < result {
      result = size;
    }
  }

  result
}
```

Make the function name and types be rusty, take a slice instead of pointers (i'm assuming we've moved encode_tree_no_output over too)

And then we can start looking at what the function is doing – calculating stuff and returning the min using a mutable variable and a for loop over a range with side effects…

Iterators!

Enter iterators! An iterator in Rust is what gives you your map, fold, reduce, etc, and they're used often.

Idiomaticize

```rust
pub fn calculate_tree_size(ll_lengths: &[u32], d_lengths: &[u32]) -> usize {
    (0..8).map(|i| {
        encode_tree_no_output(ll_lengths, d_lengths,
                              i & 1 > 0, i & 2 > 0, i & 4 > 0)
    }).min().unwrap_or(0)
}
```

We can make that way more functional! & easy to read.
- has the word "min" in there
- has the default value if we can't find a min
- next step: precompute the bool args, maybe.

Let's look at some other examples of FP ideas in Rust!

yinz ready for lots of code?

```
typedef double CostModelFun(unsigned litlen, unsigned dist, void* context);

/* type: CostModelFun */
static double GetCostFixed(unsigned litlen, unsigned dist, void* unused) {…}

/* type: CostModelFun */
static double GetCostStat(unsigned litlen, unsigned dist, void* context) {…}

static double GetCostModelMinCost(CostModelFun* costmodel, void* costcontext) {
    // . . .
    double c = costmodel(i, 1, costcontext);
    // . . .
}

void ZopfliLZ77OptimalFixed(…) {
    // . . .
    GetCostModelMinCost(GetCostFixed, 0);
    // . . .
}

void ZopfliLZ77Optimal(…) {
    // . . .
    GetCostModelMinCost(GetCostStat, (void*)&stats);
    // . . .
}
```

45

—   defined closure "type"
— two variants, one that used the last arg
— a function that took a fn as an arg
— 2 fns that called the variants

```
fn get_cost_fixed(litlen: c_uint, dist: c_uint, _unused: c_void) -> c_double {…}

fn get_cost_stat(litlen: c_uint, dist: c_uint, context: *const c_void) -> c_double
{…}


fn get_cost_model_min_cost(
    costmodel: fn(c_uint, c_uint, *const c_void) -> c_double,
    costcontext: *const c_void) -> c_double {
    // . . .
    let c = costmodel(i, 1, costcontext);
    // . . .
}

fn lz77_optimal_fixed(…) {
    // . . .
    get_cost_model_min_cost(get_cost_fixed, ptr::null());
    // . . .
}

fn lz77_optimal(…) {
    // . . .
    let stats_ptr: *const SymbolStats = &stats;
    get_cost_model_min_cost(get_cost_stat, stats_ptr as *const c_void);
    // . . .
}
```

Translate directly.
Nicer: have the type of the closure in the sig of the function taking it as an arg.
Still have void pointer silliness.

```
fn get_cost_fixed(litlen: c_uint, dist: c_uint, _unused: Option<&SymbolStats>) ->
c_double {…}

fn get_cost_stat(litlen: c_uint, dist: c_uint, context: Option<&SymbolStats>) ->
c_double {…}


fn get_cost_model_min_cost(
    costmodel: fn(c_uint, c_uint, Option<&SymbolStats>) -> c_double,
    costcontext: Option<&SymbolStats>) -> c_double {
    // . . .
    let c = costmodel(i, 1, costcontext);
    // . . .
}


fn lz77_optimal_fixed(…) {
    // . . .
    get_cost_model_min_cost(get_cost_fixed, None);
    // . . .
}

fn lz77_optimal(…) {
    // . . .
    get_cost_model_min_cost(get_cost_stat, Some(&stats));
    // . . .
}
```

A little better than null pointers would be options, but this still isn't great because it's ALWAYS none with get_cost_fixed and ALWAYS some with get_cost_stat

```
- fn get_cost_fixed(litlen: c_uint, dist: c_uint, _unused: Option<&SymbolStats>) ->
c_double {…}
+ fn get_cost_fixed(litlen: c_uint, dist: c_uint) -> c_double {…}

