Rust Basics

Goals For Today

- Reminders & Announcements
- Arrays, Tuples, Indexing
- Functions
- Matching
- Result/Option
- Vectors and Vec!
- Homework

Following Along?
This lecture follows topics from Chapters 3, 6, and 8 of The Rust Textbook

Course Announcements

- Homework 1 Releases tonight.
  - We'll review it together at the end of today's lecture
- MP0 will release on Thursday
  - You'll review it together at the end of Thursday's lecture

Course Reminders

Are you:

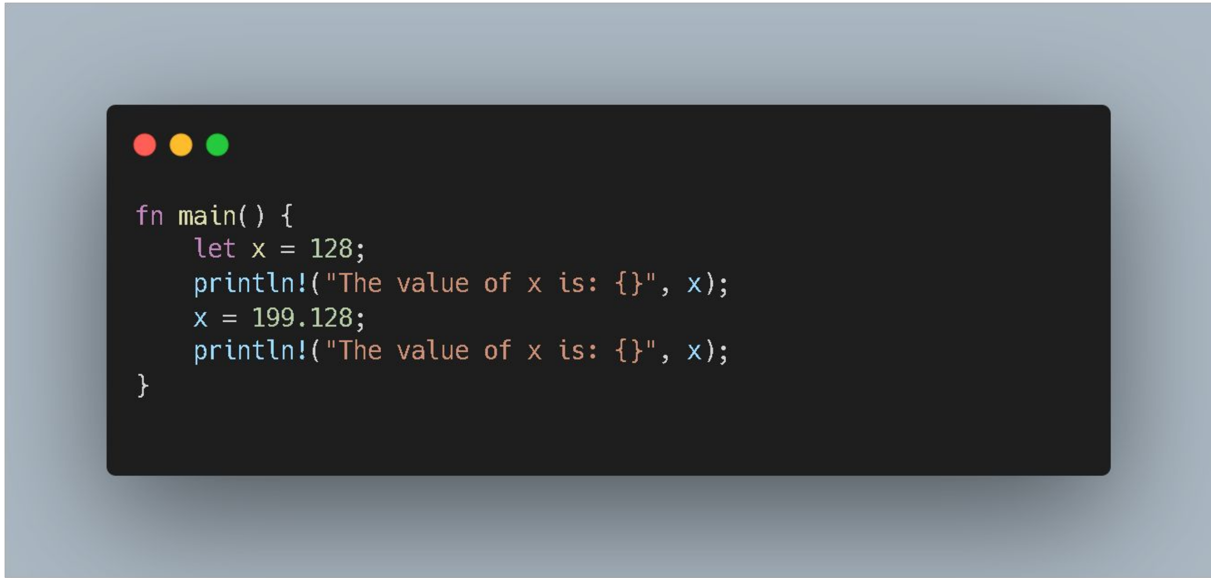
- In the Discord?
- In the PrairieLearn?

Homeworks and MPs will follow a somewhat regular release schedule.

- In general, Homeworks are released after lecture and are due in 1 week.
- MPs are released after Lecture and are due in >1 week (on a Wed or Fri)
- We are more than happy to grant extensions when requested, but we do require that you've made some progress on the HW or MP (unless you have some valid excuse)
- Homeworks will have a "Feedback Survey"- put whatever!

Review

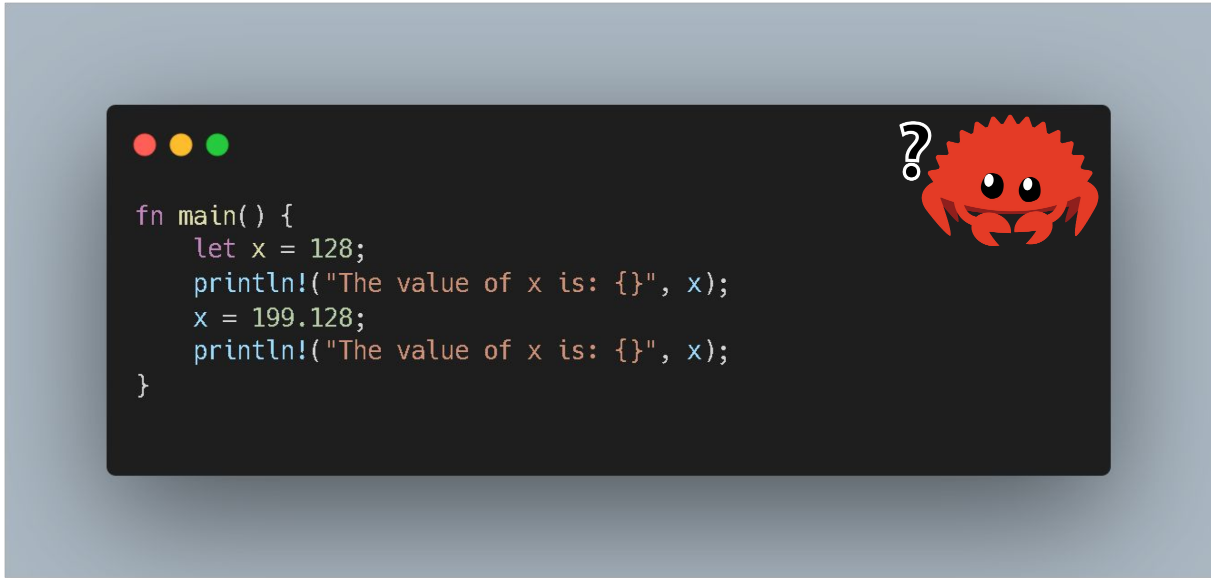
- Variables & Mutability



```rust
fn main() {
    let x = 128;
    println!("The value of x is: {}", x);
    x = 199.128;
    println!("The value of x is: {}", x);
}
```

Review

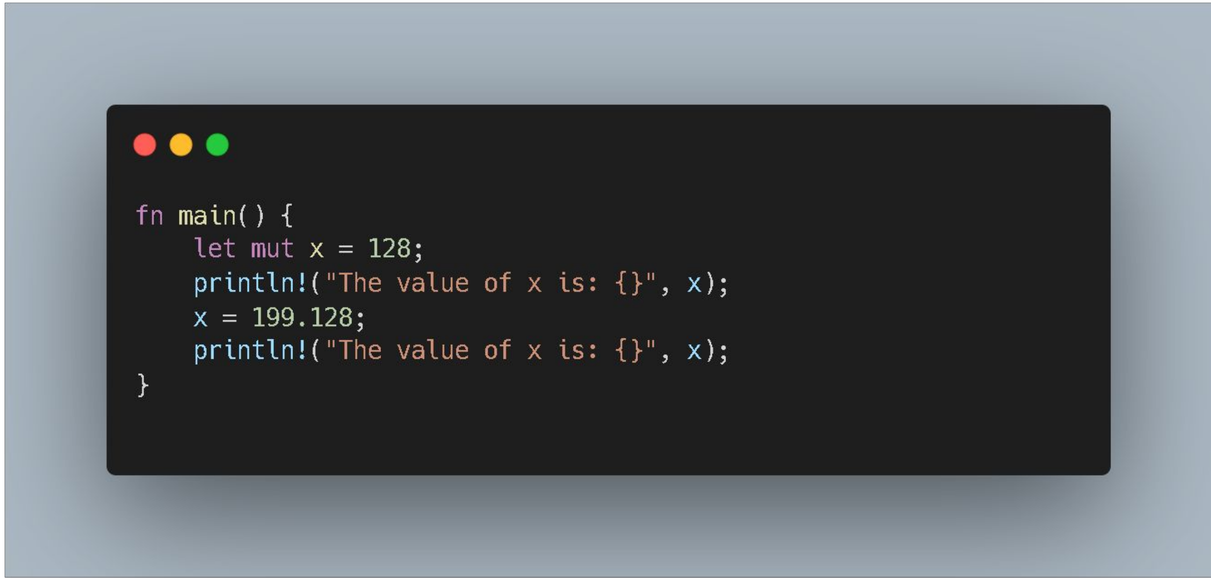
- Variables & Mutability



```
fn main() {
    let x = 128;
    println!("The value of x is: {}", x);
    x = 199.128;
    println!("The value of x is: {}", x);
}
```

