Intro To Rust

`struct`s, `enum`s, pattern matching, and other basic syntax

Cooper Pierce & Jack Duvall

Carnegie Mellon University

Discord server: https://discord.gg/V8YfmNPYRh

Course website: https://rust-stuco.github.io/

Knowledge check: https://forms.gle/inugrupwnUj2hM3B6

Outline

Cooper Pierce

- SCS Senior
- Rust Experience: mostly around LSP & Compiler implementations
- Links:
  - https://github.com/kopecs

Jack Duvall

- SCS Senior
- Rust Experience: Personal Projects, 15-451, FB Internship
- Links:
  - https://github.com/duvallj
  - https://duvallj.pw

Course Goals

Course Goals

- Give an overview of Rust's features

Course Goals

- Give an overview of Rust's features
- Be up-to-date with the language (Rust 2021)

Course Goals

- Give an overview of Rust's features
- Be up-to-date with the language (Rust 2021)
- Take you from beginner to advanced rust topics

Course Goals

- Give an overview of Rust's features
- Be up-to-date with the language (Rust 2021)
- Take you from beginner to advanced rust topics

To these ends, familiarity with topics covered in 15-122 and 15-150 is assumed.

Course Goals

- Give an overview of Rust's features
- Be up-to-date with the language (Rust 2021)
- Take you from beginner to advanced rust topics

To these ends, familiarity with topics covered in 15-122 and 15-150 is assumed.

Course Non-Goals

Course Goals

- Give an overview of Rust's features
- Be up-to-date with the language (Rust 2021)
- Take you from beginner to advanced rust topics

To these ends, familiarity with topics covered in 15-122 and 15-150 is assumed.

Course Non-Goals

- Replace https://doc.rust-lang.org/stable/book/ as the premier way to learn Rust

Course Goals

- Give an overview of Rust's features
- Be up-to-date with the language (Rust 2021)
- Take you from beginner to advanced rust topics

To these ends, familiarity with topics covered in 15-122 and 15-150 is assumed.

Course Non-Goals

- Replace https://doc.rust-lang.org/stable/book/ as the premier way to learn Rust
- Be comprehensive

Syllabus Stuff

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
    - $\Rightarrow$ we must take attendance

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
  - $\Rightarrow$ we must take attendance
  - Just let us know via email/Discord if you expect to miss a class

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
  - $\Rightarrow$ we must take attendance
  - Just let us know via email/Discord if you expect to miss a class
    - `rust-stuco-staff@lists.andrew.cmu.edu` goes to both of us

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
  - $\Rightarrow$ we must take attendance
  - Just let us know via email/Discord if you expect to miss a class
    - `rust-stuco-staff@lists.andrew.cmu.edu` goes to both of us
- Participation: 50%

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
  - $\Rightarrow$ we must take attendance
  - Just let us know via email/Discord if you expect to miss a class
    - `rust-stuco-staff@lists.andrew.cmu.edu` goes to both of us
- Participation: 50%
- Homework: 10%

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
  - $\Rightarrow$ we must take attendance
  - Just let us know via email/Discord if you expect to miss a class
    - `rust-stuco-staff@lists.andrew.cmu.edu` goes to both of us
- Participation: 50%
- Homework: 10%
- Midterm: 5%

Syllabus Stuff

- StuCo policy: **Only 2 unexcused absences**
  - $\Rightarrow$ we must take attendance
  - Just let us know via email/Discord if you expect to miss a class
    - `rust-stuco-staff@lists.andrew.cmu.edu` goes to both of us
- Participation: 50%
- Homework: 10%
- Midterm: 5%
- Final: 35%

Outline

Why Rust?

Why Rust?

- C & C++ need to be replaced

Why Rust?

- C & C++ need to be replaced
- Goals:

Why Rust?

- C & C++ need to be replaced
- Goals:
  - Great performance, easily

Why Rust?

- C & C++ need to be replaced
- Goals:
  - Great performance, easily
  - Reliable and safe

Why Rust?

- C & C++ need to be replaced
- Goals:
  - Great performance, easily
  - Reliable and safe
  - Good tooling

Why Rust?

