Rust: A Friendly Introduction

Tim Chevalier
Mozilla Research
June 19, 2013

`http://rust-lang.org/`
`https://github.com/mozilla/rust/`

This is a revised version, June 25, 2003, that corrects a few typos and adds additional notes where needed.

A language design prelude

- We designed Rust to bridge the performance gap between safe and unsafe languages.
- Design choices that seem complicated or surprising on first glance are mostly choices that fell out of that requirement.
- Rust's compiler and all language tools are open-source (MIT/Apache dual license).

this is mostly a talk about how to write code, but I couldn't resist putting in some language design content, because to explain Rust it's important to understand the problems we faced designing for a domain where performance concerns mean you can't just do things like boxing/gc'ing everything

Systems Programming

- Efficient code operating in resource-constrained environments with direct control over hardware
- C and C++ are dominant; systems programmers care about very small performance margins
- For most applications we don't care about the last 10-15% of performance, but for systems we do

ask for show of hands, how many people are C hackers, how many primarily in Java-ish languages
"what is systems programming?" (point 1)
"there's a good reason why C is dominant";
I argue that the look & feel of the language necessarily follow from designing so as to bring safety to systems programming

Well, what's wrong with C, then?

dangling pointers

buffer overflows

array bounds errors

format string errors

null pointer dereferences

double frees

memory leaks

"But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years." - Tony Hoare

"Well-typed programs don't go wrong"

Milner, "A theory of type polymorphism in programming", 1978

- What would it mean to go wrong?
- Rust's type system is designed to be sound, which means:
  - We can predict program behavior independently of language implementation

dangling pointers

buffer overflows

array bounds errors

format string errors

null pointer dereferences

double frees

memory leaks

(This gives you *more* confidence that Rust programs will be reliable, not absolute confidence. Compilers and runtime systems may have bugs. Unsafe code voids the warranty. Offer not valid in Nebraska.)

Rust: like C, but...

- One reason why C persists is that there's a simple relationship between the meaning of your code and the behavior of the underlying machine
- This correspondence makes it relatively easy to write efficient code in C
- We want Rust to preserve this relationship

Manual code review look at source code/assembly code (machine code?) side-by-side. Hard to imagine doing that in Java/Haskell/ML..
Rust **keeps the same model** as C, matching C++ idioms where they matter (esp. WRT memory allocation)

with *memory safety*

- So what is memory safety?
- One definition: programs dereference only previously allocated pointers that have not been freed

...without runtime cost

- In safe languages like Java, Python, and Haskell, abstractions come with a runtime cost:
  - boxing everything
  - garbage-collecting everything
  - dynamic dispatch
- Rust is about *zero-cost abstractions*
- there's a cognitive cost: in Rust we have to think *more* about data representation and memory allocation. But the compiler checks our assumptions.

say "soundness" but not in slide"
"to add: resource constraints, low overhead, zero-cost abstractions (cite ROC)"
(n.b. In some languages, e.g. ML, you can lose "boxing everything" if you also give up separate compilation.)

Roadmap: writing fast and safe code in Rust

- Fun With Types
- Types Can Have Traits
- Pointers and Memory
- Bigger Examples
- Testing, benchmarking...
- Questions?

The one thing I hope you remember:

THE COMPILER CAN CHECK THAT YOUR CODE USES SYSTEMS PROGRAMMING PATTERNS SAFELY

I hope you'll leave this talk wanting to learn more about Rust on your own. My goal is to break down the intimidation factor, not so much to teach you Rust in an an hour and a half. Hopefully the talk will give you a sense of why you would want to.

note to self: try **NOT** to assume C++ knowledge as a baseline

Disclaimer

- Some code examples have been simplified in trivial ways for clarity, and won't necessarily typecheck or compile
- When I post slides online I'll document changes I've made for presentation purposes

Changes needed to get code to compile will be in pink bold text in the notes

Mutability

- Local variables in Rust are *immutable* by default

```
let x = 5;
let mut y = 6;
y = x;            // OK
x = x + 1;        // Type error
```


mutability-by-accident is a huge source of bugs

Statements and expressions

- Two kinds of statements in Rust:
  - `let var = expr;`
  - `expr;` — Equivalent to `let _ignore = expr;`
- Everything is an expression; everything has a value
- Things that only have side effects have the type `( )` (“unit”)

