Lecture 2

Introduction to Rust

Slides: Neil Kaushikkar

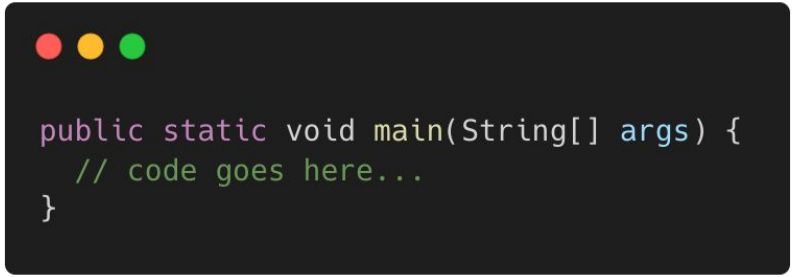
Goals For Today

- Anatomy of a Rust Program
- Variables and Mutability
- Basic Variable Types
- Macros, Typecasting, and Control Flow
- Standard Input/Output
- (Lecture code in video description)

The Main Function

```c
int main(int argc, char** argv) {
  // code goes here...
  return 0;
}
```

```rust
fn main() {
  // code goes here...
}
```



```java
public static void main(String[] args) {
  // code goes here...
}
```

Variables and Mutability

- The **let** keyword

```
fn main() {
  let x = 5;
}
```

```
fn main() {
  let x: u32 = 5;
}
```

Variables and Mutability

- The **let** keyword
- Rust will always try to guess your variable's type
- Best practice: always list variable types

```
fn main() {
  let x = 5;
}
```

```
fn main() {
  let x: u32 = 5;
}
```

Seems Easy Enough?

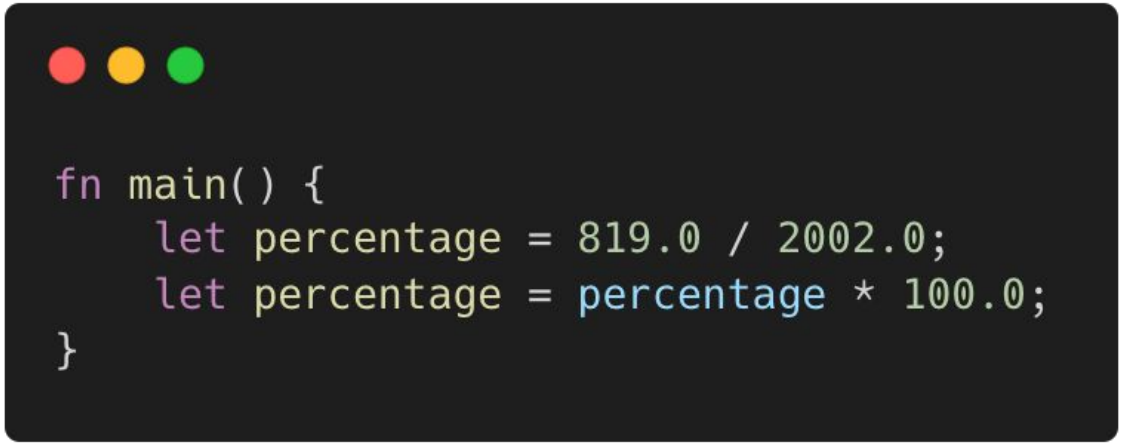
Variables and Mutability

- By default, all variables in Rust are **immutable**
- The **mut** keyword

```
fn main() {
    let mut x = 5;
    x += 1;
}
```

Shadowing

- Used when we want to change a variable a few times but then keep it constant
- Useful when we want to make an intermediate calculation



```
fn main() {
    let percentage = 819.0 / 2002.0;
    let percentage = percentage * 100.0;
}
```

Shadowing

- DANGER: the variable type can change between **let** calls
- Best practice: specify variable types for *code readability*

```
fn main() {
    let spaces = "       ";
    let spaces = spaces.len();
}
```

```
fn main() {
    let spaces: &str = "       ";
    let spaces: usize = spaces.len();
}
```

Integers

Length	Signed	Unsigned
8 bits	i8	u8
16 bits	i16	u16
32 bits	**i32 (default)**	u32
64 bits	i64	u64
128 bits	i128	u128
Size type	isize	usize