Review

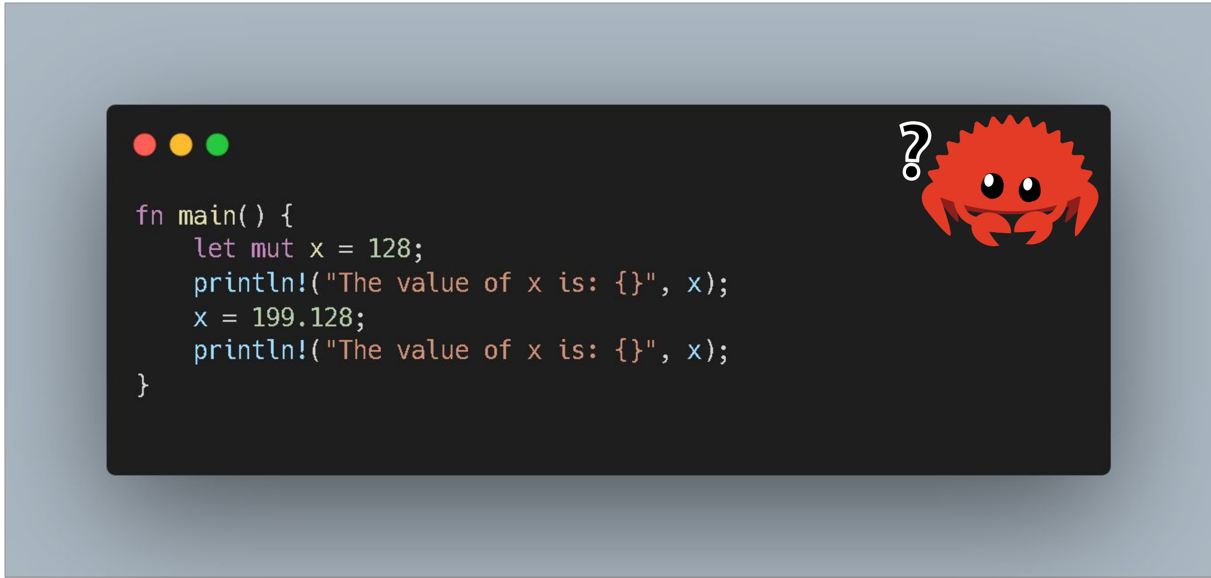
- Variables & Mutability



```
fn main() {
    let mut x = 128;
    println!("The value of x is: {}", x);
    x = 199.128;
    println!("The value of x is: {}", x);
}
```

Review

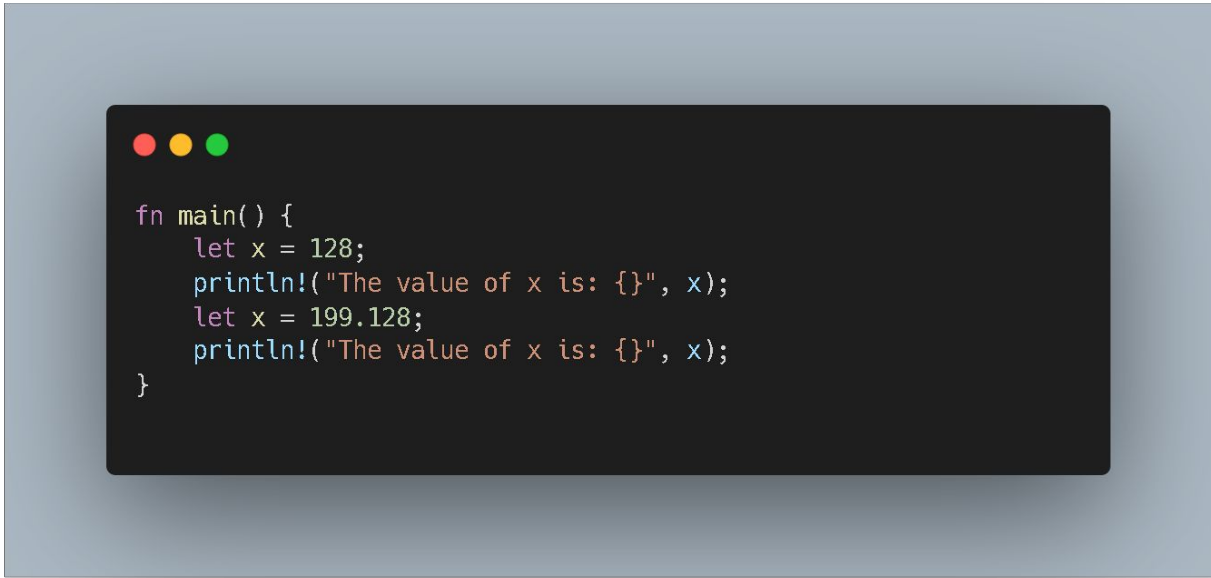
- Variables & Mutability



```
fn main() {
    let mut x = 128;
    println!("The value of x is: {}", x);
    x = 199.128;
    println!("The value of x is: {}", x);
}
```

Review

- Variables & Mutability
    - Shadowing



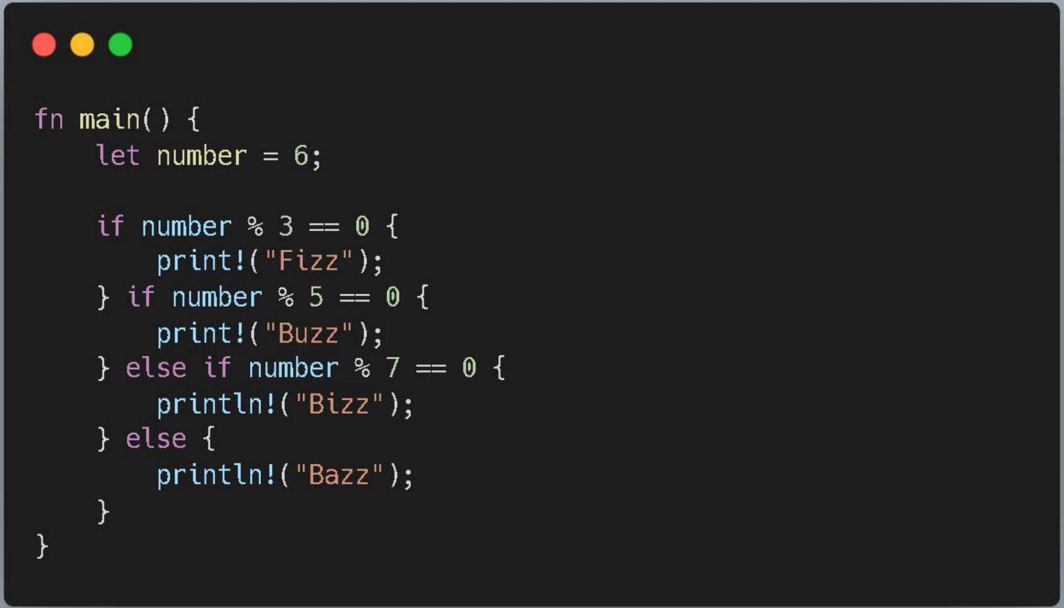
```
fn main() {
    let x = 128;
    println!("The value of x is: {}", x);
    let x = 199.128;
    println!("The value of x is: {}", x);
}
```

