

Hello Rust

An Introduction + Ruby Extensions in Rust





Memory Unsafe

Memory Safe

Low-Level Pointers

C, C++, Objective C

Garbage Collected

Ruby, Java, Go

Memory Unsafe

Memory Safe

Low-Level Pointers

C, C++, Objective C

Garbage Collected

Ruby, Java, Go

Memory Unsafe

Memory Safe

Low-Level Pointers

C, C++, Objective C

Rust

Garbage Collected

Ruby, Java, Go

Memory Unsafe

Memory Safe

Low-Level Pointers

C, C++, Objective C

Rust

Garbage Collected

Objective C

Ruby, Java, Go

Memory Unsafe

Memory Safe

Low-Level Pointers

C, C++, Objective C

Rust

Garbage Collected

~~Objective C~~

Ruby, Java, Go

Why Low Level?

- Directly control memory usage
- Avoid uncontrollable GC pauses
- Embed into other GC'ed environments (like Ruby!)

Why Safe?

- **SEGVs** are **BAD!**

How Does it Work?

Ownership!

Ownership

```
struct Point { x: int, y: int }  
struct Line  { a: Point, b: Point }  
  
fn main() {  
    let a = Point{ x: 10, y: 10 };  
    let b = Point{ x: 20, y: 20 };  
}
```

owned by main()

Ownership

```
struct Point { x: int, y: int }  
struct Line  { a: Point, b: Point }  
  
fn main() {  
    let a = Point{ x: 10, y: 10 };  
    let b = Point{ x: 20, y: 20 };  
    let line = Line{ a: a, b: b };  
}
```

copied into line

owned by main()

Ownership

```
struct Point { x: int, y: int }  
struct Line  { a: Point, b: Point }  
  
fn main() {  
    let a = Point{ x: 10, y: 10 };  
    let b = Point{ x: 20, y: 20 };  
    let line = Line{ a: a, b: b };  
}
```

owned by `main()`

`line` is dropped with its contents; `a` and `b` are dropped

"Stack Allocation" is Cool

Stack Allocation

```
struct Point { x: int, y: int } ] 128 bits  
struct Line  { a: Point, b: Point } ] 256 bits
```

```
fn main() { ] allocate 512 bits for stack storage  
    let a = Point{ x: 10, y: 10 };  
    let b = Point{ x: 20, y: 20 };  
  
    let line = Line{ a: a, b: b }  
}
```


Ownership

```
struct Point { x: int, y: int }  
struct Line  { a: Point, b: Point }  
  
fn main() {  
    let a = Point{ x: 10, y: 10 };  
    let b = Point{ x: 20, y: 20 };  
  
    let line = Line{ a: a, b: b }  
}
```

owned by this scope

Ownership

```
struct Point { x: int, y: int }
struct Line  { a: Point, b: Point }

fn main() {
    let a = Point{ x: 10, y: 10 };
    let b = Point{ x: 20, y: 20 };
    let line = Line{ a: a, b: b }
}
```

moving?!

Ownership

```
struct Point { x: int, y: int }
struct Line  { a: ~Point, b: ~Point }

fn main() {
    let a = ~Point{ x: 10, y: 10 };
    let b = ~Point{ x: 20, y: 20 };

    let line = ~Line{ a: a, b: b } ] owned by main(), moveable
}
```

Ownership

```
struct Point { x: int, y: int }
struct Line  { a: ~Point, b: ~Point }

fn main() {
  let a = ~Point{ x: 10, y: 10 };
  let b = ~Point{ x: 20, y: 20 };

  let line = ~Line{ a: a, b: b } ] heap allocated
}
```

Computations

```
use std::num::{pow, sqrt};
```

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

```
struct Line { a: ~Point, b: ~Point }
```

```
fn length(line: ~Line) -> uint {
```

```
    let Line{ a, b } = line;
```

```
    let x = pow(b.x - a.x, 2);
```

```
    let y = pow(b.y - a.y, 2);
```

```
    sqrt(x + y)
```

```
} ] line is dropped
```

owned by length()

But Wait!

I can only do one computation with a struct?!