- fn get_cost_stat(litlen: c_uint, dist: c_uint, context: Option<&SymbolStats>) ->
c_double {…}
+ fn get_cost_stat(litlen: c_uint, dist: c_uint, context: &SymbolStats) -> c_double
{…}


- fn get_cost_model_min_cost(
-     costmodel: fn(c_uint, c_uint, Option<&SymbolStats>) -> c_double,
-     costcontext: Option<&SymbolStats>) -> c_double {
+ fn get_cost_model_min_cost<F>(costmodel: F) -> c_double
+     where F: Fn(c_uint, c_uint) -> c_double
+ {
    // . . .
-     let c = costmodel(i, 1, costcontext);
+     let c = costmodel(i, 1);
    // . . .
}
```

How could we fix that?
1. Get rid of the unused argument from the first fn, always have it in the 2nd fn
2. Change the signature of the function taket as an argument to only take the two ints

```rust
fn get_cost_fixed(litlen: c_uint, dist: c_uint) -> c_double {…}

fn get_cost_stat(litlen: c_uint, dist: c_uint, context: &SymbolStats) -> c_double
{…}

fn get_cost_model_min_cost<F>(costmodel: F) -> c_double
    where F: Fn(c_uint, c_uint) -> c_double
{
    // . . .
    let c = costmodel(i, 1);
    // . . .
}

 fn lz77_optimal_fixed(…) {
     // . . .
-    get_cost_model_min_cost(get_cost_fixed, None);
+    get_cost_model_min_cost(get_cost_fixed);
     // . . .
 }

 fn lz77_optimal(…) {
     // . . .
-    get_cost_model_min_cost(get_cost_stat, Some(&stats));
+    get_cost_model_min_cost(|a, b| get_cost_stat(a, b, &stats));
     // . . .
 }
```

Here we have just the new versions from the previous slide.
1. The use of the first function is trivial
2. For the 2nd fn, we can wrap the call of get_cost_stat in a closure that always uses stats.

Isn't that neat and nicer??

```
Abstract
Data Types!
```

Such as the Option type I had for a minute there — means no null pointers, generics

Closures!

With strong typing at compile time

Traits!
(not covered)

I wanted to mention Rust's traits even though I didn't get into them today— they're similar to Haskell's type classes. Your homework is to check them out :)

Benefits?

53

Enough code – let's look at some pretty graphs. So — did I see any benefits from the rewrite in Rust?

Safety

54

Fewer possibilities of null pointers— you could just forget to check once in C and oopsie! Most places, I was able to change this so that Rust could guarantee I would have something.

Clarity (more functional)

Could name concepts better, like min
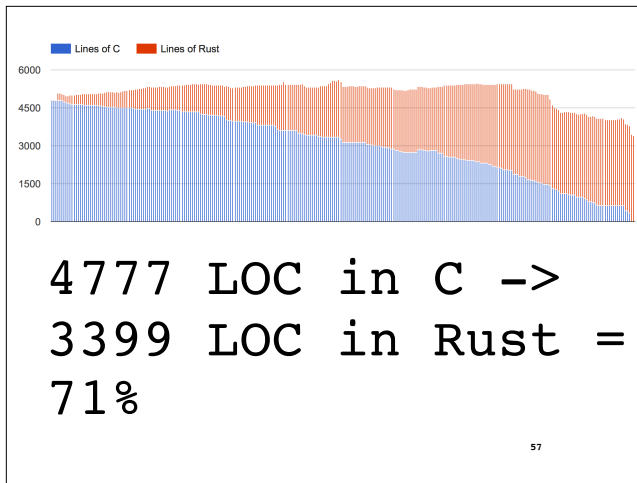Could separate and couple code more intuitively

Less code

Using iterators and closures can be much more compact.

*yes yes, compact does not always equal clearer

Rust stdlib provides a lot more functionality that you don't have to maintain