- C & C++ need to be replaced
- Goals:
  - Great performance, easily
  - Reliable and safe
  - Good tooling
- Incorporate modern ideas in progamming language design

Rust Goal: Great Performance, Easily

Rust Goal: Great Performance, Easily

- C-like abstractions, program “close to the hardware”

Rust Goal: Great Performance, Easily

- C-like abstractions, program "close to the hardware"
- "Zero-Cost Abstractions": paid at compile time, code is fast at runtime

Rust Goal: Great Performance, Easily

- C-like abstractions, program “close to the hardware”
- “Zero-Cost Abstractions”: paid at compile time, code is fast at runtime
- Use LLVM for robust optimizations on many platforms

Rust Goal: Reliable And Safe

Rust Goal: Reliable And Safe

- Ideally: "if it compiles, it's safe"

Rust Goal: Reliable And Safe

- Ideally: “if it compiles, it’s safe”
- Safe memory management is core of the language

Rust Goal: Reliable And Safe

- Ideally: “if it compiles, it’s safe”
- Safe memory management is core of the language
- Comprehensive standard library

Code You Can't Write in Rust

```c
int *my_cool_fn(void) {
    int x = 0;
    return &x;
}
```

Code You Can't Write in Rust

```c
int *my_cool_fn(void) {
    int x = 0;
    return &x;
}
```

```rust
fn my_cool_fn() -> &i32 {
    let x = 0;
    return &x;
}
```

Code You Can't Write in Rust

```
void foo() {
    auto v = std::vector{1, 2, 3, 4};
    auto& x = v[0];
    v.push_back(5);
    v[0] = 6;
    std::cout << x << " == " << v[0] << std::endl;
}
```

Code You Can't Write in Rust

```cpp
void foo() {
    auto v = std::vector{1, 2, 3, 4};
    auto& x = v[0];
    v.push_back(5);
    v[0] = 6;
    std::cout << x << " == " << v[0] << std::endl;
}
```

```rust
fn foo() {
    let mut v = vec![1, 2, 3, 4];
    let x = &v[0];
    v.push(5);
    v[0] = 6;
    println!("{} == {}", x, v[0]);
}
```

Rust Goal: Good Tooling

Rust Goal: Good Tooling

- Easy to install on any supported platform

Rust Goal: Good Tooling

- Easy to install on any supported platform
- Great package manager

Rust Goal: Good Tooling

- Easy to install on any supported platform
- Great package manager
- Robust package ecosystem

Rust Goal: Good Tooling

- Easy to install on any supported platform
- Great package manager
- Robust package ecosystem
- Linting, code formatting, documentation, testing, and autocomplete are first-class features

Outline

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.

Hello, world!

```rust
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.

Hello, world!

```rust
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.
- Optionally, we can type annotate these.

Hello, world!

```
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.
- Optionally, we can type annotate these.
- Function like macros have a `!` at the end when applying them.

Hello, world!

```rust
fn main() -> () {
    let course: i32 = 98008;
    println!("Welcome to {}!", course);
}
```

- As in C, the entry point of our program is `main`.
- Here we're returning unit, `()`, but we can return a couple different types from `main`.
- We can introduce variable bindings with `let`.
- Optionally, we can type annotate these.
- Function like macros have a `!` at the end when applying them.
- We can print values using `println!` or `print!` in the same way we would with `printf`.

Mutable variables

In most imperative languages, variables are mutable by default.

```
int fact(int n) {
    int ans = 1;
    while (n) {
        ans *= n;
        n--;
    }
    return ans;
}
```

If we want a variable to be immutable we have to enforce this with a keyword like `const` or similar.

Rust, on the other hand, flips this. If we try the same in Rust:

```rust
fn fact(n: u32) -> u32 {
    let ans = 1;
    while n != 0 {
        ans *= n;
        n -= 1;
    }
    ans
}
```

we'd see an error like

```
error[E0384]: cannot assign to immutable argument `n`
 --> src/lib.rs:5:17
  |
1 |         fn fact(n: u32) -> u32 {
  |                 - help: consider making this binding mutable: `mut n`
...
5 |                 n -= 1;
  |                 ^^^^^^ cannot assign to immutable argument
```

In order to mark a variable as mutable, we need to have `mut` at the binding site.

