CIS 198: Rust Programming

CIS 198: Rust Programm

Lecture 00: Hello, Rus

"Rus
prev

– rus

- E

This lecture online:

GitHub One-Page View • Slide View

Overview

- S
- F
- H
- M
- F
- F
- .

"Rust is a systems programming language that runs blaz
prevents nearly all segfaults, and guarantees thread safe

– rust-lang.org

- But what does that even *mean*?

Blazingly Fast

- Speed comparable to C/C++
- Rust compiles to native code
- Has no garbage collector
- Many abstractions have zero cost
- Fine-grained control over lots of things
- Pay for exactly what you need...
- ...and pay for most of it at compile time

Prevents Nearly All Segfaults

- F
- N
- (

- No null
- No uninitialized memory
- No dangling pointers
- No double free errors
- No manual memory management!

Guarantees Thread Safety

- A
- F
- F
- T

- Rust *does not allow* shared, mutable data
- Mutexes (and other atomics)
- Compiler rules for shared data (Send and Sync)

Functional

- S
- E
- (

- Algebraic datatypes (sum types / enum types)
- Pattern matching
- First-class functions
- Trait-based generics (interfaces)

Zero-Cost 100% Safe Abstraction

- \
  - Strict compile-time analysis removes need for runtim
  - Big concept: data ownership & borrowing
  - Other things like traits, generics, lifetimes, etc.

Do What You Want

- You can write `unsafe` code to get circumvent... most

- F
- N
- (
- T

D
201
201
201
201

Release Model: Trains

- F
- \

- Rust has a new stable release every six weeks
- Nightly builds are available, well, nightly
- Current stable: Rust 1.11
- Train model:

Date	Stable	Beta	Nightly
2016-08-25	🚂 1.11	🚄 1.12	🚅 1.13
2016-09-29	🚄 1.12	🚅 1.13	🚋 1.14
2016-11-10	🚅 1.13	🚋 1.14	🚃 1.15
2016-12-22	🚋 1.14	🚃 1.15	🚄 1.16

Development

- N
- S
- D
- M
- C
- V
- C

- Rust is led by the Rust Team, mostly at Mozilla Resea
- Very active community involvement - on GitHub, Red
  - Rust Source (GitHub)
  - Rust Internals Forum
  - /r/rust

Who Uses Rust?

- C
- S
- F
- r
- V
- Mozilla: Firefox, Servo
  - Firefox shipped a Rust mp4 track metadata parser las
- Skylight
- Dropbox
- Maidsafe
- OpenDnS
- wit.ai (Facebook)
- Codius
- F
- F

Some Big Rust Projects

- 8
- F
- V
- M
- F
- C
- C
- C
- Cargo
- Servo
- Piston
- mio
- Web
  - nickel.rs
  - iron
  - hyper
- Redox
- Rust itself!

Administrivia

- 8
- F
  V
- V
- Y
  c
- M

- 8 homeworks (50%), final project (40%)
- Participation (10%)
- Weekly Rust lecture: Tue. 4:30-6:00pm, Towne 321
- Mini-Course lecture: Tue. 6:00-7:30pm, Towne 100
- Piazza
  - We will be using Piazza for announcements; make sur
    gotten emails!
- Consult the website for the schedule, slides, and hon
- Class source material hosted on GitHub.
  - Corrections welcome via pull request/issue!
- Course is in development - give us feedback!

Administrivia: Homeworks (50%

- C
- N
  y
- F

- 8 homeworks.
- Released each Tuesday and (usually) due the followi
  Wednesday night at midnight.
- We will be using GitHub Classroom.
  - Click the link to make a private repo for every homew
    will be your submission.
- You get 5 late days (24 hours each). You may use up
  days on an assignment.
  - 20% penalty per day after 2 late days *or* if you're out
    days.
- Make sure your code is visible on GitHub *at midnight*
  - Late days applied automatically if you push to your re
    the due date. (*Do* let us know if you would like to use
    submission instead.)

Prerequisites

- M

Offic

- A
- C
  s

- CIS240 is suggested.
- Not a hard limit, although you may struggle with refe
  you haven't taken 240.
- Familiarity with Git and using the command line.
  - There are links on hw0 about both of these things.

Administrivia: Office Hours

- T
- C
- F
- F

- Monday, Tuesday, Wednesdays

Office hours held in the Levine 6th floor lounge.