Arithmetic and Boolean Operators

- All the arithmetic operations you know and love: +, - , *, /, %, etc.
- Rust also uses the arithmetic assignment operators: +=, -=, *=, etc.
- All the boolean expressions you're familiar with: !, >, <, ==, etc.
- Rust **DOES NOT** support increment operators: ++, --

Char, Bool, Float

- **char** - represents a single character, declared using single quotes
- **bool** - either **true** or **false**
- **f32** and **f64** (default) - floating point types

```rust
fn main() {
  let ferris: char = '🦀';
  let is_rust_the_best: bool = true;
  let version: f64 = 2.3;
  let previous: f32 = 2.2;
}
```

Typecasting

- Try and interpret some value **AS** another type
- The **as** keyword

```
fn main() {
  let my_double: f32 = 0.0;
  let my_double = 0.0f32;
  let my_double = 0.0 as f32;

  let upcast = my_double as f64;
}
```

Useful Typecasting

The Unit Type

- Represented by **()**
- Has exactly one value: itself
- Used when there is no other meaningful value that could be returned
- Most common location: return type of functions that don't return anything
- **void** in C/C++ and Java

```
fn main() {
  // code here
}
```

```
fn main() -> () {
  // code here
}
```

Comments

- Inline comments (2 slashes)
- Multi-line comments (/* */)
- Doc comments (3 slashes)

```rust
/// This is a doc comment
/// notice the 3 slashes!
fn main() {
    // this is an inline comment
    println!("Hello there!");

    /*
     * This is
     * a multi-line
     * comment.
     */
}
```

Strings

- **String** - the most common string type
- **&str** - the most primitive string type
- The difference: **String** has **ownership** over its characters

Macros

- Macros are pieces of code that **write other code** for you (the programmer)
- A function signature in Rust need to declare the total number of parameters and the type of each parameter
- Macros, on the other hand, can take a variable number of parameters

Macros & Standard Output

- **println!(<format string>, arg1, arg2...)** - print some formatted text
- **print!(<format string>, arg1, arg2...)** - print formatted text (no newline)
- **format!(<format string>, arg1, arg2...)** - create a formatted **String**

```rust
fn main() {
  println!("Hello World!");
  print!("No newline after me...");

  let fav_num: u64 = 2002;
  let s: String = format!("My favorite number is {}!", fav_num);
}
```

Creating Strings

Control Flow

- Same as if many other languages you are familiar with: **if**, **else if**, **else**
- No parentheses required (!!!)

```
fn main() {
    let my_bool = true;

    if my_bool {
        println!("CS 128 Honors");
    }
}
```