no whitespace-sensitivity

Functions

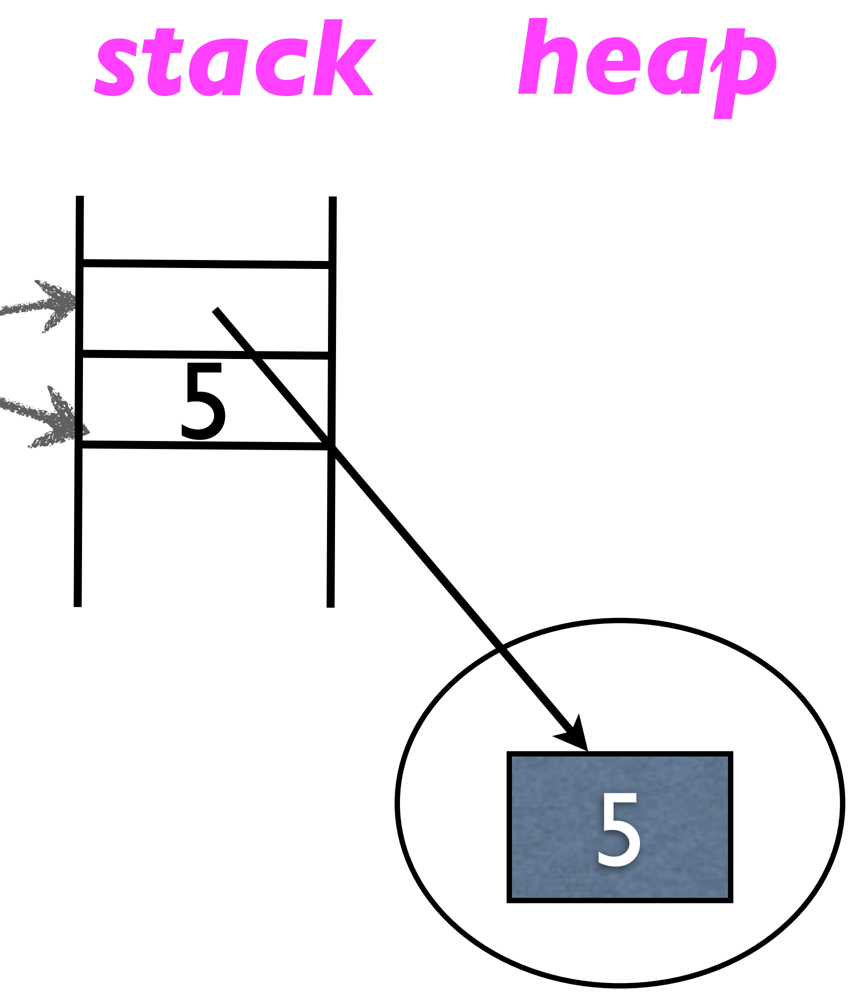
```
fn f(x: int) -> int {
    x * x
}
```

No semicolon

```
fn f(x: int) -> int {
    return(x * x);
}
```

Pointers

```
let x: int = f();
let y: @int = @x;
assert!(*y == 5);
/* Doesn't typecheck */
// assert!(y == 5);
```



- Rust has type inference, so you can usually leave off the types. I'm leaving them for pedagogical purposes.
- Rust @-pointers can't be null

For most of the talk to make sense, you have to understand the difference between pointer and pointed-to

Pointers and mutability

```
let mut x: int = 5;
increment(&mut x);
assert!(x == 6);
// ...
fn increment(r: &mut int) {
    *r = *r + 1;
}
```

Enumerations

Rust

```
enum Color
{
    Red,
    Green,
    Blue
}
```

C

```
typedef enum {
    Red,
    Green,
    Blue
} color;
```


Relate to “fast and trustworthy”. Enum types let us write code that we know is exhaustive.
In C: fast because enums are a compile-time thing, they just turn into small integers at runtime
C has two major problems here:
1. Missing cases
2. Being able to access fields of variants without checking tags

Matching on enumerations

Rust

```
fn f(c: Color) {
    match c {
        Red => // ...
        Green => // ...
        Blue => // ...
    }
}
```

C

```
void f(color c) {
    switch (c) {
        case Red: { /* ... */
            break;
        }
        case Green: { /* ... */
            break;
        }
        case Blue: { /* ... */
            break;
        }
    }
}
```

(omitted return type means ())

show that C lets you add nonsense cases & (more importantly) leave out cases
mention: in Rust, no fall-through; must include a default case (_ => ())
point out again that match is an expression

Nonsense cases

Rust

```
fn f(c: Color) {
    match c {
        Red => // ...
        Green => // ...
        17 => // ...
    }
}
```

Type error

C

```
void f(color c) {
    switch (c) {
        case Red: { /* ... */
            break;
        }
        case Green: { /* ... */
            break;
        }
        case Blue: { /* ... */
            break;
        }
        case 17: { /* ... */
            break;
        }
    }
}
```


C accepts this because enums are “just” ints
But it probably indicates a programmer error

Non-exhaustive matches

Rust

```
fn f(c: Color) {
    match c {
        Red => // ...
        Green => // ...
```

Exhaustiveness error

```
    }
}
```

C

```
void f(color c) {
    switch (c) {
        case Red: { /* ... */
            break;
        }
        case Green: { /* ... */
            break;
        }
    }
}
```


the C version is perfectly OK!
* what Rust gives you: checking that you have one case per variant, no missing cases and no nonsense cases.
This is hugely important in a large code base when you change a data structure. Knowing the compiler will flag these errors gives you great peace of mind.
Type system **tells** you that c is one of three possible values, instead of any int-sized value. Constraining the set of things a given variable could be is very useful, and gives you the ability to know you're handling all cases.

Enums can have fields

```
enum IntOption {
    None,
    Some(int)
}
```

```
typedef struct IntOption {
    bool is_some;
    union {
        int val;
        void nothing;
    }
}
```

Rust ↖ **C** ↗

Option => safe replacement for possibly-null pointers
Showing a specific version here, mention that in general this works on any type
this is nothing new -- Haskell/ML/etc. have had it for decades -- what's newish (but not unique) is having it in a systems language
example on R is a bit contrived since it's just making C null pointers explicit. Bear with me!

Checking for Null

C

```
IntOption opt = random_value();

if (opt.is_some) {
    printf("%d\n", opt.val);
}
```


What if you left off the "if"? (Would dereference a null pointer.)
Rust has a way to protect you from making that mistake.

