CIS 198: Intro to COBOL

CIS 198: Intro to COBOL

- Designed in 1959 (57 years ago!)
- We will be using the COBOL2014 standard.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. hello-world.
PROCEDURE DIVISION.
    DISPLAY "Hello, world!"
    .
```

CIS 198: Rust Programming

Lecture 00: Hello, Rust!

This lecture online:
GitHub One-Page View • Slide View

Overview

"Rust is a systems programming language that runs blazingly fast, prevents nearly all segfaults, and guarantees thread safety." – rust-lang.org

What *is* Rust?

Rust is:

- Fast
- Safe
- Functional
- Zero-cost

Fast

- Rust compiles to native code
- Rust has no garbage collector
- Most 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

Safe

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

Functional

- First-class functions
- Trait-based generics
- Algebraic datatypes
- Pattern matching

Zero-Cost 100% Safe Abstractions

- Rust's defining feature
- Strict compile-time checks remove need for runtime
- Big concept: Ownership

Release Model

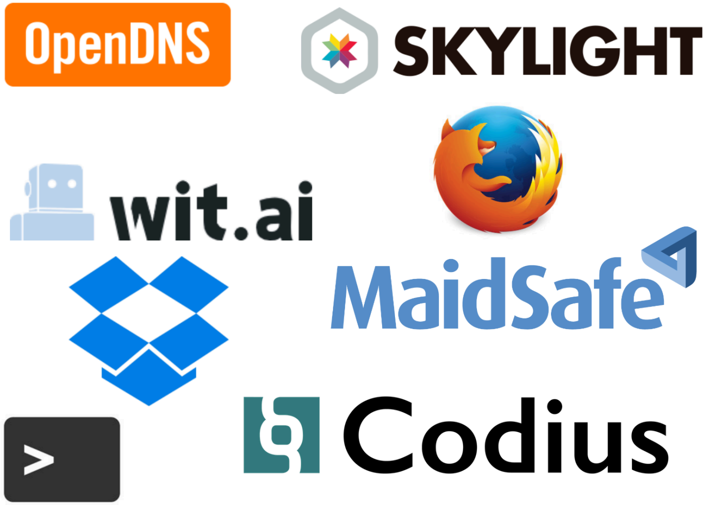
- Rust has a new stable release every six weeks
- Nightly builds are available, well, nightly
- Current stable: Rust 1.5
- Rust 1.6 will be out tomorrow (1/21)!
  - This is the version we'll be using in this class
- Train model:

Date	Stable	Beta	Nightly
2015-12-10	🚂 1.5	🚝 1.6	🚄 1.7
2016-01-21	🚝 1.6	🚄 1.7	🚅 1.8
2016-03-03	🚄 1.7	🚅 1.8	🚄 1.9

Development

- Rust is led by the Rust Team, mostly at Mozilla Research.
- Very active community involvement - on GitHub, Reddit, irc.
    - Rust Source
    - Rust Internals Forum
    - /r/rust

Who Uses Rust?



Big Rust Projects

- Servo
- Piston
- MIO
- nickel.rs
- iron
- lalrpop
- cargo
- Rust itself!

Administrivia

- 8-9 homeworks (50%), final project (40%) (may change)
- Participation (10%)
- Weekly Rust lecture: Wed. 4:30-6:00pm, Towne 321
- Mini-Course lecture: Tue. 6:00-7:30pm, Berger Auditorium
- Piazza
  - We will be using Piazza for announcements; make sure you have gotten emails!
- Consult the website for the schedule, slides, and homework.
- Class source material generally hosted on GitHub.
  - Corrections welcome via pull request/issue!
- Course is in development - give us feedback!

Administrivia: Office Hours

- David - Mon 4:30-6:00pm
- Terry - Tues 6:00-7:30pm
- Kai - Weds 6:00-7:00pm

Office hours held in the Levine 6th floor lounge.

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

Administrivia: Homeworks (50%)

- 8-9 homeworks.
- Released on Wednesdays and (usually) due the following Wednesday night at midnight.
- We will be using Classroom for GitHub.
    - Click the link to make a private repo for every homework, which will be your submission.
- Students start the semester with a total of 5 late days, which provide an extra 24 hours each. You may use up to 2 late days on an assignment.
    - If (and only if) you submit an assignment more than 2 days late or are out of late days, there is a 20% penalty per day beyond the late day extension.

Helpful Links

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

Let's Dive In!

Hello, Rust!

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

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

Basic Rust Syntax

Variable Bindings

- Variables are bound with `let`:

```
let x = 17;
```

- Bindings are implicitly-typed: the compiler infers based on context.
- The compiler can't always determine the type of a variable, so sometimes you have to add type annotations.

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

- Variables are inherently immutable:

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

Variable Bindings

- 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 shadowed binding for `x` above lasts until it goes out of scope.
- Above, we've effectively lost the first binding, since both `x`s are in the same scope.
- Patterns may also be used to declare variables:

```
let (a, b) = ("foo", 12);
```

Expressions

- (Almost!) everything is an expression: something which returns a value.
  - Exception: variable bindings are not expressions.
- The "nothing" type is called "unit", which is written `()`.
  - The *type* `()` has only one value: `()`.
  - `()` is the default return type.
- Discard an expression's value by appending a semicolon. Now it returns `()`.
  - Hence, if a function ends in a semicolon, it returns `()`.

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

Expressions

- Because everything is an expression, we can bind many things 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 (most basic) string interpolation operator
    - Similar to Python, Ruby, C#, and others; like `printf`'s `"%s"` in C/C++.

Comments

```rust
/// 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

Primitive Types

- `bool`: spelled `true` and `false`.
- `char`: spelled like `'c'` or `'😺'` (`char`s are Unicode!).
- Numerics: specify the signedness and size.
  - `i8`, `i16`, `i32`, `i64`, `isize`
  - `u8`, `u16`, `u32`, `u64`, `usize`
  - `f32`, `f64`
  - `isize` & `usize` are the size of pointers (and therefore have machine-dependent size)
  - Literals are spelled like `10i8`, `10u16`, `10.0f32`, `10usize`.
  - Type inference for non-specific literals default to `i32` or `f64`:
    - e.g. `10` defaults to `i32`, `10.0` defaults to `f64`.
- Arrays, slices, `str`, tuples.
- Functions.