```
fn fact(mut n: u32) -> u32 {
    let mut ans = 1;
    while n != 0 {
        ans *= n;
        n -= 1;
    }
    ans
}
```

This then permits later assignments through that binding.

Shadowing

```
fn main() {
    let x = 1;
    println!("x is {}", x);
    let x = 98008;
    println!("x is {}", x);
}
```

What about this code? Does it run afoul of our rules about changing variables?

Shadowing

```
fn main() {
    let x = 1;
    println!("x is {}", x);
    let x = 98008;
    println!("x is {}", x);
}
```

What about this code? Does it run afoul of our rules about changing variables?

No! We haven't changed anything here—there just happens to be a second, new variable we've also called `x`.

While at first this might seem similar to mutating the same variable, there are many semantic differences.

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

Here we've declared two variables which happen to have the same name.

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

Here we've declared two variables which happen to have the same name.

```
let mut fruits = ["mango", "apple", "banana"];
fruits = fruits.len();
```

While at first this might seem similar to mutating the same variable, there are many semantic differences.

```
let fruits = ["mango", "apple", "banana"];
let fruits = fruits.len();
```

Here we've declared two variables which happen to have the same name.

```
let mut fruits = ["mango", "apple", "banana"];
fruits = fruits.len();
```

This, to the contrary, results in a compiler error! We can't assign a value of type `usize` to a variable of type `[&str; 3]`.

References and Borrowing

Instead of working directly with pointers (often called “raw” pointers in Rust), we’ll typically use references instead.

```rust
fn main() {
    let x = 9;
    let y = 2;
    assert_eq!(compute_sum(&x, &y), 11);
}

fn compute_sum(a: &i32, b: &i32) -> i32 {
    a + b
}
```

Mutable References

What if we want to mutate a value through a reference?

```rust
fn main() {
    let x = 0;
    incr(&x);
    assert_eq!(x, 1);
}
```

```rust
fn incr(x: &i32) {
    *x += 1
}
```

Mutable References

What if we want to mutate a value through a reference?

```
fn main() {
    let x = 0;
    incr(&x);
    assert_eq!(x, 1);
}
```

```
fn incr(x: &i32) {
    *x += 1
}
```

Doesn't work!

```
error[E0594]: cannot assign to `*x`, which is behind a `&` reference
 --> src/main.rs:8:13
  |
7 |         fn incr(x: &i32) {
  |                    ---- help: consider changing this to be a mutable reference: `&mut i32`
8 |             *x += 1
  |             ^^^^^^^ `x` is a `&` reference, so the data it refers to cannot be written
```

If we want a mutable reference we need to ask for it explicitly:

```
fn incr(x: &mut i32) {
    *x += 1
}
```

If we want a mutable reference we need to ask for it explicitly:

```
fn incr(x: &mut i32) {
    *x += 1
}
```

and we need to be explicit when borrowing:

```
fn main() {
    let mut x = 0;
    incr(&mut x);
    assert_eq!(x, 1);
}
```

Note that in order to borrow `x` mutably, it has to be mutably bound.

Outline

Tuples

One of the simplest types of aggregate data in Rust is a tuple.

```
let x: (i32, bool) = (7, true);
```

Tuples

One of the simplest types of aggregate data in Rust is a tuple.

```
let x: (i32, bool) = (7, true);
```

Which we can also destructure into its components via binding:

```
let (i, b) = x;
```

Tuples

One of the simplest types of aggregate data in Rust is a tuple.

```
let x: (i32, bool) = (7, true);
```

Which we can also destructure into its components via binding:

```
let (i, b) = x;
```

or accessed by position:

```
let y = x.0 + 3;
```

Tuples can have many distinct fields, which may themselves be of any type

```
let x = (1, 3e-7, false, "Hello!");
```

and can be returned from functions, or used as arguments

```
fn divmod(n: u32, k: u32) -> (u32, u32) {
    if n < k {
        (0, n)
    } else {
        let (q, d) = divmod(n, n - k);
        (q + 1, d)
    }
}
```

Arrays

Rust also has arrays, which provide for storage for many elements which have the same type. The size of an array must be statically known, and arrays cannot be resized. We write array types `[T; N]` for an N element array with element type T.

```
let x: [i32; 5] = [0, 1, 2, 3, 4];
let mut y: [i32; 100] = [0; 100];
```

Arrays

Rust also has arrays, which provide for storage for many elements which have the same type. The size of an array must be statically known, and arrays cannot be resized. We write array types `[T; N]` for an N element array with element type T.