Destructuring in Rust

Only way to access the *i* field!

```
let opt: IntOption = random_value();

match opt {
    None => (), // do nothing
    Some(i) => io::println(fmt!("It's %d!", i))
}
```


There's literally no way to construct code that extracts out the int field without checking the tag
and again, Rust compiler checks that we covered every case and don't have overlapping cases
summing up: enums create data, pattern matching deconstructs them, and pattern matches get checked to make sure we're using data in a way that's consistent with the invariants imposed by its type

Pattern-matching and vectors

Binds tail to [2, 3] in this case

```
let x = [1, 2, 3];
match x {
    [1, ..tail] => // ...
    [_, ..tail] => // ...
    [] => // ...
}
```


one slide to both introduce vectors, and talk about slices?
vectors: constant-time-random-access, dynamically sized sequences of elements of the same type

Structs

- Similar to C `structs`
    - fields are laid out contiguously in memory, in the order they're declared
- In C, allocating memory for a struct and initializing the fields are separate
- Rust guarantees that struct fields that can be named are initialized

emphasize: no uninitialized fields
Buzzwords: records; nominal types

Struct example

```
struct Element {
    parent: Node,
    tag_name: str,
    attrs: [Attr],
}

// ...
let e: Element = mk_paragraph();
assert!(e.tag_name == “p”);
```

from Servo src/servo/dom/element.rs

Change str to ~str and [Attr] to ~[Attr]. Change “p” to ~”p”

Closures

```
fn apply(i: int, f: fn(int) -> int) -> int {
    f(i)
}

// ...

assert!(apply(4, |x| { x * x }) == 16);
```

"A function of one argument x that returns the square of x"

Change fn(int) to &fn(int)

(lambdas/anonymous/higher order functions) => This is a feature that enables better code reuse.

Also flexible control structures. Rust implements it efficiently.

kind of a boring use of closures, yes. Next slide shows a more interesting one.

Loops

```
for range(0, 10) |i| {
    println(fmt!("%u is an integer!", i));
}
```

A standard library function that applies a closure to every number between (in this case) 0 and 10

Add the line: `use std::uint::range;` **at the top of the file**

Rust's more-flexible loop constructs encourage more modular code, fewer tedious loop-counting errors

At the same time, all of this is implemented in the language itself, as libraries. You can write your own looping constructs. The generated code is just as fast as C code that uses for loops.

(compiler steps)

Loops

```
for range(0, 10) |i| {
    println(fmt!("%u is an integer!", i));
}
```

expand

```
range(0, 10, |i| {
    println(fmt!("%u is an integer!", i));
})
```

inline

```
let mut j = 0;
while j < 10 {
  println(fmt!("%u is an integer!", j));
  j += 1;
}
```


this is interesting because the code is really very different... top is a (sugared) call to a higher-order function,
bottom is a direct loop

and there's no magic involved -- just syntactic sugar and simple inlining

Methods

```
struct Pair { first: int, second: int }

impl Pair {
    fn product(self) -> int {
        self.first * self.second
    }
}

fn doubled_product(p: Pair) -> int {
    2 * p.product()
}
```

Method call

Generics

- Functions can be abstracted over types, not just over values
- Data types can also have type parameters
- Generics vs. polymorphism: same concept, different terms (I'll use "generics")

Generic types: example

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


Yes, it is meant to look like templates

Generic functions: example

```
fn safe_get<T>(opt: Option<T>, default: T) -> T {
    match opt {
        Some(contents) => contents,
        None           => default
    }
}
```


"like Java generics" -- types get specified at *compile* time, type parameters have no runtime meaning
difference between this and templates: it's possible to typecheck each function separately (which means better error messages),
regardless of how it's used. the step of expanding stuff out is separate. separate compilation
in cmr's words: "Cross-library generics without header files!"

Generic functions: implementation

You write:

```
let x = safe_get(Some(16), 2);
let y = safe_get(Some(true), false);
let z = safe_get(Some('c'), 'a');
```

Compiler generates:

```
fn safe_get_int(opt: Option_int, default: int) -> int

fn safe_get_bool(opt: Option_bool, default: bool) -> bool

fn safe_get_char(opt: Option_char, default: char) -> char

enum Option_int {
    Some_int(int),
    None_int
}
// same for bool and char
```


bold orange stuff is all compiler-generated
compare to C++ templates or Java generics
compiler “expands a template”/”makes a copy” with type variables set to specific types
[anticipate question “how is this better than C++ templates?” -- one answer is traits (limiting what types something can expand to)]
Separate typechecking/compilation