Computations

```
use std::num::{pow, sqrt};
```

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

```
struct Line { a: Point, b: Point }
```

```
fn length(line: &Line) -> uint {
```

```
    let x = pow(line.b.x - line.a.x, 2);
```

```
    let y = pow(line.b.y - line.a.y, 2);
```

```
    sqrt(x + y)
```

```
} ] line is not dropped
```

borrowed by `length()`

moving is disallowed
by the compiler

Stack Allocation is Back!

```
fn main() {  
    let line = Line{ a: Point{ x: 10, y: 10 }, b: Point{ x: 20, y: 20 } };  
    println!("length: {:u}", length(&mut line));  
}
```

```
fn length(line: &mut Line) -> uint {  
    let x = pow(line.b.x - line.a.x, 2);  
    let y = pow(line.b.y - line.a.y, 2);  
    sqrt(x + y)
```

```
} ] line is not dropped
```

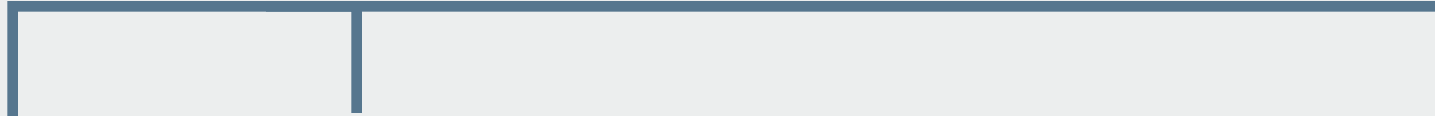
moving is disallowed by the
compiler, so passing a stack
value's memory address is safe

Object Orientation, Can Haz?

```
use std::num::{pow,sqrt};
```

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

```
struct Line { a: Point, b: Point }
```

```
impl Line {  self is almost always borrowed  
    fn length(&self) -> uint {  
        let x = pow(self.b.x - self.a.x, 2);  
        let y = pow(self.b.y - self.a.y, 2);  
        sqrt(x + y)  
    }  
}
```