Review

- Variables & Mutability
  - Shadowing
- Data Types

Review

- Variables & Mutability
  - Shadowing
- Data Types
- Control Flow



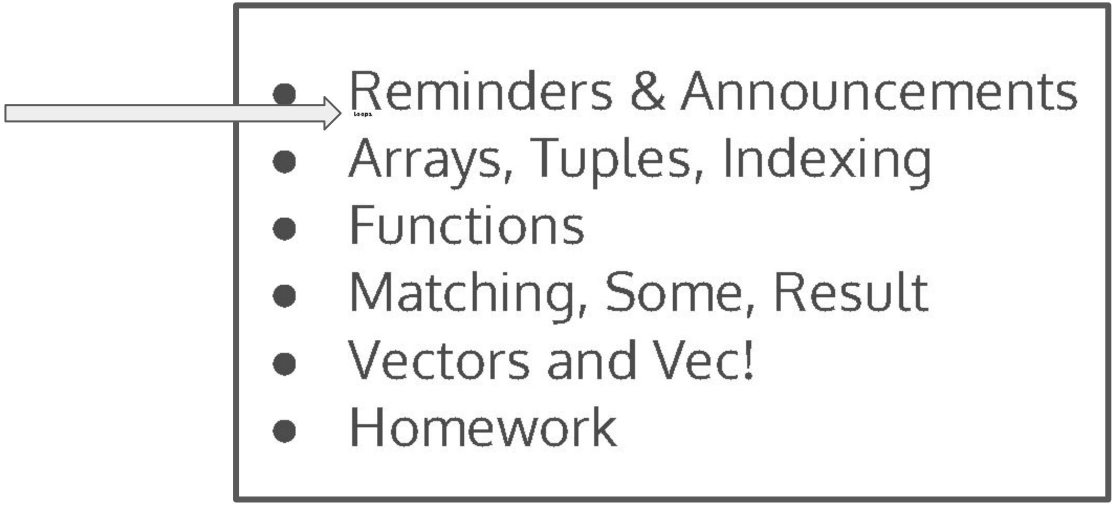
```rust
fn main() {
    let number = 6;

    if number % 3 == 0 {
        print!("Fizz");
    } if number % 5 == 0 {
        print!("Buzz");
    } else if number % 7 == 0 {
        println!("Bizz");
    } else {
        println!("Bazz");
    }
}
```

More Powerful Control Flow

- Loops
  - Returning from Loops
  - While

More Powerful Control Flow

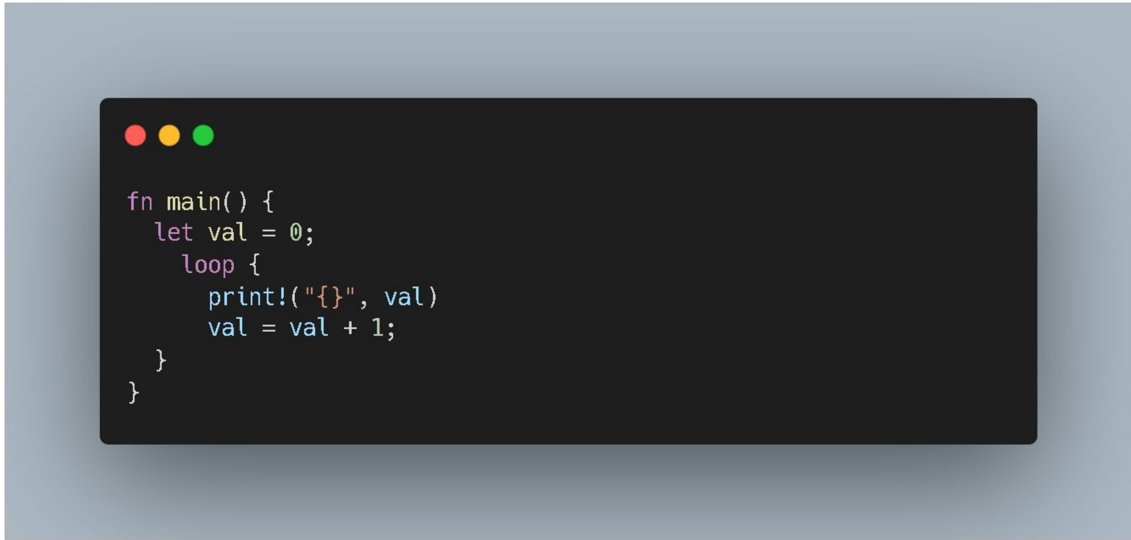
- Loops
  - Returning from Loops
  - While



- Reminders & Announcements
- Arrays, Tuples, Indexing
- Functions
- Matching, Some, Result
- Vectors and Vec!
- Homework

More Powerful Control Flow

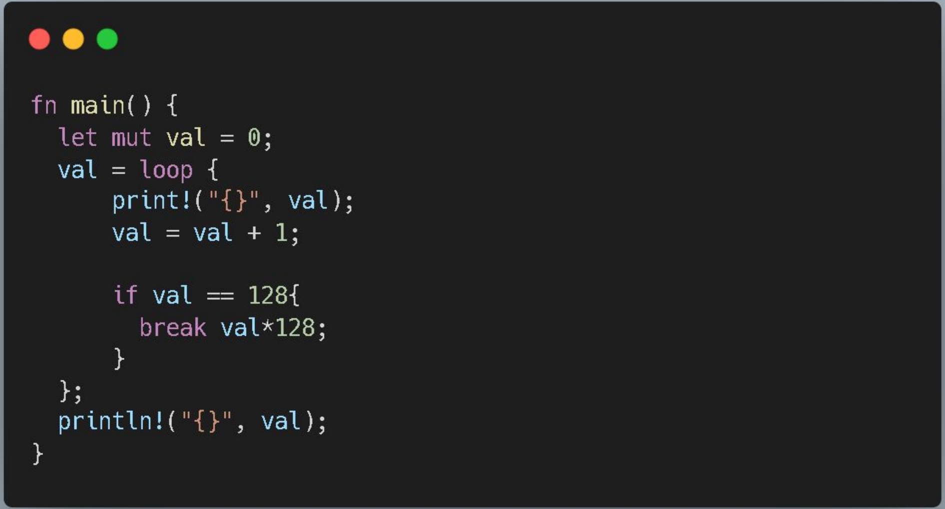
- Loops
  - Returning from Loops
  - While



```
fn main() {
  let val = 0;
    loop {
      print!("{}", val)
      val = val + 1;
  }
}
```

More Powerful Control Flow

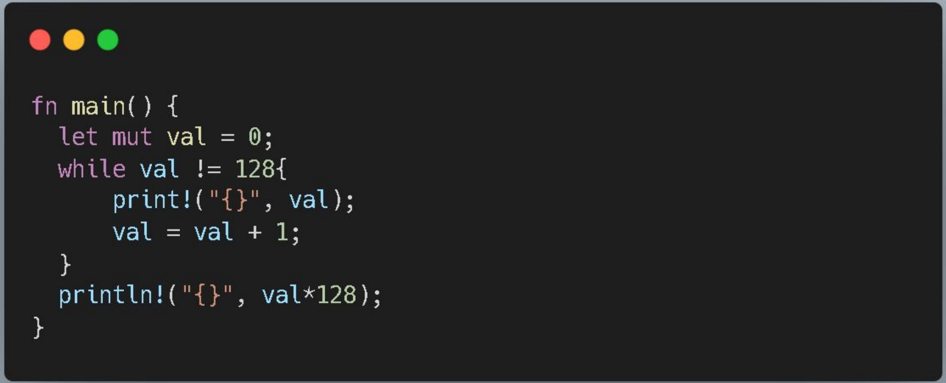
- Loops
  - Returning from Loops
  - While



```
fn main() {
  let mut val = 0;
  val = loop {
      print!("{}", val);
      val = val + 1;

      if val == 128{
        break val*128;
      }
  };
  println!("{}", val);
}
```

