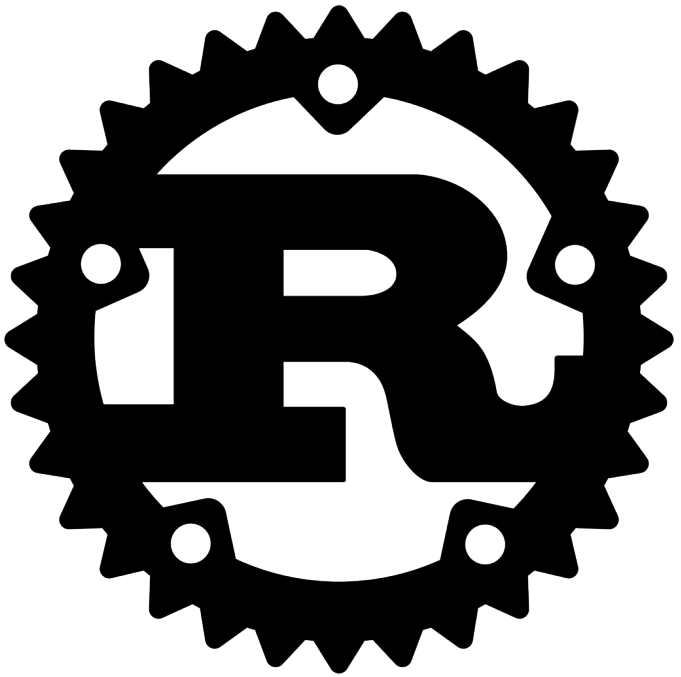
Rust



A safe, concurrent, practical language

Graydon Hoare
<graydon@mozilla.com>
October 2012

This is not a marketing talk

- Purpose:
  - Convince you there's something interesting here
  - Provide some technical details to whet your appetite

- Assuming:
  - You're a systems programmer
  - You know >3 existing non-toy languages
    - One of which is C++
    - One of which is ML, Haskell, C# or Scala
    - Lisp and Smalltalk folks: we love you too

Practical ≈ Realistic

- No silver bullets
- No free lunches
- Nothing new under the sun
- PL design has *>50 years* of history
- Most good ideas discovered in the first 20
- PL design work ≈ taste, selection, tradeoffs
- “New language” ≈ new balance, suited to times

Some Rust code: the Algol basics

```
fn main() {
    io::println("hello, world");
}
```

```
struct Point {x:int, y:int}
let a = Point {x:1, y:2};
assert 1 == a.x;
```

```
fn fact(x: int) -> int {
    if x == 1 {
        return 1;
    } else {
        return x * fact(x-1);
    }
}
```

```
enum Color {Red, Green, Blue}
let x = Red;
assert x != Blue;
match x {
    Red => foo(1),
    _   => bar(2)
}
```

```
let a: str = "hello";
let b: char = 'Ψ'; // Unicode
let c: i8: 0b1010_0000 | 0xf;
let d: u32: 0xdeadc0de;
let e: bool = true;
let f: (int, float) = (1, 1.2);
let g: [int] = [1,2,3,4];
```

```
fn foo() {
    let x = [1,2,3,4];
    let mut i = 0;
    while i < x.len() {
        bar(x[i]);
        i += 1;
    }
}
```

Some Rust code: the FP basics

Anonymous functions & type inference

```
[1,2,3].map(|x| x+1)
```

```
~[2,3,4]
```

Pattern matching & tagged unions

```
enum Shape {
    Circle(float),
    Square(float),
    Rect(float,float)
}

fn area(s: Shape) -> float {
    match s {
        Circle(r) => float::pi * (r * r),
        Square(s) => s * s,
        Rect(w,h) => w * h
    }
}
```

Seriously, everyone can do that stuff

- Rust picks up 4 *slightly* less-common things:
  1. Value types: all pointers & allocations are explicit
  2. Borrowing: static reasoning about pointer lifetimes
  3. Owned types: differentiating "move" from "copy"
  4. Traits: code reuse, generic bounds, vtables

- Rest of this talk will be about these 4 things
  - Note: 3 of 4 are about the *memory model*

Value types: overview

- Pointers explicit; non-pointer values *interior.*
- This is what all historical "systems languages" do: Algol, PL/1, C/C++, Pascal/Modula, etc.
- In newer languages it's becoming increasingly rare: C# and Go are a couple exceptions.
- We consider it an essential part of coding at this level of abstraction. No apologies.