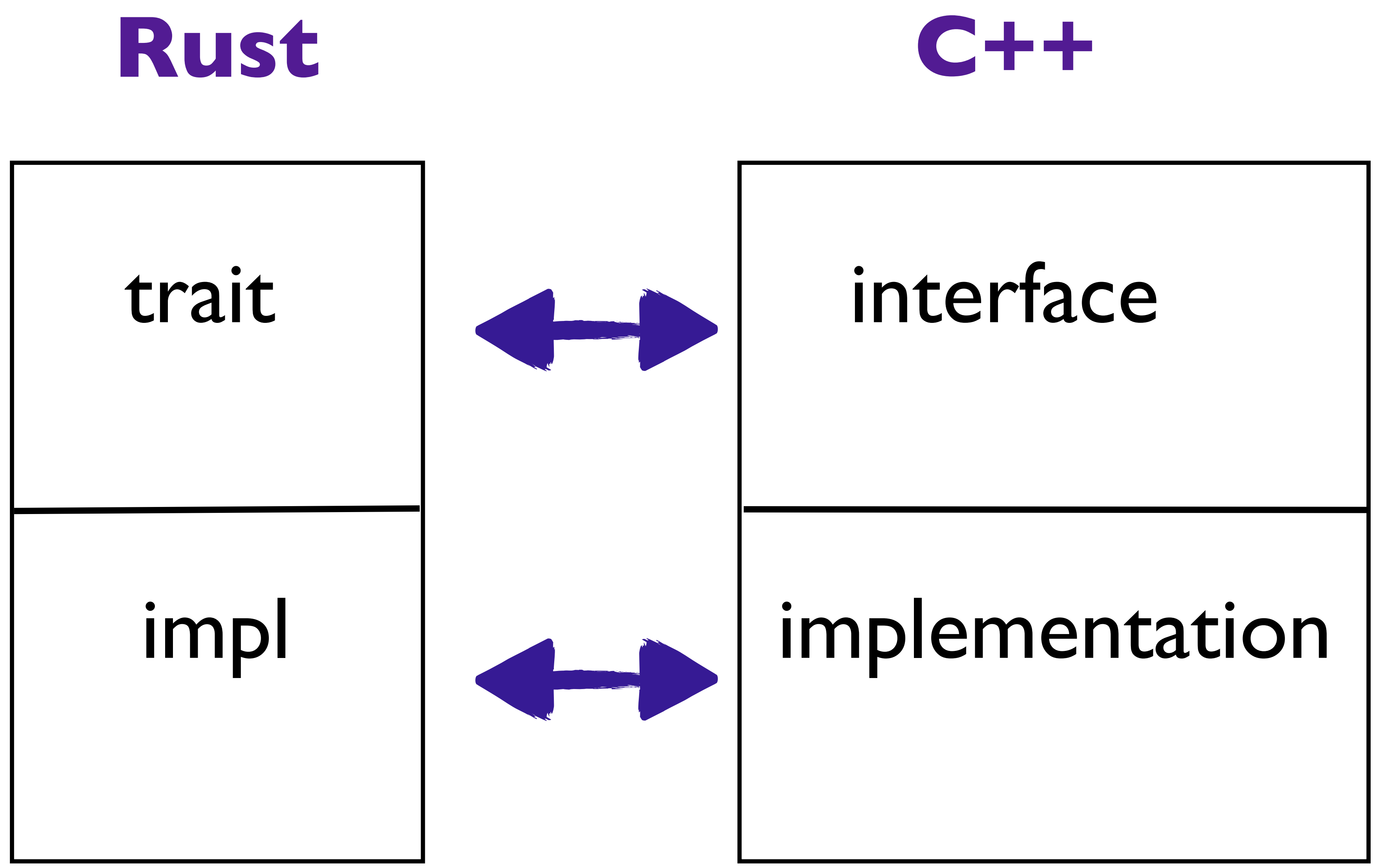
Interfaces

```
fn all_equal_to<T>(ys: [T], x: T) -> bool {
    for ys.each |y| {
        if y != x {
            return false;
        }
    }
    true
}
```

Doesn't typecheck!

The problem is that there's no general way to compare two values of an arbitrary type T for equality
We need a way to be able to say "does T implement the Eq interface?", and to be able to assume -- in a generic function T -- that the function only makes sense on types T that support the Eq interface

Types can have traits



Trait example

```
trait Mem {
    fn loadb(&mut self, addr: u16) -> u8;
    fn storeb(&mut self, addr: u16, val: u8);
}
```

sprocketnes/mem.rs

A trait defines an interface (collection of type signatures).
[Recall that] Trait functions are called methods. Methods differ from functions because they have a self parameter that's special.
You can think of self -- here -- as having type &mut T: Mem.
This trait defines the interface for types that represent a collection of memory.
In this case, to count as a Mem, a type has to support two operations -- load and store, each of which take or return a byte
(this is a 16-bit machine). In sprocket, several different types implement Mem: PPU, RAM, VRAM, ...

Trait bounds

T is bounded

```
fn store_two_bytes<T: Mem>(addr1: u16,
                           addr2: u16,
                           byte1: u8,
                           byte2: u8,
                           a_mem: &mut T) {
    a_mem.storeb(addr1, byte1);
    a_mem.storeb(addr2, byte2);
}
```


made-up example

Implementation example

```
//
// The NES' paltry 2KB of RAM
//

struct Ram { ram: [u8, ..0x800] }

impl Mem for Ram {
    fn loadb(&mut self, addr: u16) -> u8
        { self.ram[addr & 0x7ff] }
    fn storeb(&mut self, addr: u16, val: u8)
        { self.ram[addr & 0x7ff] = val }
}
```

sprocketnes/mem.rs

the impl item is a concrete implementation of the trait Mem for the type Ram
the concrete type Ram here is a fixed-length vector of bytes, but in theory it could be any type on which you can implement these operations

Static vs. dynamic dispatch

- The compiler compiles all the code we've been talking about (so far) with *static dispatch*: the function being called is known at compile time
- Static dispatch is more efficient, because call instructions always go to a known address
- You can trade performance for flexibility and use dynamic dispatch
- n.b. In languages like Java, Python, Ruby (...) dynamic dispatch is all there is. In Rust you have a choice.