```
4777 LOC in C ->
3399 LOC in Rust =
71%
```

This is the graph over time of how much code there was in C and Rust as I worked; by the end when I got it all into Rust, it was 71% of the lines and could probably continue to go down if I were to make the Rust more idiomatic and recognize other things the stdlib does.
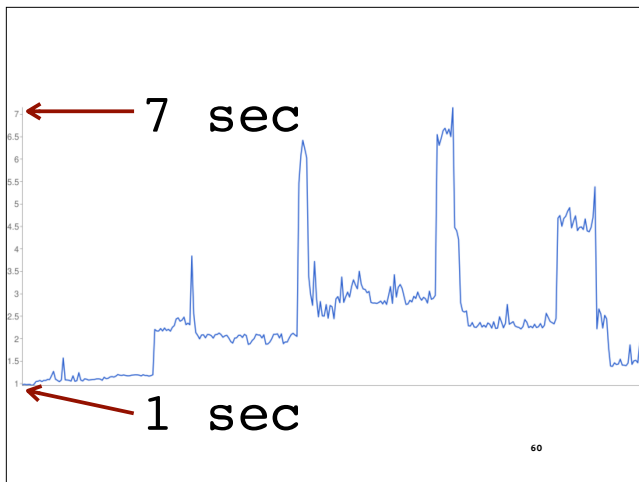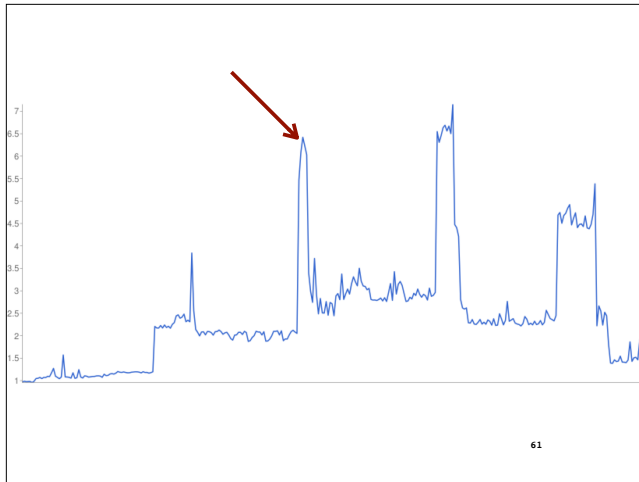

Performance

The big question: is Rust as fast as C? (Or, really, is the Rust Carol translated as fast as pretty good C?)
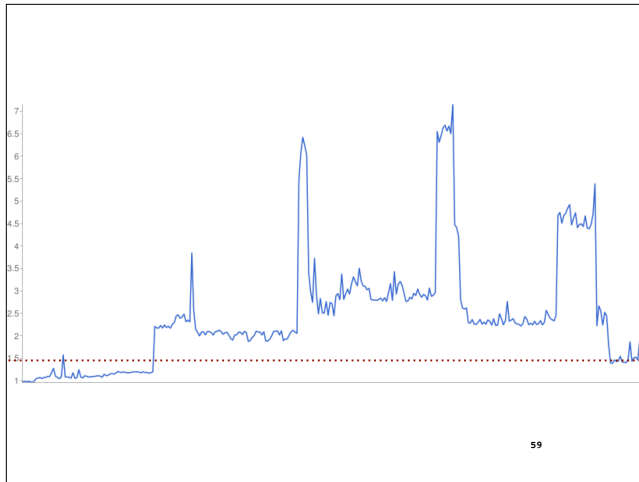
Well, I haven't gotten it back down to C speeds yet, but it's pretty close. This is the performance at each commit of mine.

C code took about 1 sec on my golden master tests. This graph is rough, only ran the tests once per commit.

Digging into some of the spikes and why the time went up, let's talk about this one

Bad news :(

And unfortunately, here's where I have some bad news, functional programming people, which probably won't surprise you:

this was because I was returning new objects instead of modifying, and the allocations were slow.

```
struct List {
    lookahead1: Node,
    lookahead2: Node,
    next_leaf_index: usize,
}

struct Node {
    weight: usize,
    leaf_counts: Vec<usize>,
}
```

63

I had this data structure, List, that held 2 lookahead Nodes.
Node has a Vec, so these both will be allocated on the heap.