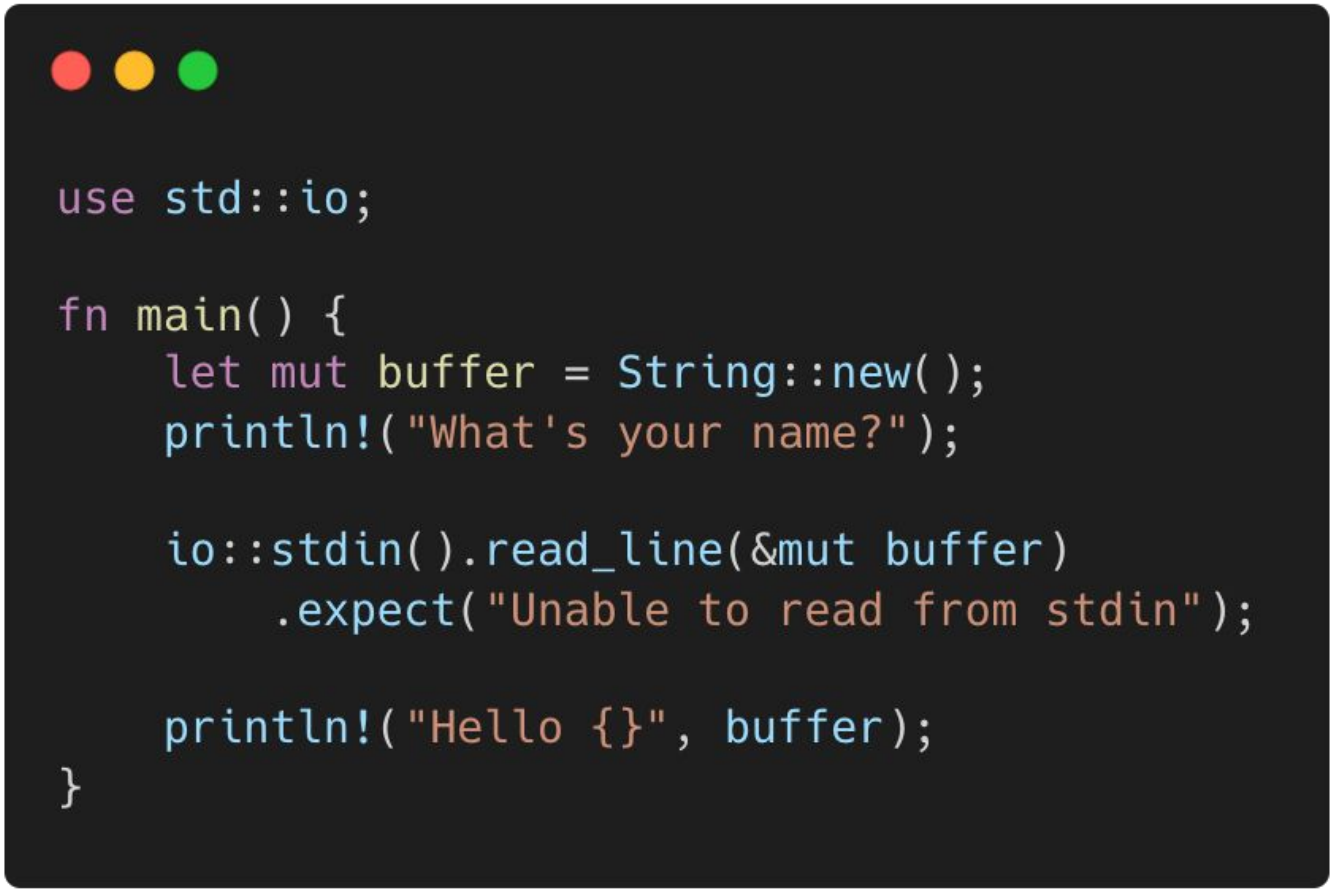
```
fn main() {
    let x = -5;

    if x < 0 {
        println!("x = {} is a negative number", x);
    } else if x > 0 {
        println!("x = {} is a positive number", x);
    } else {
        println!("x = 0");
    }
}
```

Syntactic Sugar: Control Flow

- Variable assignments **FROM** an **if**/**else if**/**else** statement
- **NOTICE:** no semicolon **AFTER** the return but there is **AFTER** the curly braces!!
- **IMPORTANT:** return types of each branch **MUST** match

```
fn main() {
    let x; i32 = -5;
    let y: i32 = if x < 0 {
    -x
  } else {
    x
  };

  println!("The absolute value of {} is {}", x, y);
}
```

Reading from Standard Input

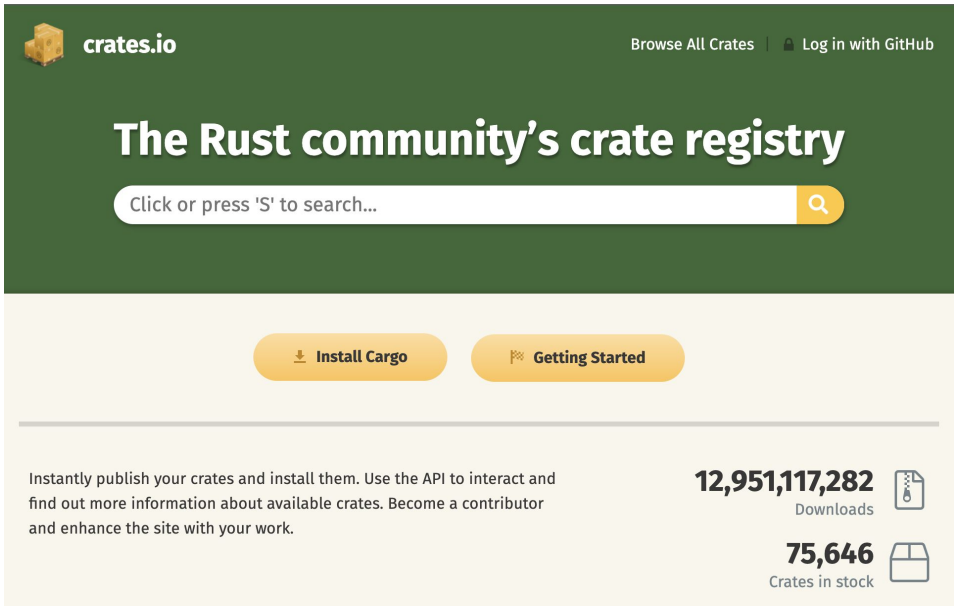


```rust
use std::io;

fn main() {
    let mut buffer = String::new();
    println!("What's your name?");

    io::stdin().read_line(&mut buffer)
        .expect("Unable to read from stdin");

    println!("Hello {}", buffer);
}
```