Dynamic dispatch

a list of objects that may have different types, so long as all types are Drawable

```
trait Drawable { fn draw(&self); }

fn draw_all(shapes: [@Drawable]) {
    for shapes.each |shape| { shape.draw(); }
}
```

from the Rust tutorial, `http://static.rust-lang.org/doc/tutorial.html`

40

Change `[@Drawable] to ~[@Drawable]`

* another for loop...
* we need the @ sigil to show where a Drawable object is stored, and to make it clear it's a pointer
* by itself, Drawable is not a type. But @Drawable / ~Drawable / ~T are types

Static vs. dynamic

```
fn draw<T: Drawable>(shapes: &[T]) {
    for shapes.each |shape|{
        shape.draw();
    }
}
```

compiler

```
fn draw_circles(shapes: &[Circle]) { ...
fn draw_rectangles(shapes: &[Rectangle])
{ ...
```

```
fn draw(shapes: &[@Drawable]) {
    for shapes.each |shape|
    {
        shape.draw();
    }
}
```

```
fn draw(shapes: ...) {
  for shapes.each |shape| {
    let vtable = shape.vtable;
    call vtable.draw(shape);
}
}
```

(pseudocode)

On the right, the generated code is doing work *at runtime* to look up the draw method for each object.
On the left, the compiler generates a copy at *compile time* of the draw function for each shape type that draw gets used with.
as with templates, “the compiler generates a copy of every parameterized fn and ty”)

Traits: summing up

- Traits provide us with code reuse for no runtime cost, when using static dispatch
- Can use dynamic dispatch for greater flexibility, when you're willing to pay the cost
- In Rust, you can use either style depending on context; the language doesn't impose a preference on you
- (Traits are inspired by Haskell type classes, but don't worry if you don't know about those)

Unifying static/dynamic dispatch is a new thing
emphasize code reuse = safety, because less duplication = less bugs

Changing gears

(questions so far?)

Memory in Rust

- Existing languages tend to either not support explicit pointers (e.g. Java, Haskell) or support them without detecting obvious errors (e.g. C/C++). There's another way!
- Rust's performance goals don't admit garbage-collecting everything as a solution
- At the same time, want to avoid the hazards of manual memory management

memory in Rust; using pointers safely.
Brian said "pointerless vs. pointerful" comparison is good
point 2 = fast, point 3 = safe

Pointers and memory

45

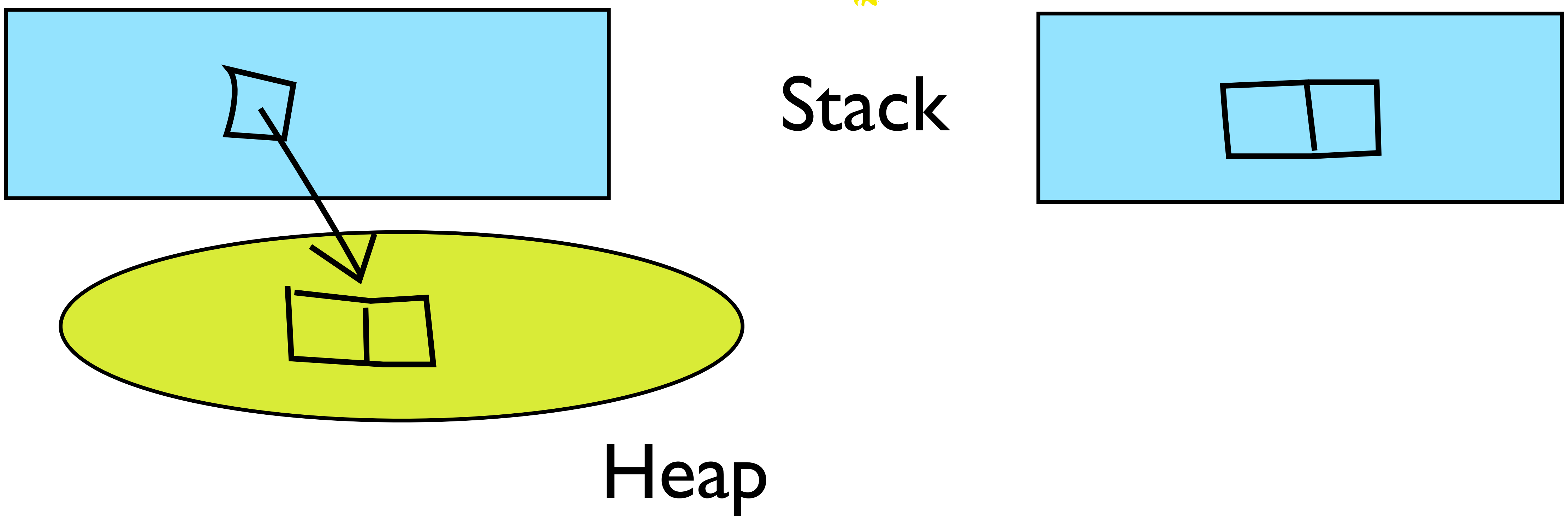
Crucial point is that "pointerless" languages (Java, Haskell, ML, dynamic langs...) have to box everything; they lack the ability to talk about non-pointer-sized things in the language

(1, 2) in Haskell always [*] means a pointer to a heap-allocated record with two data fields
(the compiler can optimize this sometimes, but the language gives no guarantees)
makes it simpler to compile polymorphism, b/c the compiler knows the size of everything. But that's not the only way!
** can't rely on this optimization if *predictable* (consistent) performance is important

Boxed vs. unboxed

```
fn f(p: @(int, int)) { }
```

```
fn f(p: (int, int)) { }
```



in some languages, you wouldn’t be able to express this distinction -- compound data would always live on the heap.
In Rust, you can choose whether it lives on the stack or in the heap.
Difference is that stack-allocated data has a natural lifetime corresponding to lexical scope -- no need for GC/etc.
(Same caveat as in slide 8: (n.b. In some languages, e.g. ML, you can lose “boxing everything” if you also give up separate compilation.))

"Rust has three kinds of pointers?"