More Powerful Control Flow

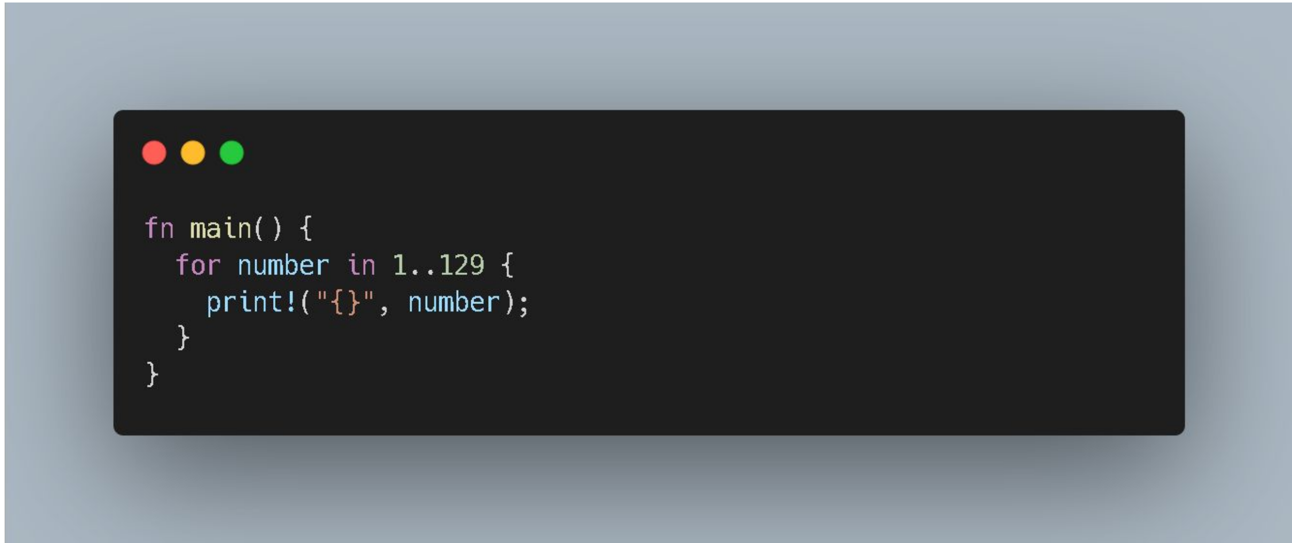
- Loops
  - Returning from Loops
  - While



```
fn main() {
  let mut val = 0;
  while val != 128{
      print!("{}", val);
      val = val + 1;
  }
  println!("{}", val*128);
}
```

More Powerful Control Flow

- Loops
  - Returning from Loops
  - While
  - For

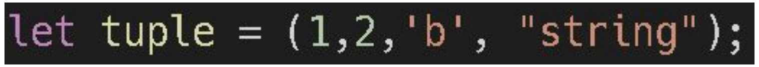


```
fn main() {
  for number in 1..129 {
    print!("{}", number);
  }
}
```

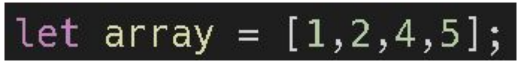
Arrays, Tuples, Indexing

AKA Compound types

- There are two primitive compound types
  - Tuples and Arrays



```
let tuple = (1,2,'b', "string");
```

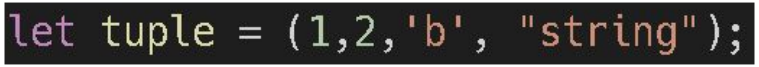


```
let array = [1,2,4,5];
```

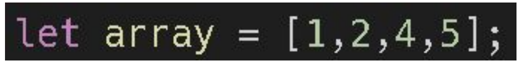
Arrays, Tuples, Indexing

AKA Compound types

- There are two primitive compound types
  - Tuples and Arrays



```
let tuple = (1,2,'b', "string");
```



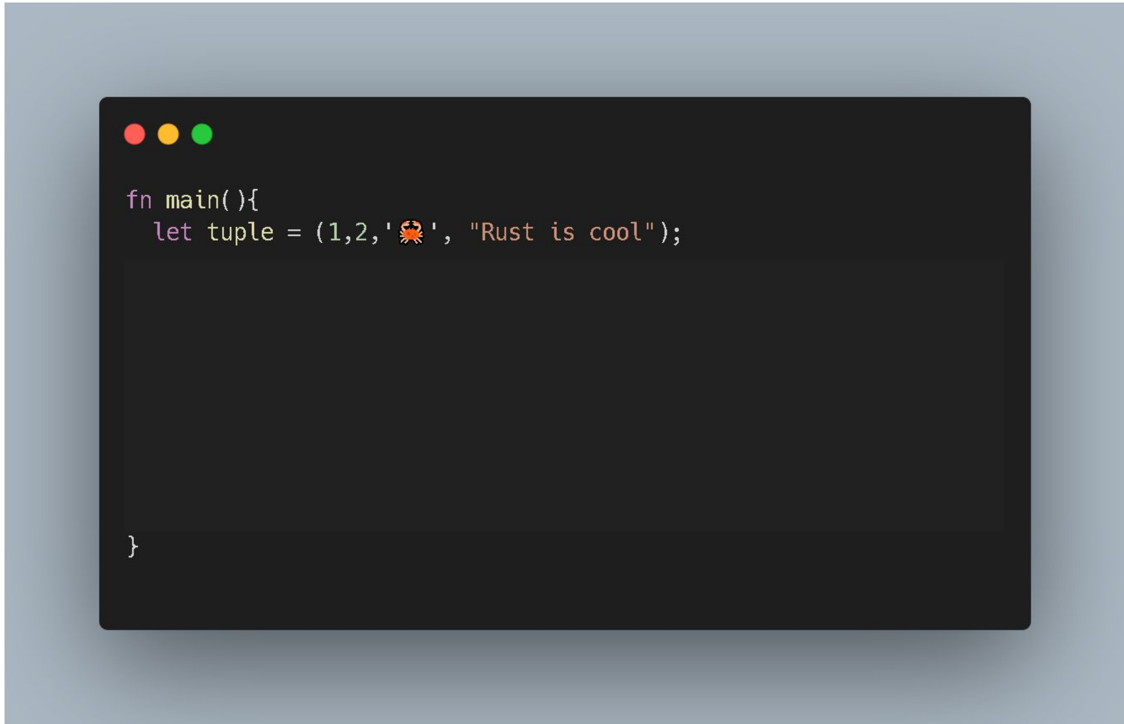
```
let array = [1,2,4,5];
```

Fixed Length	Fixed Length
(potentially) Multiple Types	Same type

Arrays, Tuples, Indexing

Tuples

- **Instantiating**

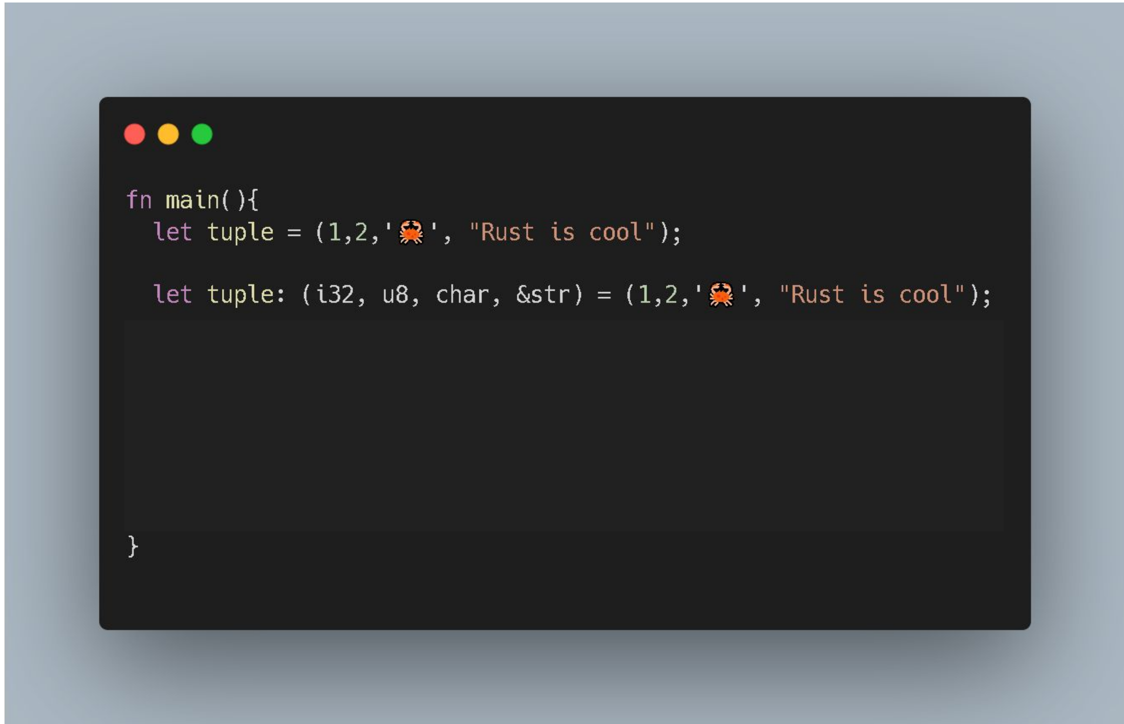


```
fn main(){
  let tuple = (1,2,'🦀', "Rust is cool");
}
```