```
current_list.lookahead2 = Node {
    weight: next_leaf.weight,
    leaf_counts: vec![
        current_list
            .lookahead1
            .leaf_counts
            .last()
            .unwrap() + 1
    ],
};
```

64

And I decided to be kind of functional, by making a new Node with a new Vec when the lookahead needed to be something different.

In a lot of spots that got run a lot.

```
current_list.lookahead2.weight =
    next_leaf.weight;


current_list.lookahead2.leaf_counts[0] =
    current_list
        .lookahead1
        .leaf_counts
        .last()
        .unwrap() + 1;
```

If instead I changed the Node instead of allocating a new one, performance got reasonable again.

Gotta watch the allocations.
There might be another way though…

```
fn do_something(list: &mut List)

let mut list = …
do_something(&mut list);


                vs


fn do_something(mut list: List) -> List

let list = …
let list = do_something(list);
```
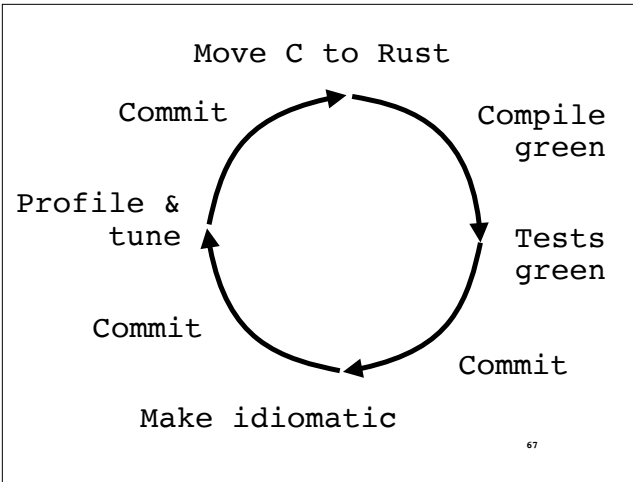
Instead of taking a mutable borrow, take ownership (which can give you mutability since you own it now) and return a "new" thing that's really just the thing you mutated

Annoying if you want to mutate multiple things at once?
Not super common except for builder pattern?

In Rust you have the choice and you can make the tradeoffs for your case.

```
            Move C to Rust

     Commit              Compile
                          green

Profile &
   tune                  Tests
                         green

     Commit

                 Commit

        Make idiomatic
                          67
```

So do what I say and not what I did— if your goal is to maintain or improve performance, your iterations should probably look more like this: after each function gets moved over, spend time checking performance and catching those regressions earlier instead of at the end like I did.

```
C-like Rust is
   slower than
idiomatic Rust?

                        68
```

My best guess for the rest of the slowness is that the Rust code is still pretty C-like, and isn't taking advantage of optimizations that idiomatic Rust could be. I'd like to do more work to see if this is the case.

Future work

- Remove all `unsafe`
- Use more iterators
- Stream input/output
- Refactor forever

69

Other things I'd like to do that I ran out of time to do before this talk are getting rid of all the uses of `unsafe` so that we're getting all the safety Rust could provide, using more iterators over elements instead of for loops to not have to worry about going past the end of arrays, Streaming is something the C library could use that I think Rust would make easier, so that's an enhancement actually, and of course I could move code around forever, as we all could on all of our code.

So… should you rewrite your C library in Rust?

¯_(ツ)_/¯

70

I dunno, maybe!

my point:
- Incremental rewrites from C to Rust are possible.
- Rust has FP concepts that can improve C code as you rewrite it
- Have reasons for rewriting and measure progress against the reasons.

71

What I hope you take away from my less-than-convincing results is that it is possible to rewrite C libraries in Rust in an incremental manner, Rust gives you FP concepts to use in that process, and that if you choose to undertake a rewrite, you should have a good reasons for doing so and you should measure against those goals along the way.

References (is.gd/c_rust)

- Repo of my code
- These slides
- FFI chapter in The Rust Programming Language book
- Rust FFI Omnibus
- Working Effectively with Legacy Code by Michael Feathers

72

Thank You

Carol (Nichols || Goulding)
@carols10cents