- Actually, Rust has four kinds of pointers
- But the secret is, C++ does too
  - In C++, `*T` can mean many different things; the particular meaning in a given context lives in the programmer's head
  - In Rust, the pattern is explicit in the syntax; therefore checkable by the compiler

- The difference is, Rust helps you remember which kind you're using at any given moment

Different Patterns

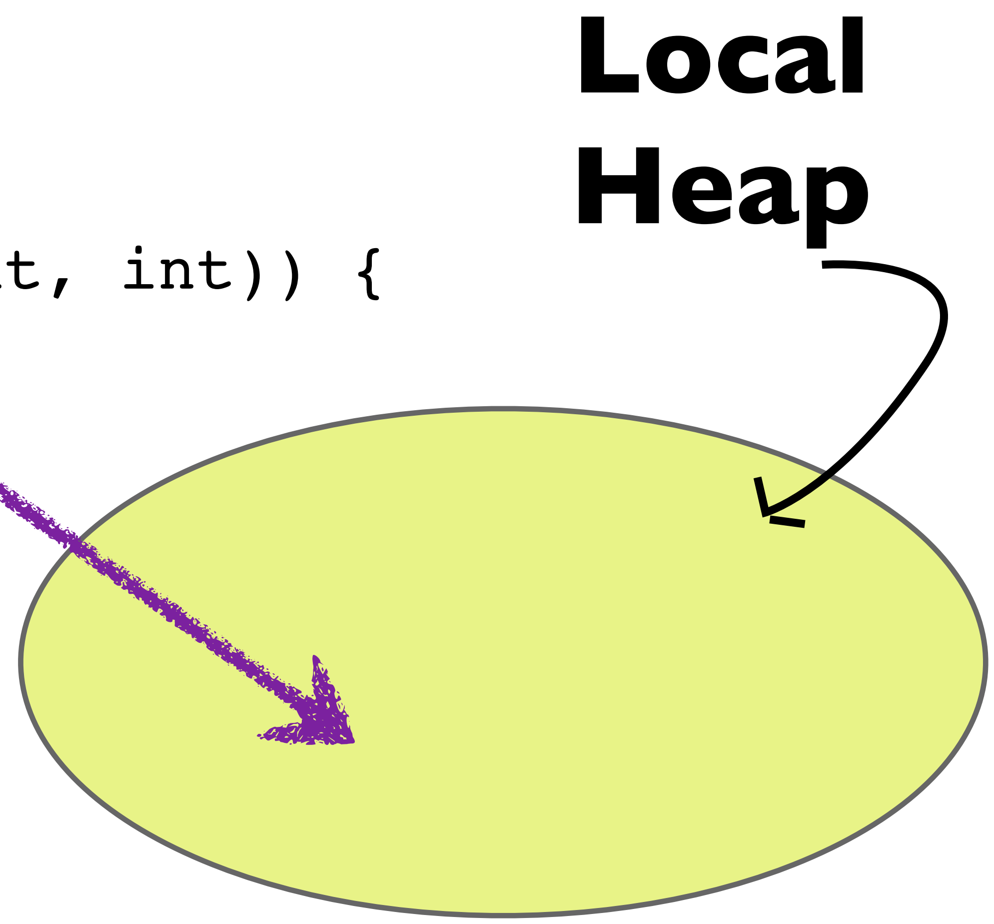
- Managed pointer to T
- Owned pointer to T
- Borrowed pointer to T
- Unsafe pointer to T

Rust	C++
@T	*T
~T	*T
&T	*T
*T	*T

The Rust compiler checks that code uses each pointer type consistently with its meaning.

Graydon points out "also, compiler can prove it's safe"
and yes, C++ has references/smart pointers/etc., but the treatment of these in Rust is better-integrated,
more uniform, more easily checkable...
the C++ compiler *can't* check it since it doesn't know what type you meant!

Managed Pointers



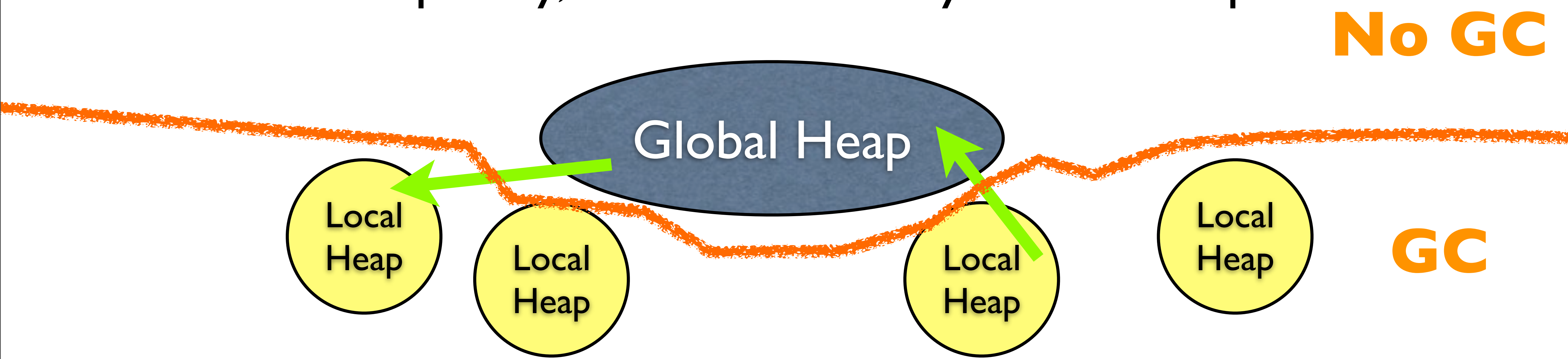

```
fn remember(s: &mut Set, foo: @(int, int)) {
// ... add to set ...
}
```

- `foo` is a pointer into the local heap
- The local heap is called "managed" because...
  - the caller need not manually free pointers into it; the compiler/runtime frees it when it's no longer needed, either using garbage collection or by generating automatic reference-counting code

- (which one it uses is an implementation detail)

Owned pointers

- Conceptually, there are actually several heaps:



- An allocation in the global heap has a single *owner* (a block scope or parent data structure that's responsible for freeing that allocation).