- Any changes will be announced.
- Check the website or Google calendar for the up-to-d schedule.

Helpful Links

Hell

```
fn
}
```

- A
  t

- The Rust Book (our course textbook)
- Official Rust Docs
- Rust By Example
- Rust Playpen
  - Online editing and execution!

Let's Dive In!

Hello, Rust!

```
fn main() {
    println!("Hello, CIS 198!");
}
```

- All code blocks have links to the Rust playpen so you
  them!

Basic Rust Syntax

- \
- E
  c
- I
  s
- \

Variable Bindings

- E

- T
  &

```
let
let
{

} //

prin
```

- Variables are bound with `let`:

```
let x = 17;
```

- Bindings are implicitly-typed: the compiler infers bas
  context.
- If the compiler can't determine the type of a variable
  sometimes you have to add type annotations.

```
let x: i16 = 17;
```

- Variables are immutable by default:

```
let x = 5;
x += 1; // error: re-assignment of immutable variable x
let mut y = 5;
y += 1; // OK!
```

Variable Bindings

- F
- F
- A

```
fn f
  //
}
```

- F

- Bindings may be shadowed:

```
let x = 17;
let y = 53;
let x = "Shadowed!";
// x is not mutable, but we're able to re-bind it
```

- The first binding for x is shadowed until the second b goes out of scope.

```
let x = 17;
let y = 53;
{
    let x = "Shadowed!";
} // This second binding goes out of scope

println!("{}", x); // ==> 17
```

Functions

- N
  - Functions are declared with `fn`.
  - Return types are indicated with `->`.
  - Arguments are written `var: type`.

```
fn foo(x: u32) -> i32 {
  // ...
}
```

  - Functions are called with parens:

```
foo(42);
```

Functions

- Must explicitly define argument and return types.
  - The compiler could probably figure this out, but Rust'
    decided it was better practice to force explicit functio

- T

```
fn s
}
fn s


}
fn s
}
```

- T

Functions

-]

- The final expression in a function is its return value.
  - Use `return` for *early* returns from a function.

```
fn square(n: i32) -> i32 {
    n * n
}

fn squareish(n: i32) -> i32 {
    if n < 5 { return n; }
    n * n
}
```

```
fn square_bad(n: i32) -> i32 {
    n * n;
}
```

- The last one won't even compile!
  - Why? It ends in a semicolon, so it evaluates to `()`.

Unit

- The "nothing" type is called "unit", which is written `()`
  - The *type* `()` has only one value: `()`.

- (
  e
- (
- [
  i

```
fn t
fn b
fn c
```

Expressions

- E
  t
- A

- (Almost!) everything is an expression: something whi
  evaluates to a value.
  - Exception: variable bindings are not expressions.
- `()` is the default return type.
- Discard an expression's value by appending a semico
  it evaluates to `()`.
  - Hence, if a function ends in a semicolon, it returns `()`.

```
fn foo() -> i32 { 5 }
fn bar() -> () { () }
fn baz() -> () { 5; }
fn qux()       { 5; }
```

Expressions

```
///
///
///
///
fn f
}
```

- Because everything is an expression, we can bind ma
  to variable names:

```
let x = -5;
let y = if x > 0 { "greater" } else { "less" };
println!("x = {} is {} than zero", x, y);
```

- Aside: `"{}"` is Rust's basic string interpolation operat
  - Similar to Python, Ruby, C#, and others; like `printf`'s
    C/C++.

Comments

```
/// Triple-slash comments are docstring comments.
///
/// `rustdoc` uses docstring comments to generate
/// documentation, and supports **Markdown** formatting.
fn foo() {
    // Double-slash comments are normal.

    /* Block comments
     * also exist /* and can be nested! */
     */
}
```

Types

- E
- C
- N

- A
- F

Primitive Types

- I
- a
- A
- A

- `bool`: `true` and `false`.
- `char`: `'c'` or `'😺'` (`char`s are Unicode!).
- Numeric types: specify the signedness and bit size.
  - `i8`, `i16`, `i32`, `i64`, `isize`
  - `u8`, `u16`, `u32`, `u64`, `usize`
  - `f32`, `f64`
  - `isize` & `usize` are the size of pointers (and therefore machine-dependent size)
  - Literals can have a type suffix: `10i8`, `10u16`, `10.0f32`, `1`
  - Otherwise type inference defaults to `i32` or `f64`:
    - e.g. `10` defaults to `i32`, `10.0` defaults to `f64`.
- Arrays, slices, `str`, tuples.
- Functions.