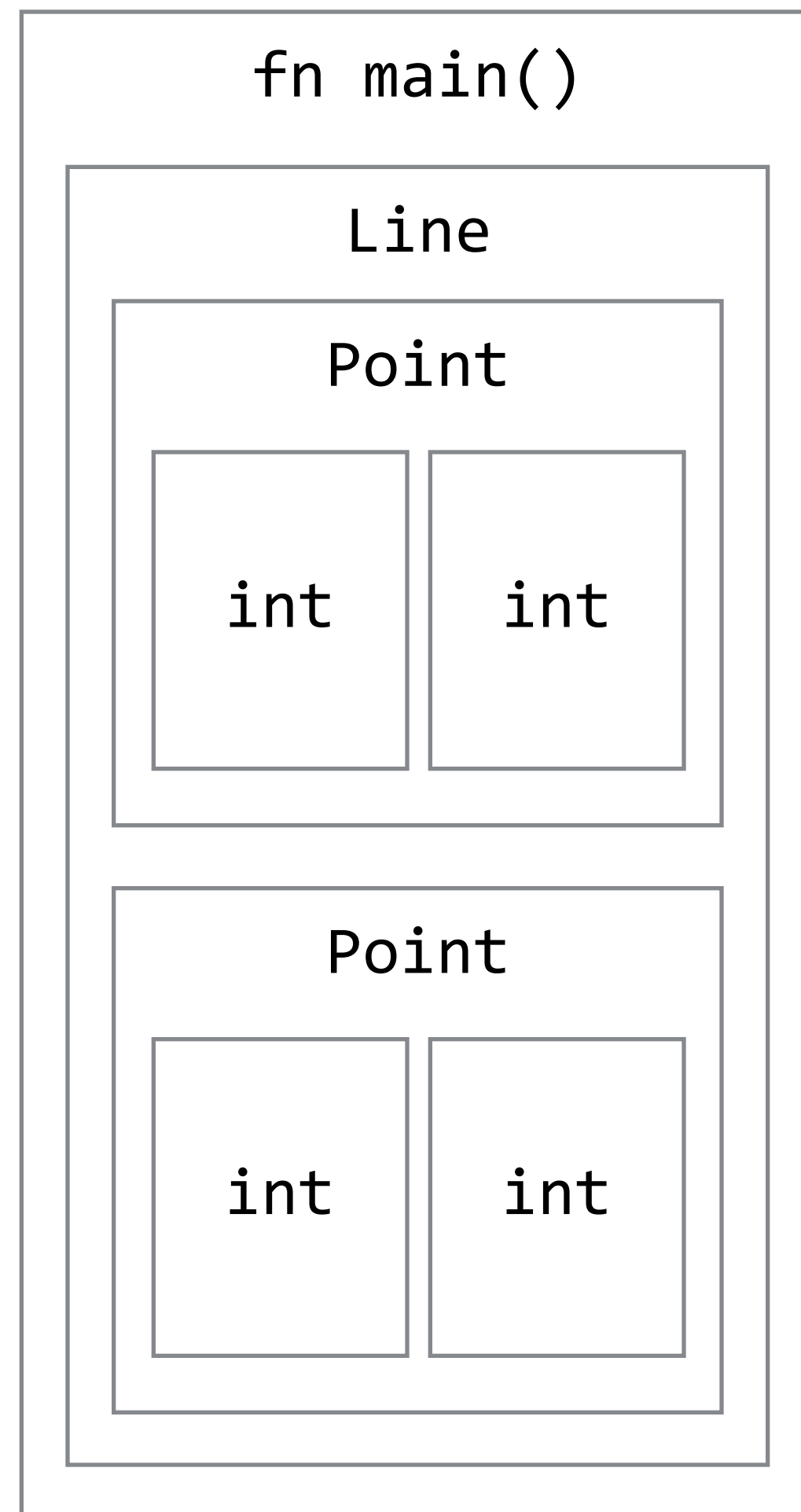
Net Result

- You never have to malloc or free
- You never have to retain or release
- Rust will deallocate a value when the current owner is done with it
- The compiler will guarantee that borrowed values are not stolen

Normal Idioms