```
let x: [i32; 5] = [0, 1, 2, 3, 4];
let mut y: [i32; 100] = [0; 100];
```

Accessing an element in the array is fairly standard:

```
y[0] = x[1] + x[3];
assert_eq!(y[0], 4);
```

What if we index out-of-bounds?

```
let mut x = [1, 2, 3];
x[4] = 7;
```

What if we index out-of-bounds?

```rust
let mut x = [1, 2, 3];
x[4] = 7;
```

Unlike C, there's no undefined behaviour here! Instead, the program will "panic"—there are some settings for exactly what this means, but by default you'll get a backtrace and the program will terminate.

```
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is 1', src/main.rs:4:5
stack backtrace:
   0: rust_begin_unwind
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/core/src/panicking.rs:107:14
   2: core::panicking::panic_bounds_check
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/core/src/panicking.rs:75:5
   3: playground::main
             at ./src/main.rs:4:5
   4: core::ops::function::FnOnce::call_once
             at /rustc/db9d1b20bba1968c1ec1fc49616d4742c1725b4b/library/core/src/ops/function.rs:227:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```c
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```c
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

- What if `x` is a null pointer?

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```c
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

- What if `x` is a null pointer?
- What if `x` doesn't point to `n` elements?

Slices

Often in C we might operate on an array by the use of a pointer to its initial element:

```
int sum(int *x, size_t n) {
    int sum = 0;
    for (size_t i = 0; i < n; ++i) {
        sum += x[i];
    }
    return sum;
}
```

This is error prone in several ways.

- What if `x` is a null pointer?
- What if `x` doesn't point to `n` elements?
- What if `x` is an otherwise invalid pointer?

We can avoid these issues by using a "slice" type in Rust.

`[T]` is an unsized type representing some contiguous sequence of elements of type `T`—this isn't very useful on its own, because we don't know how big it is!

We can avoid these issues by using a “slice” type in Rust.

`[T]` is an unsized type representing some contiguous sequence of elements of type `T`—this isn’t very useful on its own, because we don’t know how big it is!

Using a reference, we can get something we do know the size of:

- `&[T]` is the type of shared slices
- `&mut [T]` is the type of mutable/exclusive slices

Both of these will additionally store a length, along with a pointer to the start of the slice.

So if we want to sum an array in Rust, we might instead have:

```rust
fn sum(xs: &[i32]) -> i32 {
    let mut sum = 0;
    for x in xs {
        sum += x;
    }
    sum
}
```

So if we want to sum an array in Rust, we might instead have:

```
fn sum(xs: &[i32]) -> i32 {
    let mut sum = 0;
    for x in xs {
        sum += x;
    }
    sum
}
```

which we could use like so:

```
let x = [1, 2, 3, 4];

assert_eq!(sum(&x[ .. ]), 10);
assert_eq!(sum(&x[1.. ]),  9);
assert_eq!(sum(&x[ ..2]),  3);
```

Vec<T>

… but this is pretty restrictive. What if I want a dynamically sized array?

Vec<T>

… but this is pretty restrictive. What if I want a dynamically sized array?

```
// We can construct these like arrays, with the vec! macro
let mut x = vec![0, 1, 2, 3, 4];
let y = vec![0; 100];
```

Vec<T>

… but this is pretty restrictive. What if I want a dynamically sized array?

```
// We can construct these like arrays, with the vec! macro
let mut x = vec![0, 1, 2, 3, 4];
let y = vec![0; 100];
```

Because the sizing is dynamic, we can add to these:

```
x.push(5);
x.push(6);
assert_eq!(x.len(), 7);
assert!(match x.pop() { Some(6) => true, _ => false });
```

Next Class

- Pattern matching
- `impl` blocks
- Ownership, lifetimes, and the borrow system

Homework

Install Rust: https://rustup.rs/

- You can do this on your own machine
- You can also do work on the cluster machines
  - `unix.andrew.cmu.edu` has Rust 1.61.0 pre-installed.
  - (this is a couple versions behind, so there is a minor chance you run into warnings about new library features)

We also recommend setting up Rust Analyzer in whatever editor you prefer to use. If you use Jetbrains products then CLion also has good support.