Arrays

- T
- A
- M
- C
- H

```
let
let
```

- Initialized one of two ways:

```
let arr1 = [1, 2, 3]; // (array of 3 elements)
let arr2 = [2; 30];   // (array of 30 values of `2`)
println!("{}", arr1[1]); // ==> 2
```

- `arr1` is of type `[i32; 3]`; `arr2` is of type `[i32; 30]`
- Arrays types are written `[T; N]`.
  - `N` is a compile-time *constant*. Arrays cannot be resized
  - Array access is bounds-checked at runtime.
- Arrays are indexed with `[]` (like most other language

Slices

- T
- S
- &
- s

let
let
let
let

[1]str

size,

- Type `&[T]`.
- A "view" into an array *by reference*.
- Not created directly, but are borrowed from other va
- Can be mutable or immutable.
- How do you know when a slice is still valid?
  - Coming soon... (ownership shenanigans!)

```
let arr = [0, 1, 2, 3, 4, 5];
let total_slice = &arr;          // Slice all of `arr`
let total_slice = &arr[..];      // Same, but more explicit
let partial_slice = &arr[2..5]; // [2, 3, 4]
```

Strings

- F
- I
- (
  b

```
let
let
let
```

- Two types of Rust strings: `String` and `&str`.
- `String` is a heap-allocated, growable vector of chara
- `&str` is a type[1] that's used to slice into `String`s.
- String literals like `"foo"` are of type `&str`.

```
let s: &str = "galaxy";
let s2: String = "galaxy".to_string();
let s3: String = String::from("galaxy");
let s4: &str = &s3;
```

[1] `str` is an unsized type, which doesn't have a compile-ti
size, and therefore cannot exist by itself.

Tuples

- Fixed-size, ordered, heterogeneous lists
- Index into tuples with `foo.0`, `foo.1`, etc.
- Can be destructured in `let` bindings, and used for va
  bindings.

```
let foo: (i32, char, f64) = (72, 'H', 5.1);
let (x, y, z) = (72, 'H', 5.1);
let (a, b, c) = foo; // a = 72, b = 'H', c = 5.1
```

- T
- T
  p
- (

```
let
fn a
}
//
let
```

Function Objects

- (

```
let
let
```

- N
  (

- Two types:
  - Function pointers (a reference to a normal function)
  - Closures
- Types are much more straightforward than C functio
  pointers:

```
let x: fn(i32) -> i32 = square;
```

- Can be passed by reference:

```
fn apply_twice(f: &Fn(i32) -> i32, x: i32) -> i32 {
    f(f(x))
}

// ...

let y = apply_twice(&square, 5);
```

Casting

- 7
- T
  t
- A
- <
- (

- Cast between types with `as`:

```
let x: i32 = 100;
let y: u32 = x as u32;
```

- Naturally, you can only cast between types that are s
  cast between.
  - No casting `[i16; 4]` to `char`! (This is called a "non-sca
  - There are unsafe mechanisms to overcome this, if you
    what you're doing.

Vec<T>

```
// E
let

// \
let
v1.p
v1.p
v1.p

let

// \
let
let
```

- *The* standard library type
- This is in Rust's "prelude": you don't need to import a
  to use it.
- A `Vec` (read "vector") is a heap-allocated growable arr
  - (cf. Java's `ArrayList`, C++'s `std::vector`, etc.)
- `<T>` denotes a generic type.
  - The type of a `Vec` of `i32`s is `Vec<i32>`.
- Create `Vec`s with `Vec::new()` or the `vec!` macro.
  - Namespacing: `new` is a function defined for the `Vec` st

Vec<T>

let
let

- L

- \
  f

```rust
// Explicit typing
let v0: Vec<i32> = Vec::new();

// v1 and v2 are equal
let mut v1 = Vec::new();
v1.push(1);
v1.push(2);
v1.push(3);

let v2 = vec![1, 2, 3];

// v3 and v4 are equal
let v3 = vec![0; 4];
let v4 = vec![0, 0, 0, 0];
```

Vec<T>

- F
- F
- F
- F
- T
- M

```
let
let
prin
```

```
let v2 = vec![1, 2, 3];
let x = v2[2]; // 3
```

- Like arrays, vectors can be indexed with `[]`.
  - Vectors must be indexed with `usize` values (guarante the same size as a pointer).
  - Other values can be cast to `usize`:

```
let i: i8 = 2;
let y = v2[i as usize];
```

- Vectors has an extensive stdlib method list, which ca found at the offical Rust documentation.