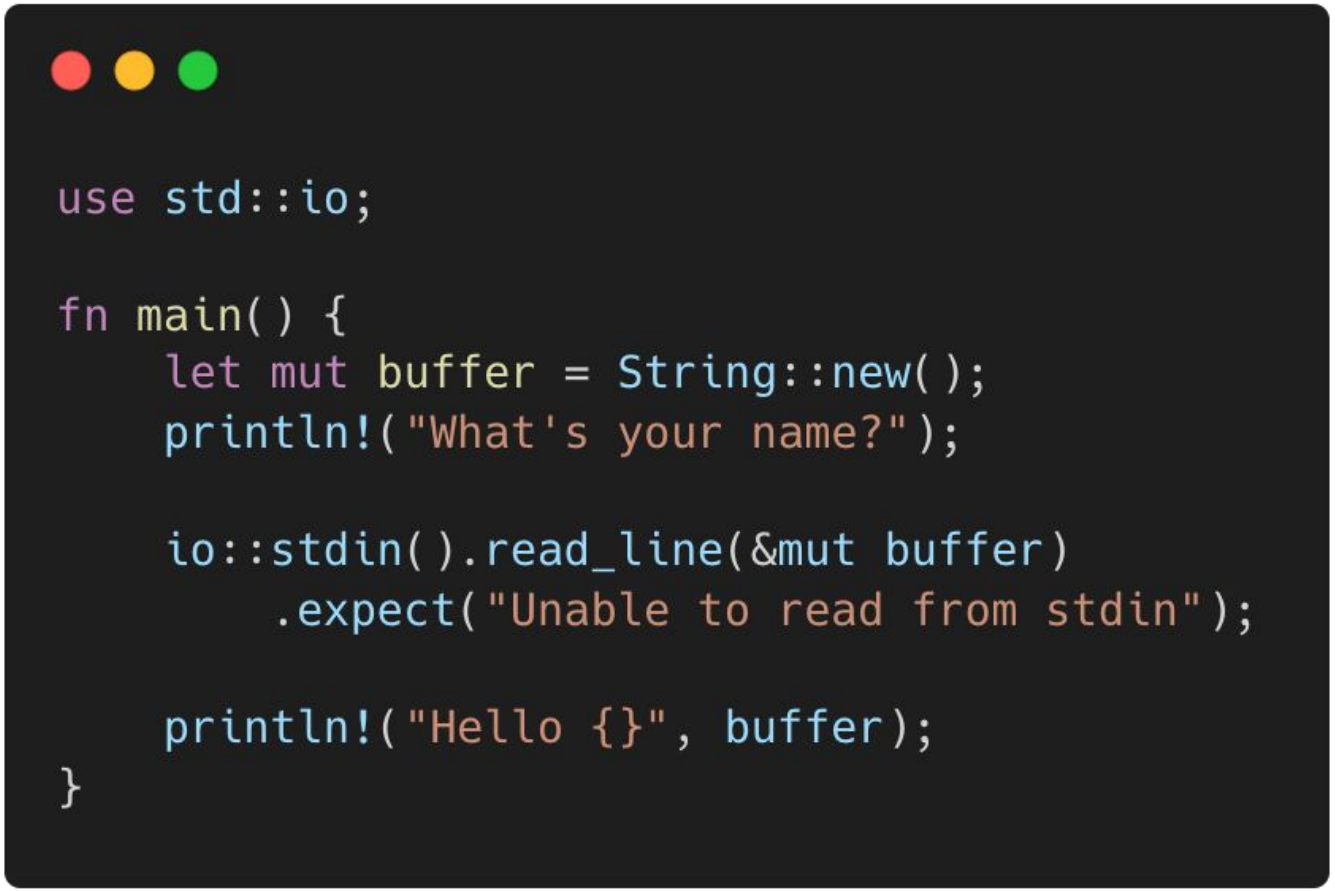
What are Crates?

What are Crates (in Rust)?