Arrays, Tuples, Indexing

Tuples

- Instantiating
- **Typing**

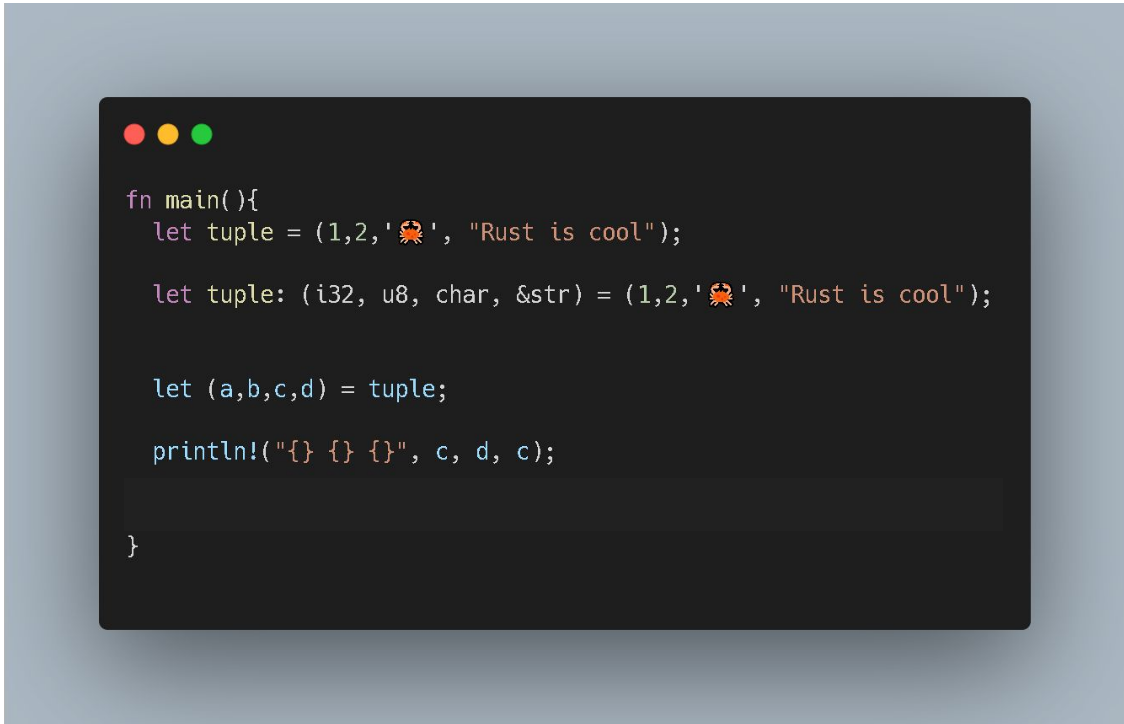


```
fn main(){
  let tuple = (1,2,'🦀', "Rust is cool");

  let tuple: (i32, u8, char, &str) = (1,2,'🦀', "Rust is cool");




}
```

Arrays, Tuples, Indexing

Tuples

- Instantiating
- Typing
- **Destructuring**

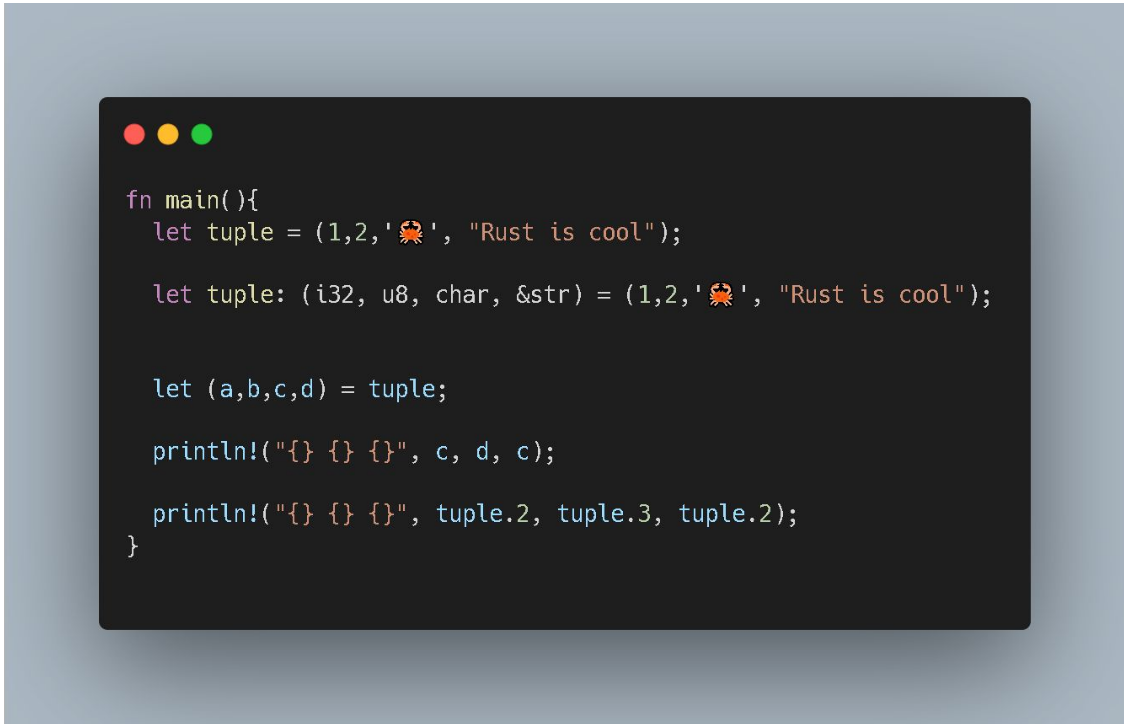


```rust
fn main(){
  let tuple = (1,2,'🦀', "Rust is cool");

  let tuple: (i32, u8, char, &str) = (1,2,'🦀', "Rust is cool");


  let (a,b,c,d) = tuple;

  println!("{} {} {}", c, d, c);


}
```