References

- Reference *types* are written with `&`: `&i32`.
- References can be taken with `&` (like C/C++).
- References can be *dereferenced* with `*` (like C/C++).
- References are guaranteed to be valid.
  - Validity is enforced through compile-time checks!
- These are *not* the same as (raw) pointers!
- More ownership shenanigans next week.

```rust
let x = 12;
let ref_x = &x;
println!("{}", *ref_x); // 12
```

Control Flow

```
if x
} el
} el


}
```

- N
- E
  S

```
if x

}
```

If Statements

- L
- W

```
let
whil


}
```

```
if x > 0 {
    10
} else if x == 0 {
    0
} else {
    println!("Not greater than zero!");
    -10
}
```

- No parens necessary.
- Entire if statement is a single expression and will eva
  single value
  - Every arm must end with an expression of the same t
  - That type must be `()` if you omit the `else` branch:

```
if x <= 0 {
    println!("Too small!")
}
```

Loops

- l

- Loops come in three flavors: `while`, `loop`, and `for`.
  - `break` and `continue` exist just like in most languages
- `while` works just like you'd expect:

```
let
loop


}
```

```
let mut x = 0;
while x < 100 {
    x += 1;
    println!("x: {}", x);
}
```

Loops

- f

```
// l
for

}
```

- `loop` is equivalent to `while true`.
  - Plus, the compiler can make optimizations knowing th infinite.

```
let mut x = 0;
loop {
    x += 1;
    println!("x: {}", x);
}
```

Loops

- s
- (

```
let
for

}
```

- `for` is the most different from most C-like languages
  - `for` loops use an *iterator expression*:
  - `n..m` creates an iterator from n to m (*exclusive*).
  - `n...m` (*inclusive* range) is currently experimental!

```
// Loops from 0 to 9.
for x in 0..10 {
    println!("{}", x);
}
```

Loops

```
let

matc

}
```

- p
-]
- _

- Some data structures can be used as iterators, like V
- Only slices can be iterated over, not arrays.

```
let xs = [0, 1, 2, 3, 4];
for x in &xs {
    println!("{}", x);
}
```

Match statements

```
let
let

matc




}
```

- T
- I
- \
- U
- F

```
let x = 3;

match x {
    1 => println!("one fish"),  // <- comma required
    2 => {
        println!("two fish");
        println!("two fish");
    },  // <- comma optional when using braces
    _ => println!("no fish for you"), // "otherwise" case
}
```

- `match` takes an expression (`x`) and branches on a list `=> expression` statements.
- The entire match evaluates to one expression.
  - Like `if`, all arms must evaluate to the same type.
- `_` is commonly used as a catch-all (cf. Haskell, OCaml

Match statements

- M
- (
- (
- \
  k
- E
  (

```
let x = 3;
let y = -3;

match (x, y) {
    (1, 1) => println!("one"),
    (2, j) => println!("two, {}", j),
    (_, 3) => println!("three"),
    (i, j) if i > 5 && j < 0 => println!("On guard!"),
    (_, _) => println!(":<"),
}
```

- The matched expression can be any expression (l-val
  including tuples and function calls.
  - Matches can bind variables. _ is a throw-away variabl
- You *must* write an exhaustive match in order to comp
- Use `if`-guards to constrain a match to certain condit
- Patterns can get very complex, as we'll see later.

Macros!

- F
- U

  p

```
prin
// =
prin
// =
```

- Macros are like functions, but they're named with `!` a
- Can do generally very powerful stuff.
  - They actually generate code at compile time!
- Call and use macros like functions.
- You can define your own with `macro_rules! macro_` blocks.
  - These are *very* complicated. More later!
- Because they're so powerful, a lot of common utilitie defined as macros.