Different tasks, different heaps
Could mention as an aside that the managed heap is also per-task and the exchange heap can be used to move data between tasks
pointers can point from one heap to the other

Preventing copies

```
fn h(b: ~[int]) { ... }
fn g(a: ~[int]) { ... }

fn f(n: uint) {
    let v: ~[int] = vec::from_elem(n, 2);
    h(v);
    g(v);
}
```

This says "pass the argument by value"

Typechecker rejects this call

Before I talk about the last kind of pointer (borrowed) I want to talk about move semantics

the location of v gets zeroed out when we call h. So the call to g wouldn't be sound -- g would
get a dangling pointer. Rust's typechecker prevents that. In addition, we don't interpret the call as a copy
because v is a big value. Calling h "transfers ownership" of v to h

Borrowed pointers

```
fn f(a_big_number: uint) -> uint {
    let mut big_vector = ~[];
    for range(0, a_big_number) |n| {
        big_vector += [n];
    }
    sum(big_vector)
}

fn sum(v: &[int]) -> int { ... }
```

The type tells us that sum "borrows" v -- it can't return it as a result

I explained that we can't just go wantonly copying big data structures. There has to be a single pointer
to them.
Borrowed pointers let us have multiple pointers to the same data structure, as long as it's obvious who the
owner is -- the owner is responsible for freeing it/cleaning it up.
this is a bit misleading since &[]... is not just "a reference to a vector"...

No refcounting/etc. needed for managing v -- it gets deallocated automatically on exit from f
Typechecker checks that v is valid for the whole time sum uses it

A bad example

```
struct Cat { }

struct WebCam {
    target: &Cat
}

fn make_a_cat() {
    let cat = Cat::new();
    let webcam = WebCam::new(&cat);
    send_cat_to_moon(cat);
    take_photograph(&webcam);
}

fn take_photograph(webcam: &WebCam) {
    webcam.target.focus();
    webcam.snap();
}
```

Field that's a reference to a Cat

The typechecker will reject this code

The pointer to `cat` inside `webcam` is now dangling

This slide omits the definition for the static methods Cat::new and WebCam::new (since I didn't mention static methods in the talk). Also, I omitted field definitions for the Cat struct. Finally, the reference to Cat inside WebCam actually needs lifetime variables, which I didn't talk about.

assume Cat is not copyable...

This would crash in C++. Rust catches it at compile time. A different solution is to use GC (which would mean cat gets kept alive) but we don't want to force everything to use it. So in Rust,

code like this runs full speed. No GC overhead.

Borrowed pointers (summary)

- It's perfectly *safe* to borrow a pointer to data in a stack frame that will *outlive* your own
- It's also *efficient:* the compiler proves statically that lifetimes nest properly, so borrowed pointers need no automatic memory management
- Rust accomplishes both of these goals *without* making the programmer responsible for reasoning about pointer lifetimes (the compiler checks your assumptions)

Why bother?

- Rust makes you think a lot about borrowed pointers and ownership. What do you get in return?
    - The ability to write common patterns (interior pointers that can be returned from functions, lexically nested chains of borrows) and know that no dangling pointers or memory leaks will occur at runtime
    - You would also have to do the same reasoning if you were writing systems code in C++. Rust gives you to the tools to make that reasoning explicit and to help the compiler help you check it.
    - Rust's type system also helps avoid expensive copy operations that you didn't intend to do

PROBABLY SKIP NEXT BIT

Traits and pointers: an extended example

privacy annotation **doc comment**

```
pub trait Container {
    /// Return the number of elements in the
container
    fn len(&self) -> uint;

    /// Return true if the container contains no
elements
    fn is_empty(&const self) -> bool;
}
```

"A container, by definition, supports the `len` and `is_empty` operations"

I didn't use these remaining slides in the talk. They probably won't compile. Read at your own risk!

Trait inheritance

this method must be called on a mutable reference to a T that has the Mutable trait

```
pub trait Mutable: Container {
    /// Clear the container, removing all values.
    fn clear(&mut self);
}
```

"A mutable container, by definition, is a container that supports the additional `clear` operation"

Concrete type: HashMap

```
pub struct HashMap<K,V> {
    priv k0: u64,
    priv k1: u64,
    priv resize_at: uint,
    priv size: uint,
    priv buckets: ~[Option<Bucket<K, V>>],
}

struct Bucket<K,V> {
    hash: uint,
    key: K,
    value: V
}
```

(details aren't too important, I just wanted to show you the type that we're implementing Container on)

Traits and pointers: an extended example

"K is any type that has the Hash and Eq traits"

```
impl<K:Hash + Eq,V> Container for HashMap<K, V> {
    /// Return the number of elements in the map
    fn len(&const self) -> uint {
        self.size
    }

    /// Return true if the map contains no elements
    fn is_empty(&const self) -> bool {
        self.len() == 0
    }
}
```


pretty straightforward, just note the Hash + Eq syntax for multiple bounds
HashMap also has to implement Mutable, and then there's the whole Map trait, but no room for that...