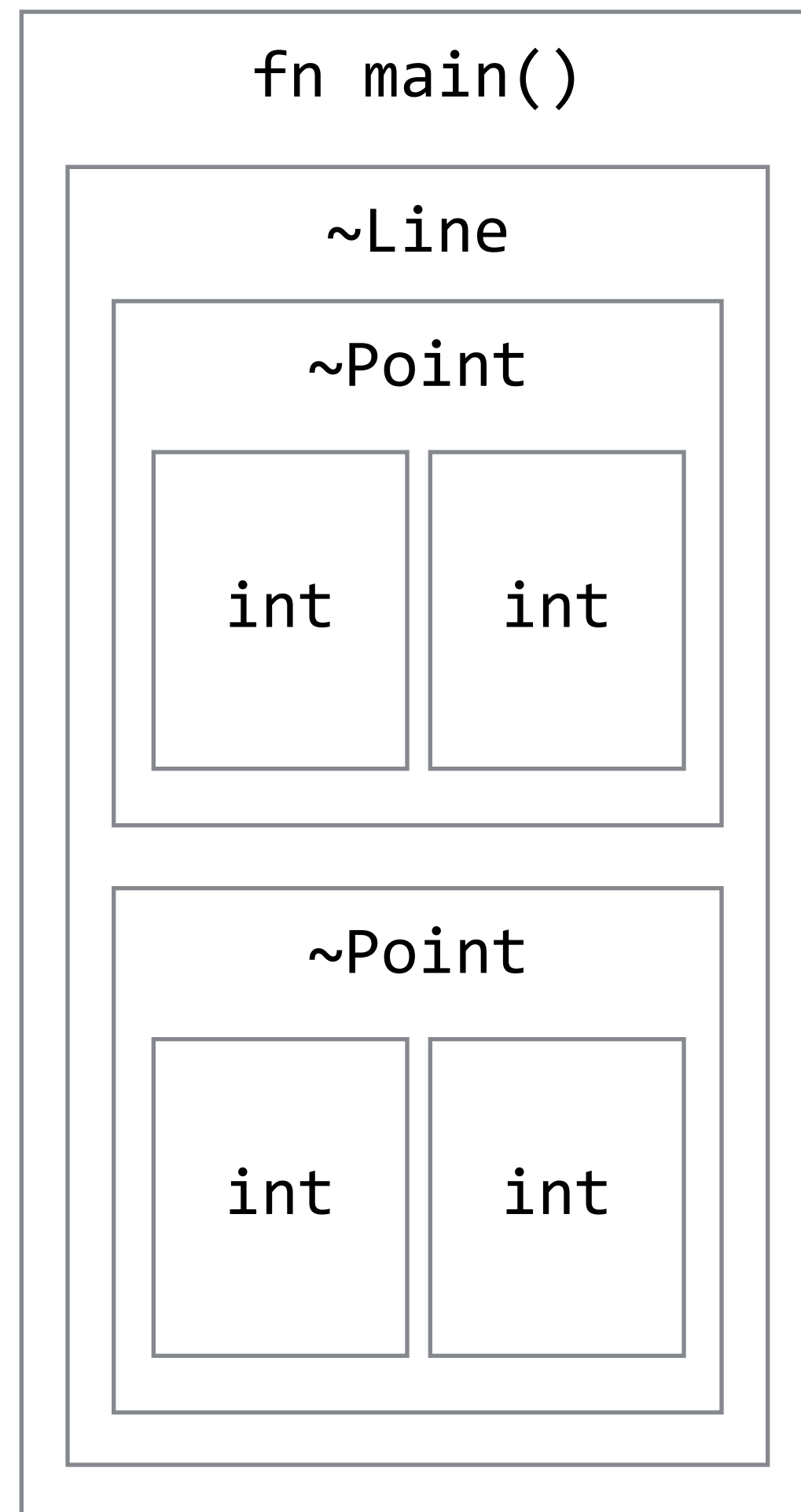
- Prefer stack allocation to heap allocation if you can get away with it
- Usually borrow values (`&T` visually blends away, `~T` is an outlier)
- Mutability is explicit (`&T` vs. `&mut T`)
- OO methods virtually always take `&self` or `&mut self`