Value types

Data Overhead

Rust:

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

64bits
64bits

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

64bits
64bits
64bits
64bits

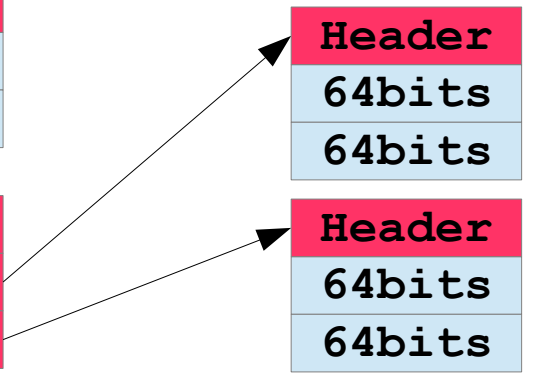
Java:

```
class Point { long x;
              long y; }
```

Header
64bits
64bits

```
class Rect { Point a;
             Point b; }
```

Header
64bits
64bits



Value types: cont'd

- These costs add up
  - Especially bad on arrays, "dense" collections
  - Count allocator slop, headers, alignment...

- Avoiding heap when you can *really matters*
  - PLDI'09 Mitchell & Sevitsky, Java heap costs:
    - 80% overhead in common library structures!
    - Worse on 64bit

Value types: cont'd

Pointers are always explicit, and there are 4 types

Rust syntax	Name	C++ analogy
`&T`	borrowed pointer	`T&`
`@T`	managed pointer	`std::shared_ptr<T>`
`~T`	owning pointer	`std::unique_ptr<T>`
`*T`	raw pointer	`T*`

Two of these are interesting, two are boring

The two boring pointer types

- `@T` (managed pointer): GC'ed, task-local, general purpose, shallow copies, cycles OK.

- `*T` (raw pointer): No guarantees, arithmetic allowed, dereferencing is unsafe. Exists for interop with C and places where you need to break the rules.

- Nothing more will be said about either of these.

Borrowing: overview

- Pointers (implicitly) statically qualified by the lifetime of the thing they point to.
- This is commonly called a *region system*. Several safe ones exist: ML Kit, Cyclone, Real-Time Java, ParaSail.
- Captures the *unsafe idiom* of arenas, stack-discipline in allocation lifetimes; arguably a special case of the generational GC hypothesis.

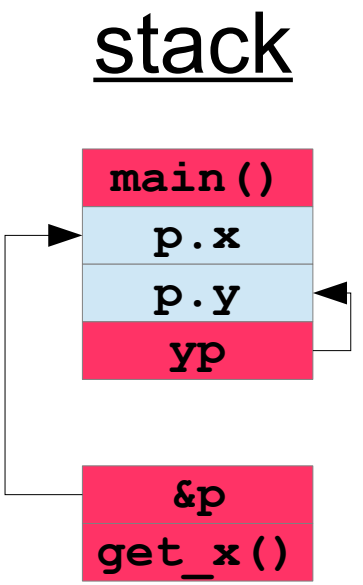
Borrowing

- In C++, `T&` is a pointer with *idiomatic meaning*
  - "someone else will (probably) keep referent alive"
  - is somewhat second class, can't reassign
- In Rust, `&T` means same, but it's *guaranteed*
  - If it compiles, it *will not* point to garbage memory
  - is first class, can copy, reassign, *return*

Borrowing: cont'd

- Plain values are allocated on the stack
- Pass (and return!) via borrowing, proven safe

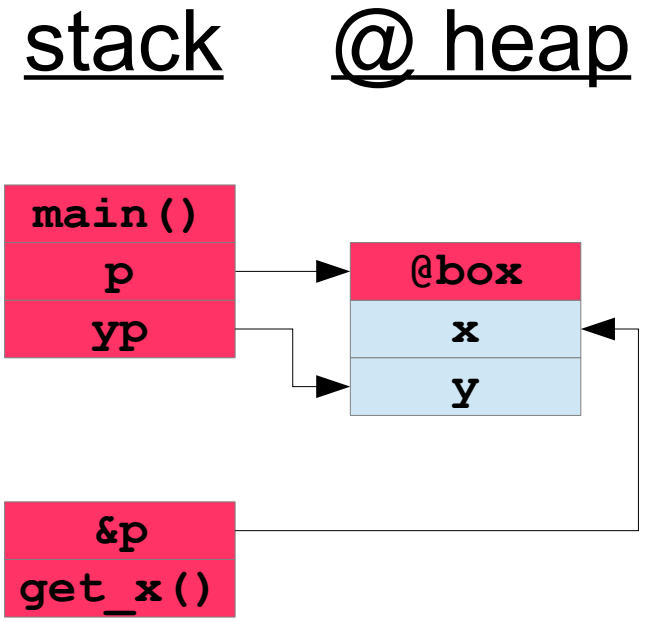
```
struct Point { x: int, y: int }

fn main() {
    let p = Point {x: 10, y: 11};
    let yp = get_y(&p);
}


fn get_y(pp: &r/Point) -> &r/int {
   return &pp.y;
}
```