The `Map` trait: more with borrowed pointers

```
pub trait Map<K, V>: Mutable {
    /// Return true if the map contains a value for the specified key
    fn contains_key(&self, key: &K) -> bool;

    /// Visit all keys
    fn each_key(&self, f: &fn(&K) -> bool) -> bool;

    /// Return a reference to the value corresponding to the key
    fn find<'a>(&'a self, key: &K) -> Option<&'a V>;

    /// Insert a key-value pair into the map. An existing value for a
    /// key is replaced by the new value. Return true if the key did
    /// not already exist in the map.
    fn insert(&mut self, key: K, value: V) -> bool;

    /// Removes a key from the map, returning the value at the key if the key
    /// was previously in the map.
    fn pop(&mut self, k: &K) -> Option<V>;
}
```

Change this to each

removed some methods for clarity
Notice that if you implement this trait, you *can* implement a hash map with C-style, no-overhead pointers (you *could* use automatic GC in the implementation but it doesn't force you to)
Graydon says font is too small

Borrowed pointers: an extended example

```
impl<K:Hash + Eq,V> Mutable for HashMap<K, V> {
    /// Clear the map, removing all key-value pairs.
    fn clear(&mut self) {
        for uint::range(0, self.buckets.len()) |idx| {
            self.buckets[idx] = None;
        }
        self.size = 0;
    }
}
```


Talk about for loops and closures more (and how they compile into actual loops)

Borrowed pointers: an extended example

```
impl<K:Hash + Eq,V> Map<K, V> for HashMap<K, V> {

    /// Return true if the map contains a value for
the specified key
    fn contains_key(&self, k: &K) -> bool {
        match self.bucket_for_key(k) {
            FoundEntry(_) => true,
            TableFull | FoundHole(_) => false
        }
    }



}
```


Talk about or-patterns. Otherwise, does this really need to be here?

Borrowed pointers example

```
impl<K:Hash + Eq,V> Map<K, V> for HashMap<K, V> {

     /// Visit all key-value pairs
    fn each<'a>(&'a self,
                blk: &fn(&K, &'a V) -> bool) -> bool {
        for self.buckets.each |bucket| {
            for bucket.each |pair| {
                if !blk(&pair.key, &pair.value) {
                    return false;
                }
            }
        }
        return true;
    }
}
```


talk about for-loop protocol more
talk about early-return and return-out-of-closures
Graydon says avoid explaining the for loop protocol

Borrowed pointers example

```
impl<K:Hash + Eq,V> Map<K, V> for HashMap<K, V> {

    /// Return a reference to the value corresponding
to the key
    fn find<'a>(&'a self, k: &K) -> Option<&'a V> {
        match self.bucket_for_key(k) {
            FoundEntry(idx) =>
               Some(self.value_for_bucket(idx)),
            TableFull | FoundHole(_) => None,
        }

    }

}
```


This is not too different from contains_key

Borrowed pointer example

```
impl<K:Hash + Eq,V> Map<K, V> for HashMap<K, V> {

     /// Removes a key from the map, returning the value at the key if
the key
    /// was previously in the map.
    fn pop(&mut self, k: &K) -> Option<V> {
        let hash = k.hash_keyed(self.k0, self.k1) as uint;
        self.pop_internal(hash, k)
    }

}
```


the interesting part is that we return the value by-move... but how to get this across without going into too many tedious details about pop_internal?

Miscellaneous Fun Stuff

- Lightweight unit testing (heavily used in Rust libraries):

```
#[test]
fn test_find() {
      let mut m = HashMap::new();
      assert!(m.find(&1).is_none());
      m.insert(1, 2);
      match m.find(&1) {
          None => fail!(),
          Some(v) => assert!(*v == 2)
      }
}
```

- `rustc map.rs --test -o maptest` generates an executable `maptest` plus code that runs tests and prints out neatly-formatted results

This is skippable

Benchmarking

```
#[bench]
fn bench_uint_small(b: &mut BenchHarness) {
    let mut r = rng();
    let mut bitv = 0 as uint;
    do b.iter {
        bitv |= (1 << ((r.next() as uint) % uint::bits));
    }
}
```

- `rustc --bench -o bitv_bench bitv.rs` generates a bitv_bench executable that runs this benchmark fn repeatedly and averages the results

this feature is still nascent

Macros

- We've seen a few macros already, `assert!` and `fail!`
- Macros allow you to extend Rust's syntax without burdening the compiler

```
macro_rules! fail(
    () => (
        fail!("explicit failure")
    );
    ($msg:expr) => (
        standard_fail_function($msg, file!(), line!())
    );
)
```


This code won't compile (I elided the gory details of how Rust implements fail)
macros also allow for static checking of printf arguments
fail! and assert! were once baked into the language, and now they're modular

Deriving

- Some traits can be automatically derived (the compiler writes the implementations for you)

```
/// The option type
#[deriving(Clone, Eq)]
pub enum Option<T> {
    None,
    Some(T),
}
```


Talk about clone?
Similar to deriving in Haskell

Any questions?

- Thanks to:
  - The Rust Team: Graydon Hoare, Patrick Walton, Brian Anderson, Niko Matsakis, John Clements, Jack Moffitt, Dave Herman
  - All of the Rust contributors: https://github.com/mozilla/rust/blob/master/AUTHORS.txt

List URLs for Rust, Servo, any projects that I drew on
if you had fun, organize something during the unconference time