Ownership and Leaks

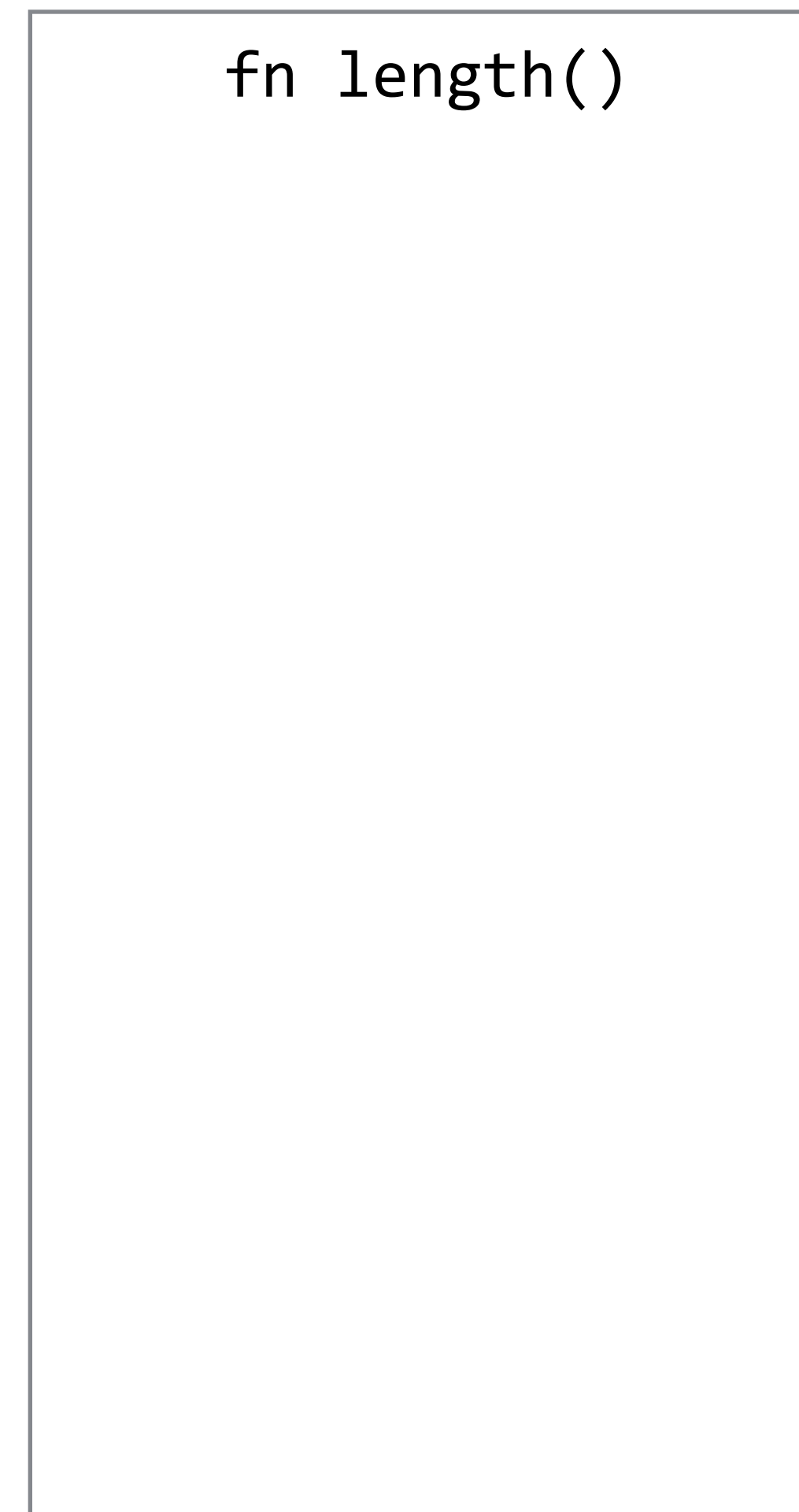
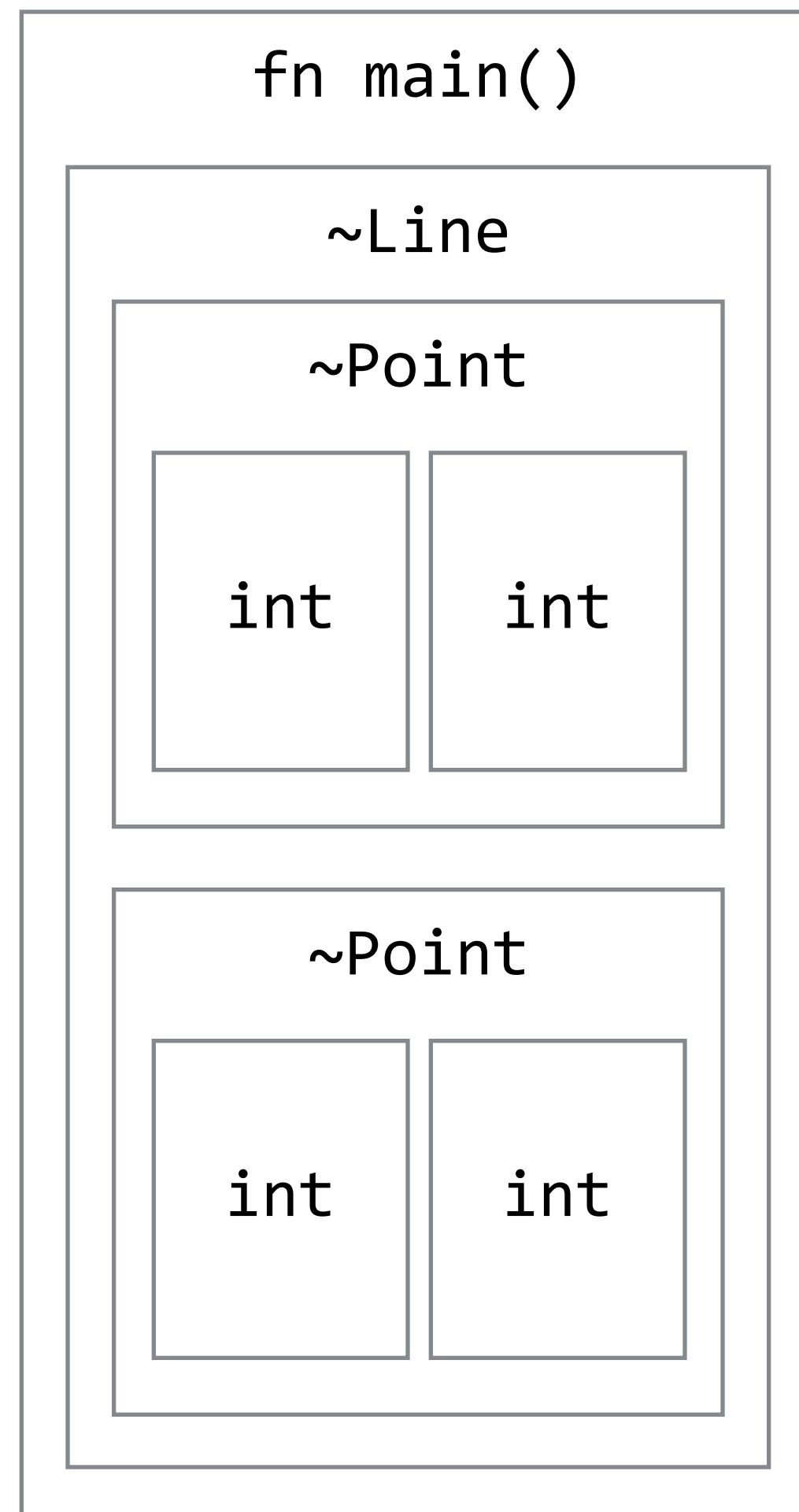