Arrays, Tuples, Indexing

Tuples

- Instantiating
- Typing
- Destructuring
- **Indexing**

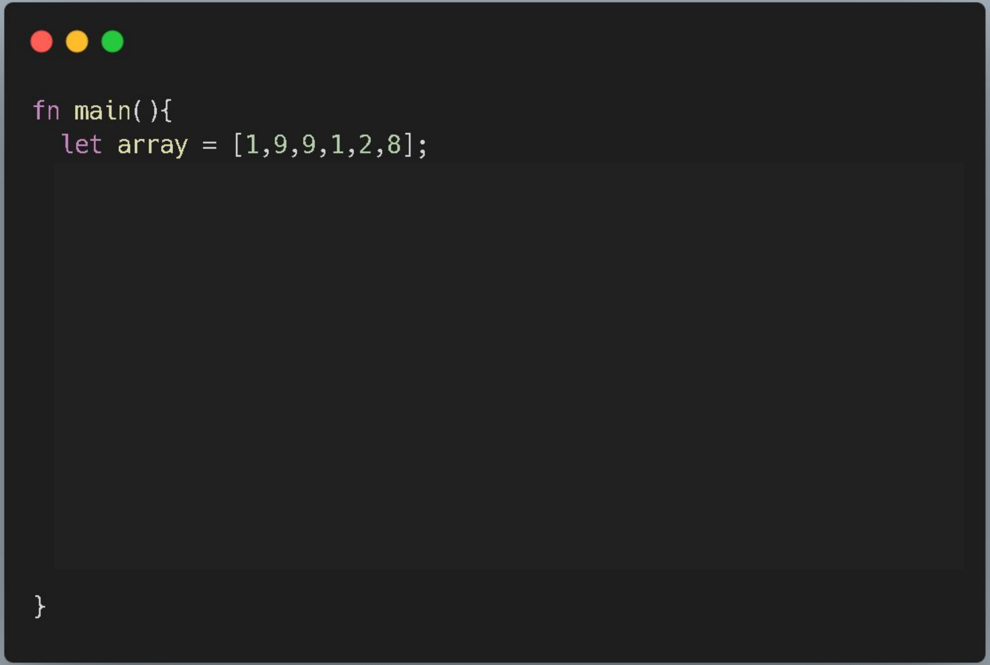


```
fn main(){
  let tuple = (1,2,'🦀', "Rust is cool");

  let tuple: (i32, u8, char, &str) = (1,2,'🦀', "Rust is cool");


  let (a,b,c,d) = tuple;

  println!("{} {} {}", c, d, c);

  println!("{} {} {}", tuple.2, tuple.3, tuple.2);
}
```

Arrays, Tuples, Indexing

Arrays

- **Instantiating**
- Typing
- Indexing/Assigning

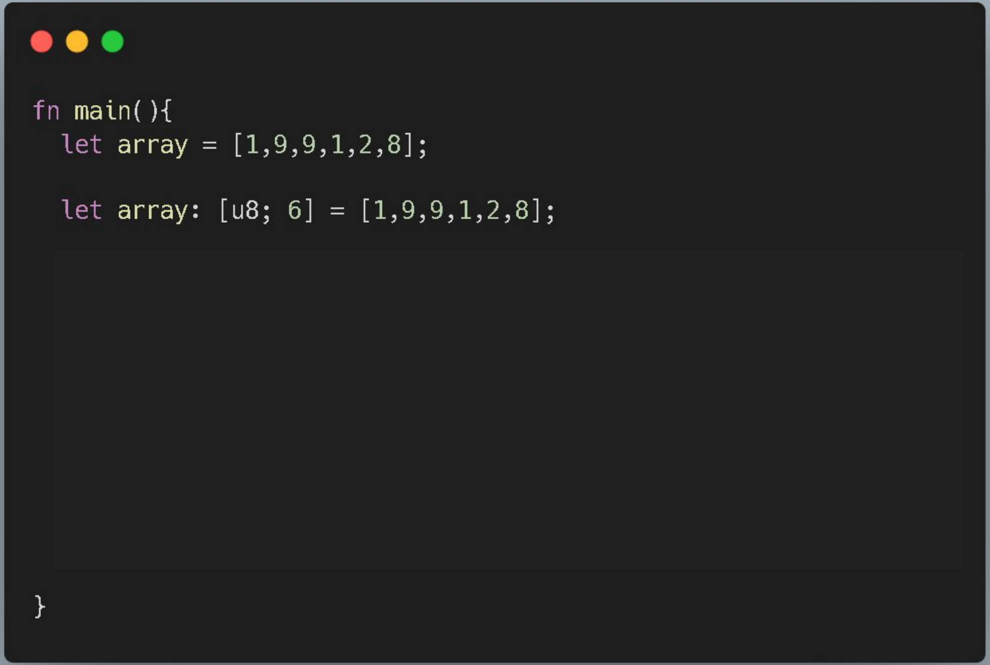


```
fn main(){
  let array = [1,9,9,1,2,8];



}
```

Arrays, Tuples, Indexing

Arrays

- Instantiating
- **Typing**
- Indexing/Assigning

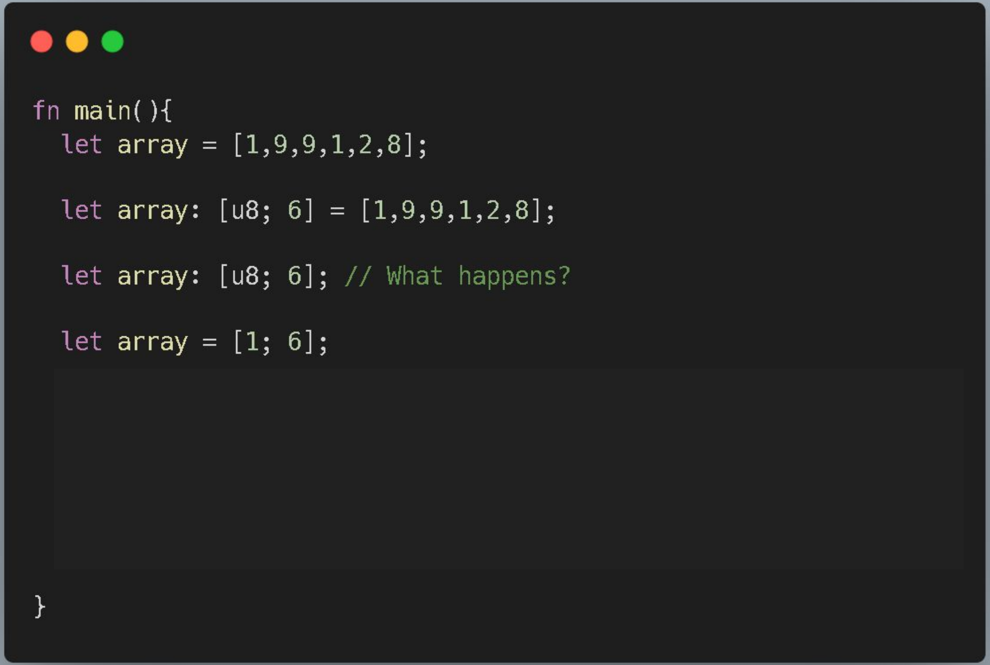


```
fn main(){
  let array = [1,9,9,1,2,8];

  let array: [u8; 6] = [1,9,9,1,2,8];









}
```

Arrays, Tuples, Indexing

Arrays

- Instantiating
- **Typing**
- Indexing/Assigning

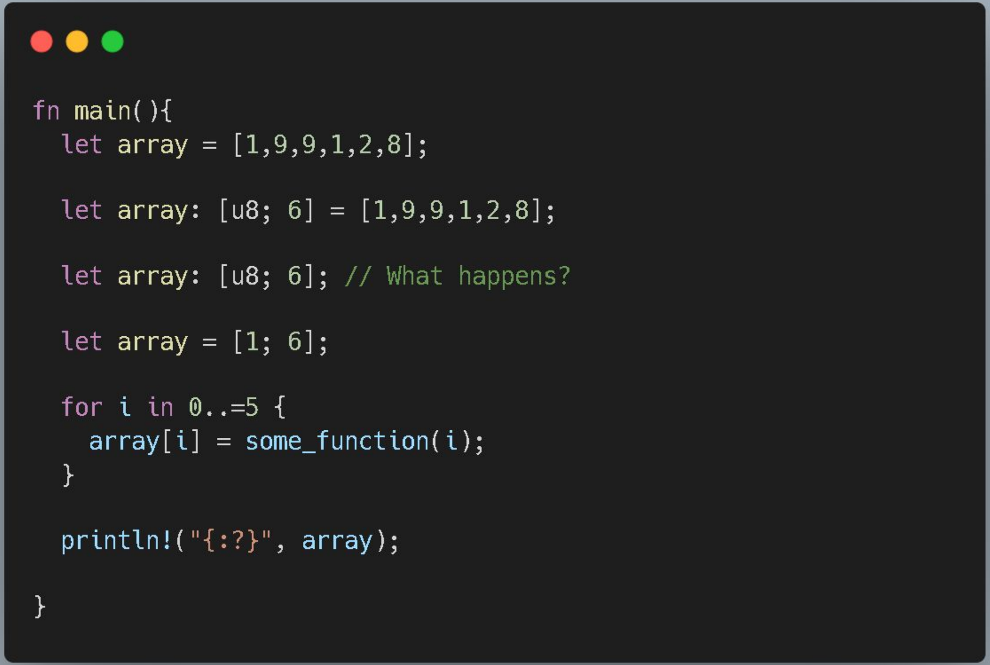


```
fn main(){
  let array = [1,9,9,1,2,8];

  let array: [u8; 6] = [1,9,9,1,2,8];

  let array: [u8; 6]; // What happens?

  let array = [1; 6];




}
```