Borrowing: cont'd

- Can borrow *any* pointer type
  - Managed and owning pointers auto-borrow

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

fn main() {
    let p = @Point {x: 10, y: 11};
    let yp = get_y(p);
}


fn get_y(pp: &r/Point) -> &r/int {
   return &pp.y;
}
```



Borrowing: conclusion

- Borrowing is fantastic!
  - The GC can completely ignore borrowed pointers
    - Possible to write pointer-rich Rust code w/o *any* GC
  - Zero runtime cost
    - No headers, no allocation slop, no GC
    - Can *safely* point into middle of other allocations
  - Guaranteed live
    - Subsume arenas, stack pointers, temporary uses
    - Used for environment capture in stack closures too!

Owned types: overview

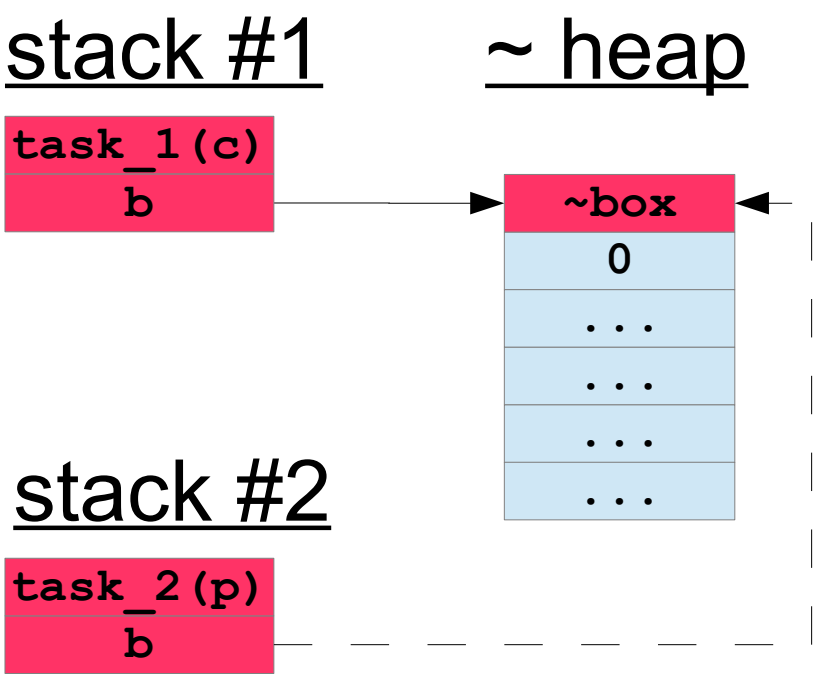
- Owning pointers go by many names: unique, linear / affine, substructural, ownership.
- Present in many languages: Clean, C++, Linear Lisp, Linear ML, Mercury.
- Simple idea: 1:1 relationship between pointer:pointee.
- Seems useless! But actually very useful.
- Supports *resource accounting* and *ownership transfer.*
- In Rust's case: between concurrent tasks.

Owned types: cont'd

- One diagram should make it clear
  - Two tasks, big message, O(1 ptr) send, no locks:

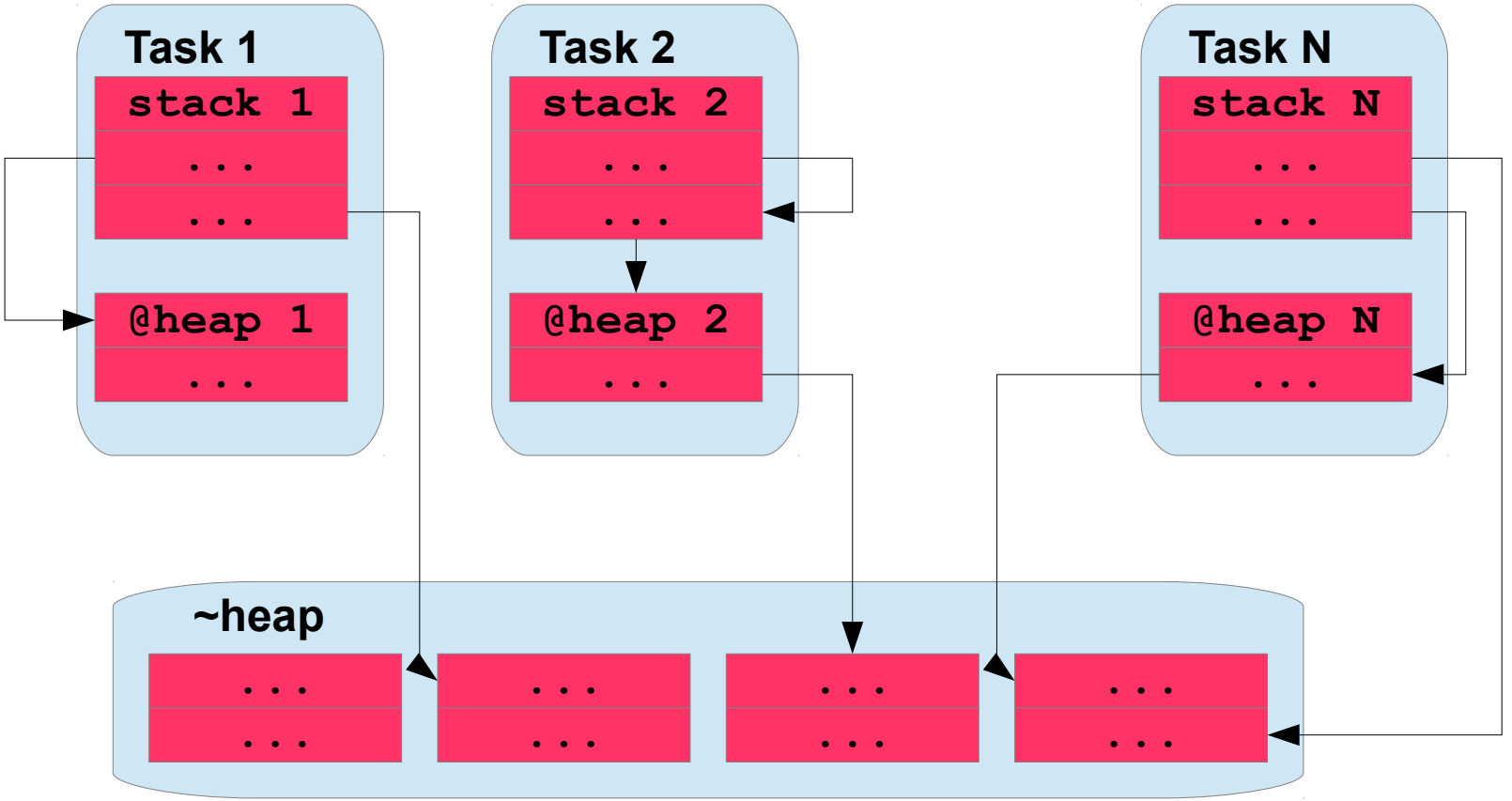
```
struct Msg { buf:[int * 1024] }