`print!` & `println!`

- l
  s

```
let
// f
```

- Print stuff out.
- Use `{}` for general string interpolation, and `{:?}` for
  printing.
  - Some types can only be printed with `{:?}`, like arrays

```
print!("{}, {}, {}", "foo", 3, true);
// => foo, 3, true
println!("{:?}, {:?}", "foo", [1, 2, 3]);
// => "foo", [1, 2, 3]
```

`format!`

- E
- [

  e

```
if x

}
```

- Uses `println!`-style string interpolation to create for
  `String`s.

```
let fmted = format!("{}, {:x}, {:?}", 12, 155, Some("Hello"
// fmted == "12, 9b, Some("Hello")"
```

panic!(msg)

- a
- a
- l

```
#[te
fn t



}
```

- Exits current task with given message.
- Don't do this lightly! It is better to handle and report explicitly.

```
if x < 0 {
    panic!("Oh noes!");
}
```

`assert!` & `assert_eq!`

- `assert!(condition)` panics if `condition` is `false`.
- `assert_eq!(left, right)` panics if `left != right`.
- Useful for testing and catching illegal conditions.

```
#[test]
fn test_something() {
    let actual = 1 + 2;
    assert!(actual == 3);
    assert_eq!(3, actual);
}
```

`unreachable!()`

- S

```
fn s

}
```

- Used to indicate that some code should not be reach
- `panic!`s when reached.
- Can be useful to track down unexpected bugs (e.g. optimization bugs).

```
if false {
    unreachable!();
}
```

F

`unimplemented!()`

- Shorthand for `panic!("not yet implemented")`

```
fn sum(x: Vec<i32>) -> i32 {
    // TODO
    unimplemented!();
}
```

Rust Environment & To

- r
  p
- r
- T
  c

Rustc

- F
- (
- E
- F

- `rustc program.rs` compiles `program.rs` into an exec
  `program`.
  - Warnings are enabled by default.
  - Read all of the output! It may be verbose but it is *very*
- `rustc` doesn't need to be called once for each file like
  - The build dependency tree is inferred from module d
    in the Rust code (starting at `main.rs` or `lib.rs`).
- Typically, you'll instead use `cargo`, Rust's package ma
  and build tool.

Cargo

- C
  c
- M

```
[pac
name
vers
auth

[dep
uuid
rand

[pro
opt-
debu
```

- Rust's package manager, build tool
- Create a new project:
  - `cargo new project_name` (library)
  - `cargo new project_name --bin` (executable)
- Build your project: `cargo build`
- Run your tests: `cargo test`

Cargo.toml

- Cargo uses the `Cargo.toml` file to declare and manag
  dependencies and project metadata.
  - TOML is a simple format similar to INI.
- More in your first homework assignments.

- A
- C
- A
  s
- U
  C
- \

```
#[te
fn i

}
```

```
[package]
name = "Rust"
version = "0.1.0"
authors = ["Ferris <cis198@seas.upenn.edu>"]

[dependencies]
uuid = "0.1"
rand = "0.3"

[profile.release]
opt-level = 3
debug = false
```

`cargo test`

- N
- F
- F
  g

- A test is any function annotated with `#[test]`.
- `cargo test` will run all annotated functions in your p
- Any function which executes without crashing (`panic`
  succeeds.
- Use `assert!` (or `assert_eq!`) to check conditions (an
  on failure)
- You'll use this in HW01.

```
#[test]
fn it_works() {
    // ...
}
```

H

- [
- I
  F
- S
  y

`cargo check`

- Not available by default!
- Run `cargo install cargo-check` to install it.
- Functionally the same as `cargo build`, but doesn't a
  generate any code.
  - => Faster!

HW00: Hello Cargo & Hello F

- [
- I
  a

- Due Wednesday, 2016-09-07, 11:59pm.
- Install `rustup`: manages installations of multiple vers
  Rust.
  - Supports Linux / OS X / Windows.
  - Or use the 19x VM.
- Submitting with GitHub Classroom is as easy as ~~pie~~ p
  your private repo.

HW01: Finger Exercises

- C
- S
- M

Som

- Due Wednesday, 2016-09-07, 11:59pm.
- Introduction to Rust with "finger exercises". Use this
  a resource!
  - Sieve of Eratosthenes, Tower of Hanoi

Next Time

- Ownership, references, borrowing
- Structured data: structs, enums
- Methods

Some code examples taken from *The Rust Programming L*