Arrays, Tuples, Indexing

Arrays

- Instantiating
- Typing
- **Indexing/Assigning**

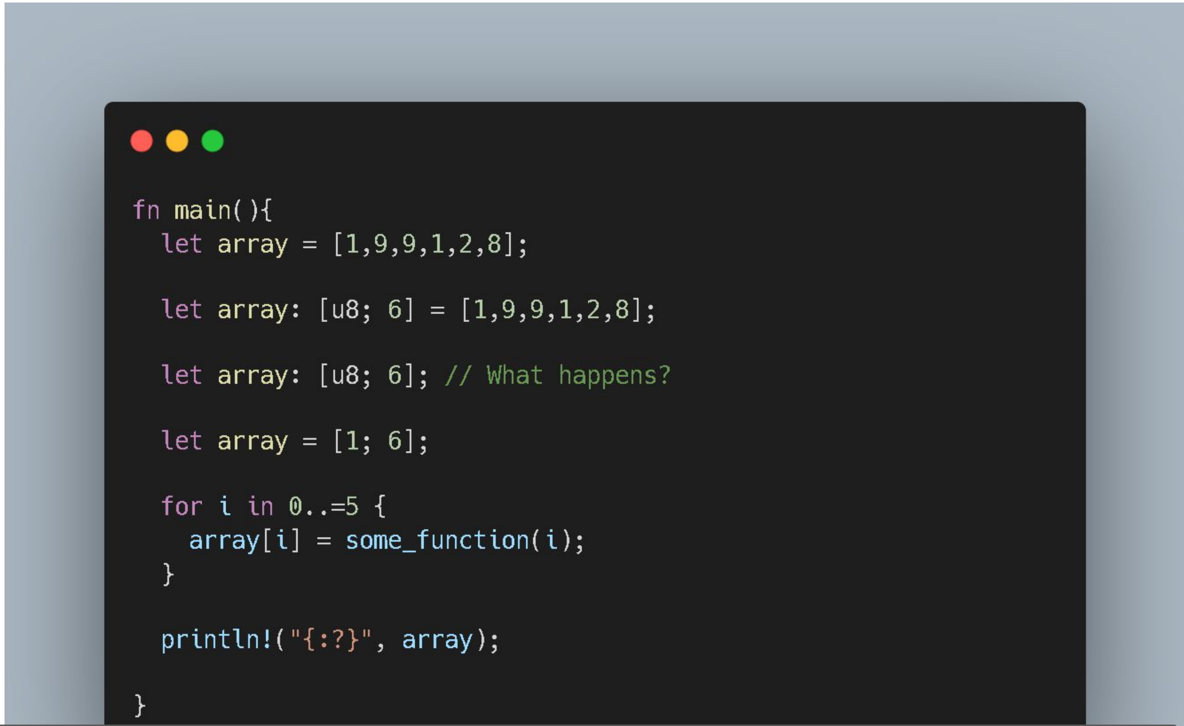


```rust
fn main(){
  let array = [1,9,9,1,2,8];

  let array: [u8; 6] = [1,9,9,1,2,8];

  let array: [u8; 6]; // What happens?

  let array = [1; 6];

  for i in 0..=5 {
    array[i] = some_function(i);
  }

  println!("{:?}", array);

}
```

Arrays, Tuples, Indexing

Arrays

- Instantiating
- Typing
- **Indexing/Assigning**



```rust
fn main(){
  let array = [1,9,9,1,2,8];

  let array: [u8; 6] = [1,9,9,1,2,8];

  let array: [u8; 6]; // What happens?

  let array = [1; 6];

  for i in 0..=5 {
    array[i] = some_function(i);
  }

  println!("{:?}", array);

}
```

Standard Output

```
[1, 9, 9, 1, 2, 8]
```

Functions

Live demo :)

Functions

Statements vs Expressions

- **Statements** are instructions that perform an action and do not return a value
- **Expressions** evaluate to some value and return that value

Match Statements

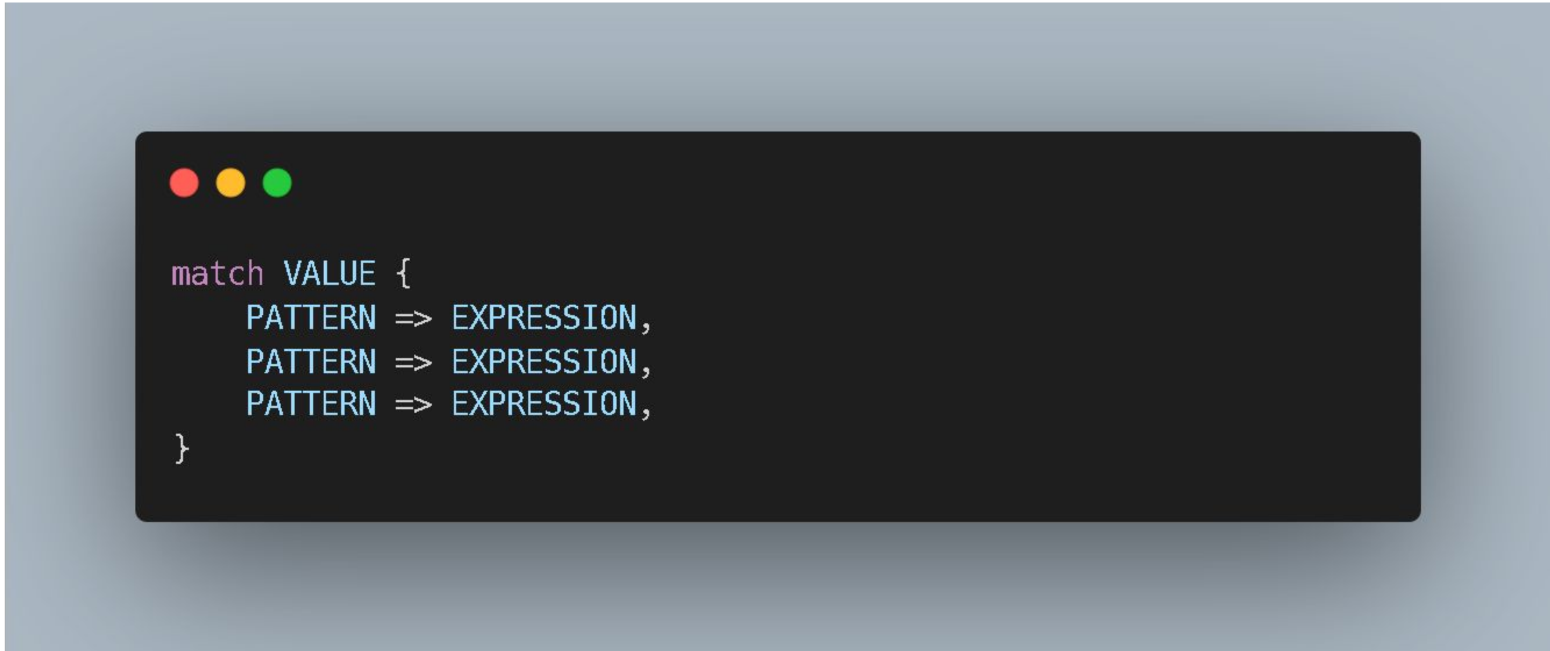
You'll learn much more about these topics (Match, Enums, Result/Option, and how they all interact) on Thursday, but we want to introduce them now as a warm-up

Rust has a powerful control flow operator called **`match`**

- You can compare some value to a series of patterns, then execute some code based on which pattern matches