Ownership and Leaks

Ownership and Leaks



Ownership and Leaks



Ownership and Leaks

fn main()

fn length()

~Line

~Point

int

int

~Point

int

int

Ownership and Leaks

```
fn main()
```

Ownership and Leaks

Task Local Storage

Because it's "permanent", moving ~T
into it can cause leaks

Long-Lived Loops

Long-lived loops may contain
ever-growing Arrays that "leak"

It's never "I forgot to free"

Traits

Area

```
struct Square { width: f32 }
struct Circle { radius: f32 }

trait Shape {
    fn area(&self) -> f32;
}

impl Shape for Square {
    fn area(&self) -> f32 { self.width * self.width }
}

impl Shape for Circle {
    fn area(&self) -> f32 { pow(3.14 * self.radius, 2) }
}
```

Usage

```
use mylib::{Shape, Circle, Square};

fn area<T: Shape>(shape: &T) -> f32 {
    shape.area()
}

fn main() {
    let circle = Circle{ radius: 5 };
    let square = Square{ width: 10 };
    println!("Inscribed circle: {}", area(&circle) - area(&square));
}
```


Usage

```
use mylib::{Shape, Circle, Square};
```

```
fn area<T: Shape>(shape: &T) -> f32 {  
    shape.area()  
}
```

<T: Shape>
any type that
implements Shape

```
fn main() {  
    let circle = Circle{ radius: 5 };  
    let square = Square{ width: 10 };  
    println!("incrimbed circle: {}", area(&circle) - area(&square));  
}
```

Generates, Under the Hood

```
use mylib::{Shape,Circle,Square};

fn circle_area(shape: &Circle) -> f32 {
    shape.area()
}

fn square_area(shape: &Square) -> f32 {
    shape.area()
}

fn main() {
    let circle = Circle{ radius: 5 };
    let square = Square{ width: 10 };
    println!("incribed circle: {:?}", circle_area(&circle) - square_area(&square));
}
```

Operator Overloading

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

impl Add<Point, Point> for Point {
    fn add(&self, other: &Point) -> Point {
        Point{ x: self.x + other.x, y: self.y + other.y }
    }
}

fn log_add<T: Add>(first: &T, second: &T) {
    println!("{:?}", first + second);
}
```

<http://bit.ly/rust-go-http>

http://bit.ly/rust-go-http

```
use http::{Request, Response, HttpErr};
use http::status::Found;

fn load_page(title: &str) -> Page {
    let body = try!(File::read(str));
    Page::new(title, body)
}

fn view(res: &mut Response, req: &Request) {
    match load_page(req.params["id"]) {
        Err(_) =>
            res.redirect("/edit/" + title, Found),
        Ok(page) =>
            render_template(res, "view", page)
    }
}
```