fn task_1(c: Chan<~Buf>) {
    let b = ~Msg {buf:[0,...]};
    c.send(move b);
    // b now deinitialized
}
```

```
fn task_2(p: Port<~Buf>) {
    let b = p.recv()
}
```



Owned types vs. managed

Many concurrent systems have a diagram like this: managed types are task-local; owned can be sent.



Owned types: conclusion

- Owning a type means cheap send w/o copy
- Also means other curious / subtle things:
  - Converting mutable value to (deep) immutable
  - Immediate free when pointer goes out of scope
  - Destructors, RAII, top-down resource release
  - Avoiding *all* unnecessary copies, even shallow ones (which carry GC or RC accounting costs)
  - Requires thinking about ownership, copy vs. move

Traits: overview

- Three different concepts mushed together:
  - Typeclasses (a la Haskell 98, C++ "concepts")
  - Existentials (a la C++ virtuals, Java interfaces)
  - Traits (a la Self, Scala, Fortress)

- Generally, "trait" ≈ a pair of method-sets:
  - Set of methods *required* on a type
  - Set of methods *provided* on a type

Traits

```
// Define a trait

trait ToStr {
    fn to_str() -> str;
}


// Implement a trait on a type

impl int : ToStr {
    fn to_str() -> str {
       int::to_str(self)
    }
}
```

Traits: cont'd

```
// Use a trait directly as a method on known type

let x : int = 10;
let s : str = x.to_str();


// Use a trait-method via a type-parameter bound

fn f<T:ToStr>(x: &T) {
    io::println(x.to_str());
}

f(10);  // Specializes f for int, likely inlines
```

Traits: cont'd

```
// Implement same trait for a different type

impl float : ToStr {
    fn to_str() -> str {
        float::to_str(self);
    }
}


// Use multiple different instances via vtables

let xs : [@ToStr] = [10 as @ToStr, 65.3 as @ToStr];
for xs.each |x| {
    print_one(x); // Borrow &ToStr, pass vtable
}

fn print_one(x: &ToStr) {
    io::println(x.to_str());
}
```

Traits: conclusion

- Traits are a nice sweet-spot:
  - Open: can add new methods to existing types
  - Safe: generic code states exact requirements
  - Efficient: usually bind statically, specialize, inline
  - Flexible: when heterogeneous, can use vtables
  - Simple: not entangled with "is-a", data model

Conclusion

- Hopefully you see stuff that's appealing
- Please come join in, help out, report bugs
  - http://www.rust-lang.org
  - http://github.com/mozilla/rust
  - irc.mozilla.org #rust
- Thanks!