Arrays

- Arrays are generically of type `[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 languages:
  - `arr[3]` gives you the 4th element of `arr`

```
let arr1 = [1, 2, 3]; // (array of 3 elements)
let arr2 = [2; 32];   // (array of 32 `2`s)
```

Slices

- Generically of type `&[T]`
- A "view" into an array by reference
- Not created directly, but are borrowed from other variables
- Mutable or immutable
- How do you know when a slice is still valid? Coming soon...

```
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

- Two types of Rust strings: `String` and `&str`.
- `String` is a heap-allocated, growable vector of characters.
- `&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-time known 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

```rust
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
```

Casting

- Cast between types with `as`:

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

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

`Vec<T>`

- A standard library type: you don't need to import anything.
- A `Vec` (read "vector") is a heap-allocated growable array.
  - (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.
  - `Vec::new()` is an example of namespacing. `new` is a function defined for the `Vec` struct.

Vec<T>

```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>

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

- Like arrays, vectors can be indexed with `[]`.
    - You can't index a vector with an i32/i64/etc.
    - You must use a `usize` because `usize` is guaranteed to be the same size as a pointer.
    - Other integers can be cast to `usize`:

      ```rust
      let i: i8 = 2;
      let y = v2[i as usize];
      ```
- Vectors has an extensive stdlib method list, which can be found at the offical Rust documentation.

References

- Reference *types* are written with an `&`: `&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 pointers!
- Reference lifetimes are pretty complex, as we'll explore later on in the course.

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

Control Flow

If Statements

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

- No parens necessary.
- Entire if statement evaluates to one expression, so every arm must end with an expression of the same type.
    - That type can be unit `()`:

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

Loops

- 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:

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

Loops

- `loop` is equivalent to `while true`, a common pattern.
    - Plus, the compiler can make optimizations knowing that it's infinite.

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

Loops

- `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).
  - Some data structures can be used as iterators, like arrays and `Vec`s.

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

let xs = [0, 1, 2, 3, 4];
// Loop through elements in a slice of `xs`.
for x in &xs {
    println!("{}", x);
}
```

Functions

```
fn foo(x: T, y: U, z: V) -> T {
    // ...
}
```

- `foo` is a function that takes three parameters:
    - `x` of type `T`
    - `y` of type `U`
    - `z` of type `V`
- `foo` returns a `T`.
- Must explicitly define argument and return types.
    - The compiler is actually smart enough to figure this out for you, but Rust's designers decided it was better practice to force explicit function typing.

Functions

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

```rust
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 `()`.

Function Objects

- Several things can be used as function objects:
  - Function pointers (a reference to a normal function)
  - Closures (covered later)
- Much more straightforward than C function pointers:

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

- Can be passed by reference:

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

// ...

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

Macros!

- Macros are like functions, but they're named with `!` at the end.
- 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_name` blocks.
  - These are *very* complicated. More later!
- Because they're so powerful, a lot of common utilities are defined as macros.

`print!` & `println!`

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

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

format!

- Uses println!-style string interpolation to create formatted Strings.

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

panic!(msg)

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

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

`unreachable!()`

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

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

unimplemented!()

- Shorthand for `panic!("not yet implemented")`
- You'll probably see this in your homework a lot!

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

Match statements

```rust
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 of `value => 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

```
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-value), including tuples and function calls.
    - Matches can bind variables. `_` is a throw-away variable name.
- You *must* write an exhaustive match in order to compile.
- Use `if`-guards to constrain a match to certain conditions.
- Patterns can get very complex, as we'll see later.

Rust Environment & Tools

Rustc

- Rust's compiler is `rustc`.
- Run `rustc your_program.rs` to compile into an executable `your_program`.
    - Things like warnings are enabled by default.
    - Read all of the output! It may be verbose but it is *very* useful.
- `rustc` doesn't need to be called once for each file like in C.
    - The build dependency tree is inferred from module declarations in the Rust code (starting at `main.rs` or `lib.rs`).
- Typically, you'll instead use `cargo`, Rust's package manager and build system.

Cargo

- 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`
  - These get tedious to type, so shell alias to your heart's content, e.g., `cargob`/`cb` and `cargot`/`ct`
- Magic, right? How does this work?

Cargo.toml

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

```toml
[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`

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

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

`cargo check`

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

HW00: Hello Cargo & Hello Rust

- Due Monday, 2016-01-25, 11:59pm.
- Install `multirust`: manages installations of multiple versions of Rust.
    - Similar to `rvm`, `virtualenv`.
    - Linux, OS X, Windows (MSYS2)
- Install 1.5 now if you want (updating is easy: `multirust update stable`).
    - 1.6 comes out tomorrow! (1/21)
- Submitting with Classroom for GitHub is as easy as ~~pie~~ pushing to your private repo.

HW01: Finger Exercises

- Due Wednesday, 2016-01-27, 11:59pm.
- Introduction to Rust with "finger exercises". Use this lecture as 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 Language*.