```
fn edit(res: &mut Response, req: &Request) {
    let title = req.params["id"];

    let page = match load_page(title) {
        Err(_) => Page{ title: title },
        Ok(page) => page
    }

    render_template(res, "edit", page);
}

fn save(res: &mut Response, req: &Request) {
    let title = req.params["id"];
    let body = try!(req.params["body"]);

    let page = try!(Page::new(title,
        body.as_bytes()).save());

    res.redirect("/view/" + title, Found);
}
```

http://bit.ly/rust-go-http

```
import (
    "html/template",
    "io/ioutil",
    "net/http"
)

func loadPage(title string) *Page {
    filename := title + ".txt"
    body, _ := ioutil.ReadFile(filename)
    return &Page{Title: title, Body: body}
}

func getTitle(w http.ResponseWriter, r *http.Request) (string, error) {
    m := validPath.FindStringSubmatch(r.URL.Path)
    if m == nil {
        http.NotFound(w, r)
        return "", errors.New("Invalid Page Title")
    }
    return m[2], nil // The title is the second subexpression.
}

func viewHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        http.Redirect(w, r, "/edit/"+title, http.StatusFound)
        return
    }
    renderTemplate(w, "view", p)
}

func editHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    p, err := loadPage(title)
    if err != nil {
        p = &Page{Title: title}
    }
    renderTemplate(w, "edit", p)
}

func saveHandler(w http.ResponseWriter, r *http.Request) {
    title, err := getTitle(w, r)
    if err != nil {
        return
    }
    body := r.FormValue("body")
    p := &Page{Title: title, Body: []byte(body)}
    err = p.save()
    if err != nil {
        http.Error(w, err.Error(), http.StatusInternalServerError)
        return
    }
    http.Redirect(w, r, "/view/"+title, http.StatusFound)
}
```

Ruby C Extension

The Basics

```
struct Trace {  
    name: ~str  
}  
  
#[no_mangle]  
pub extern "C" fn skylight_get_name(trace: &Trace) -> ~str {  
    trace.name.clone()  
}
```


The Basics

```
#[no_mangle]
pub extern "C" fn skylight_get_name(trace: &Trace) -> ~str {
    trace.name.clone()
}

VALUE trace_get_name(VALUE self) {
    RustTrace trace;
    Data_Get_Struct(self, RustTrace, trace);
    return RUST2STR(skylight_get_name(trace));
}
```

Memory

```
#[no_mangle]
pub extern "C" fn skylight_free_str(_: ~str) {}

#define RUST2STR(string) ({
    RustString s = (string);
    VALUE ret = rb_str_new((char *)s->data, s->fill);
    skylight_free_str(s);
    ret;
})
```

Many Details

- Catching failures at the boundary
- Freeing objects idiomatically on the C side
- Invalidating C references involved in a failure
- ...

Skylight's Library

```
ffi_fn!(skylight_trace_get_name(trace: &'a Trace) -> &'a str {  
    trace.get_name()  
})
```

```
static VALUE trace_get_name(VALUE self) {  
    My_Struct(trace, RustTrace, freedTrace);
```

```
    RustSlice string;  
    return CHECK_FFI(skylight_trace_get_name(trace, &string),  
        "Could not get the name from a Trace in native code");  
}
```

ffi_fn!

- A Rust macro (yay macros!)
- Synthesizes a Rust task and catches failures
- Returns a bool to indicate success or failure
- Uses an out variable for the return value

On the C-Side

- `RUST2STR` copies Rust strings into Ruby strings
- `typedef void * RustTrace`
- `CHECK_FFI` turns a false return into a Ruby exception
- `CHECK_TYPE` and `CHECK_NUMERIC` protect the boundary
- `Transfer_My_Struct` nulls out an object's struct if passed as `~T`
- Learn `FIX2INT`, `INT2FIX`, `ULL2NUM`, `NUM2ULL`, etc.

rust.rb COMING SOON!

Questions?

@wycats

plus.yehudakatz.com

yehudakatz.com