- A crate is a binary or library
- Import code from crates with the **use** keyword
- Most commonly used crate: **std** library

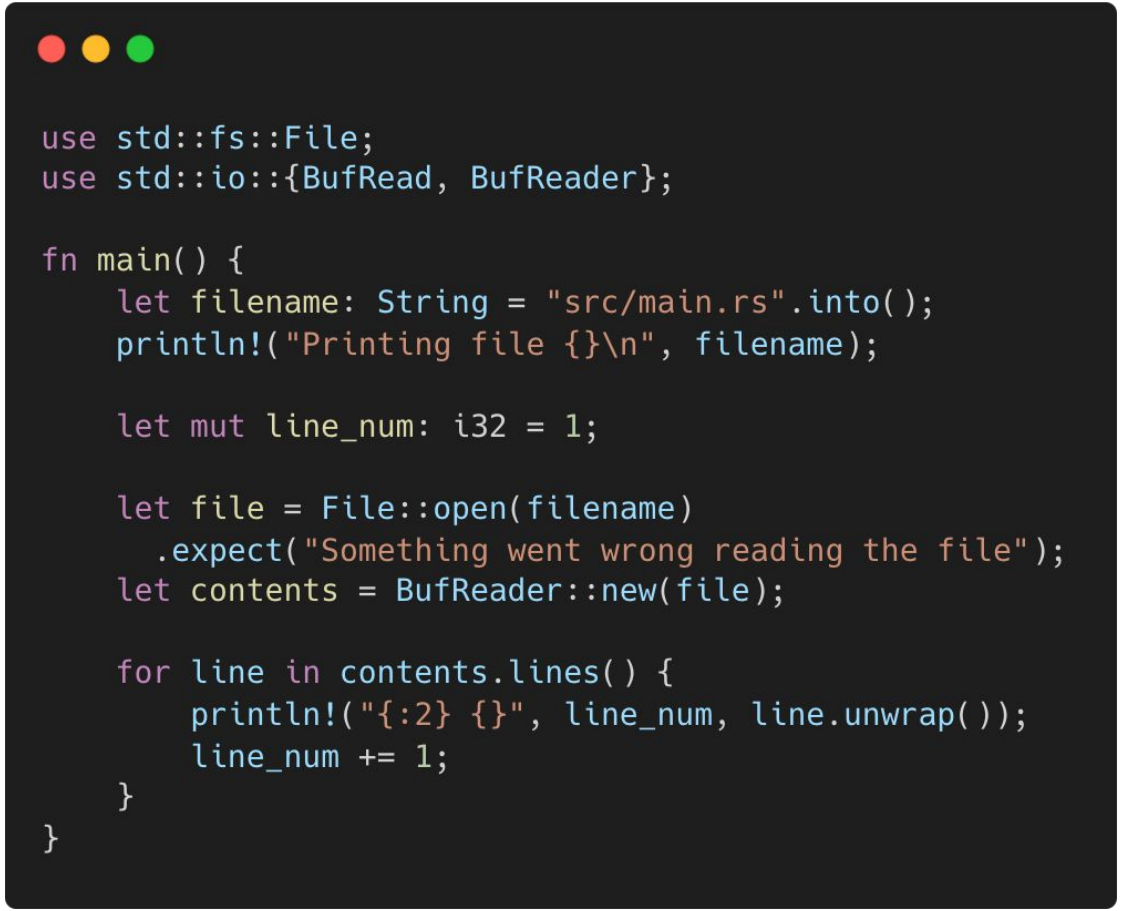


Reading from Standard Input



```rust
use std::io;

fn main() {
    let mut buffer = String::new();
    println!("What's your name?");

    io::stdin().read_line(&mut buffer)
        .expect("Unable to read from stdin");

    println!("Hello {}", buffer);
}
```

Reading from Files



```rust
use std::fs::File;
use std::io::{BufRead, BufReader};

fn main() {
    let filename: String = "src/main.rs".into();
    println!("Printing file {}\n", filename);

    let mut line_num: i32 = 1;

    let file = File::open(filename)
      .expect("Something went wrong reading the file");
    let contents = BufReader::new(file);

    for line in contents.lines() {
        println!("{:2} {}", line_num, line.unwrap());
        line_num += 1;
    }
}
```

Self-Practice Ideas

- Make a number guessing game
- Make an ad-lib program
  - Extra: write the output to a file!
- Anything else you can think of!