Match Statements

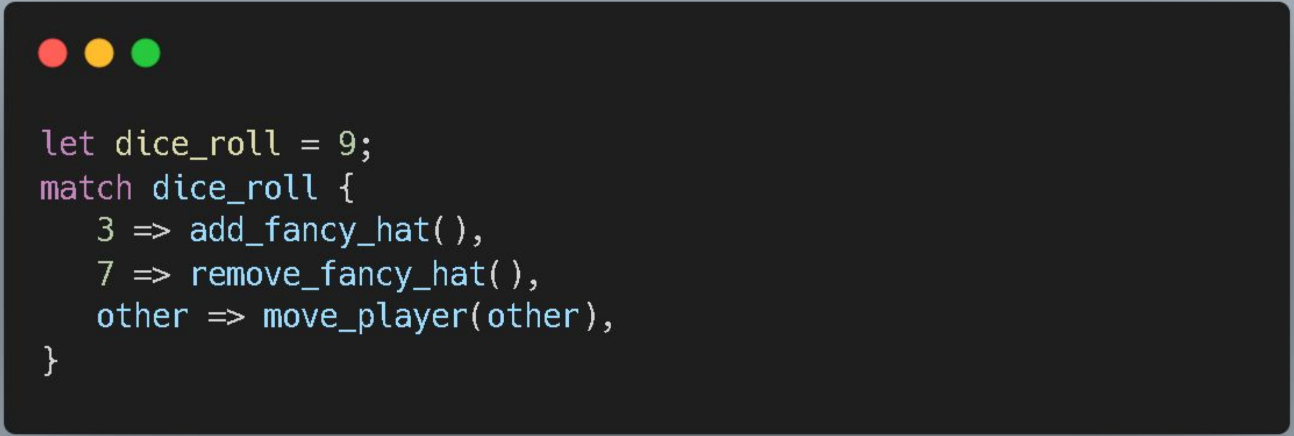
- The patterns must be **exhaustive**
    - Patterns appear in many places in Rust. It's very useful to understand how they work



```
match VALUE {
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
    PATTERN => EXPRESSION,
}
```

Match Statements

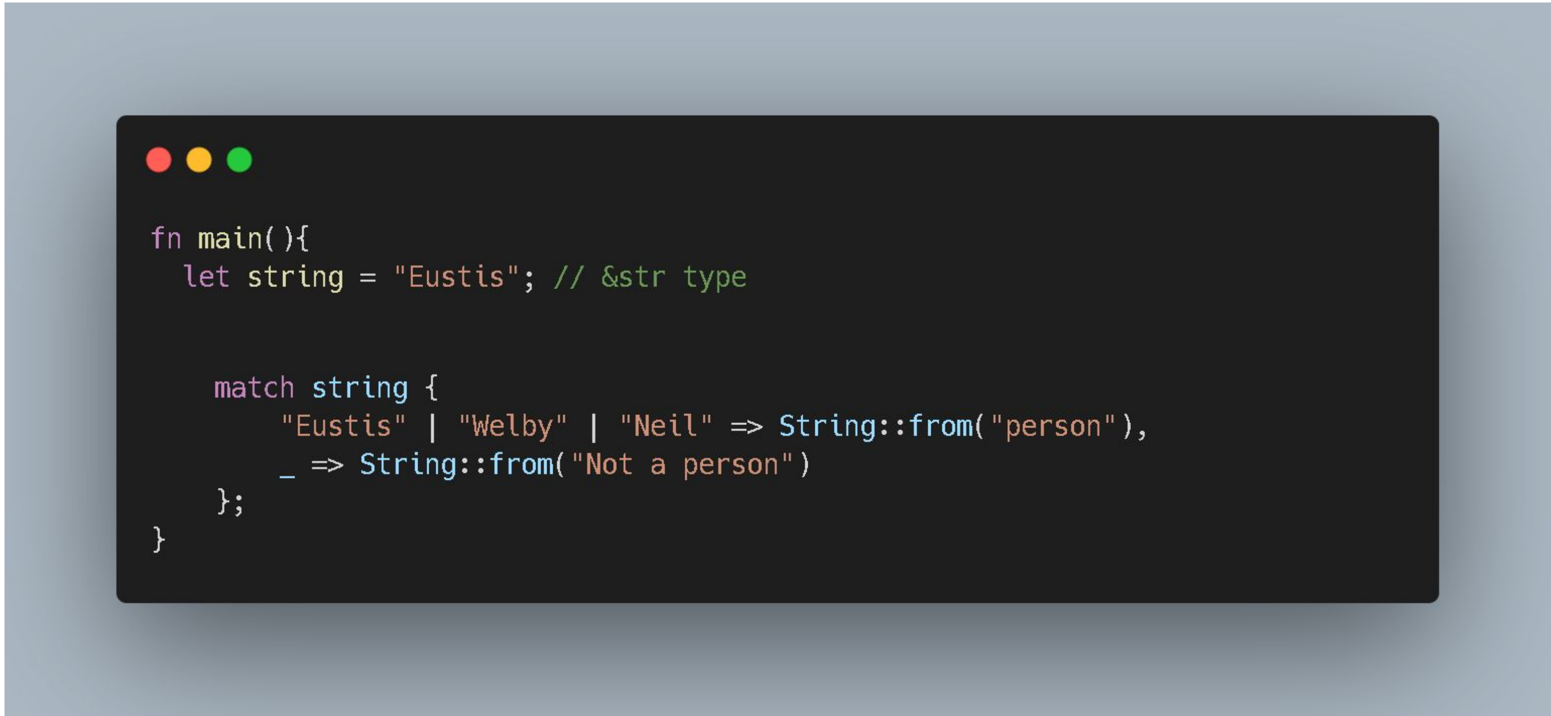
- The patterns must be **exhaustive**
  - Patterns appear in many places in Rust. It's very useful to understand how they work



```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}
```

Match Statements

- The patterns must be **exhaustive**
  - Patterns appear in many places in Rust. It's very useful to understand how they work



```rust
fn main(){
  let string = "Eustis"; // &str type


    match string {
        "Eustis" | "Welby" | "Neil" => String::from("person"),
        _ => String::from("Not a person")
    };
}
```

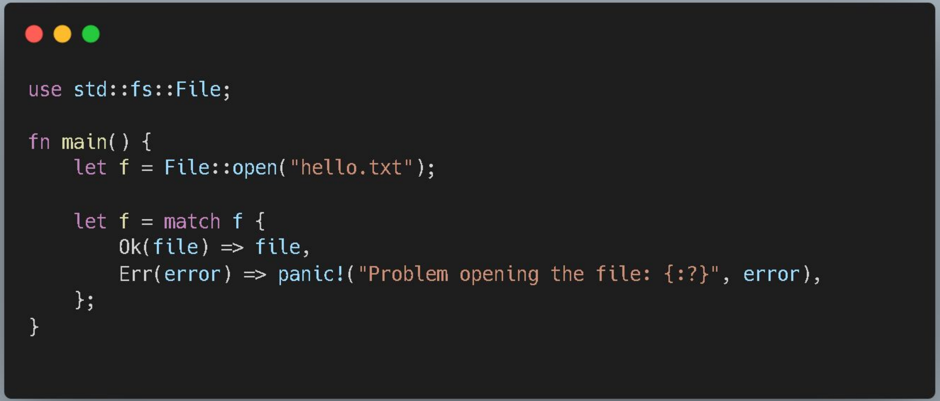
Result/Option

- We commonly use **enums** with match statements
    - Enums allow you to define a **type** by enumerating it's possible **variations**
- There are two special **enums** we want to introduce early
    - **Result**
    - **Option**

Result/Option

The **Result** enum represents the success (or failure) of some operation.

- You want to open some file but the file doesn't exist
  - Instead of crashing, we can return the fact that the result of our operation was an Error
  - Then, maybe we can create the file instead of terminating the operation
- There are two cases for the **Result** enum
  - **Ok(T)**
  - **Err(E)**

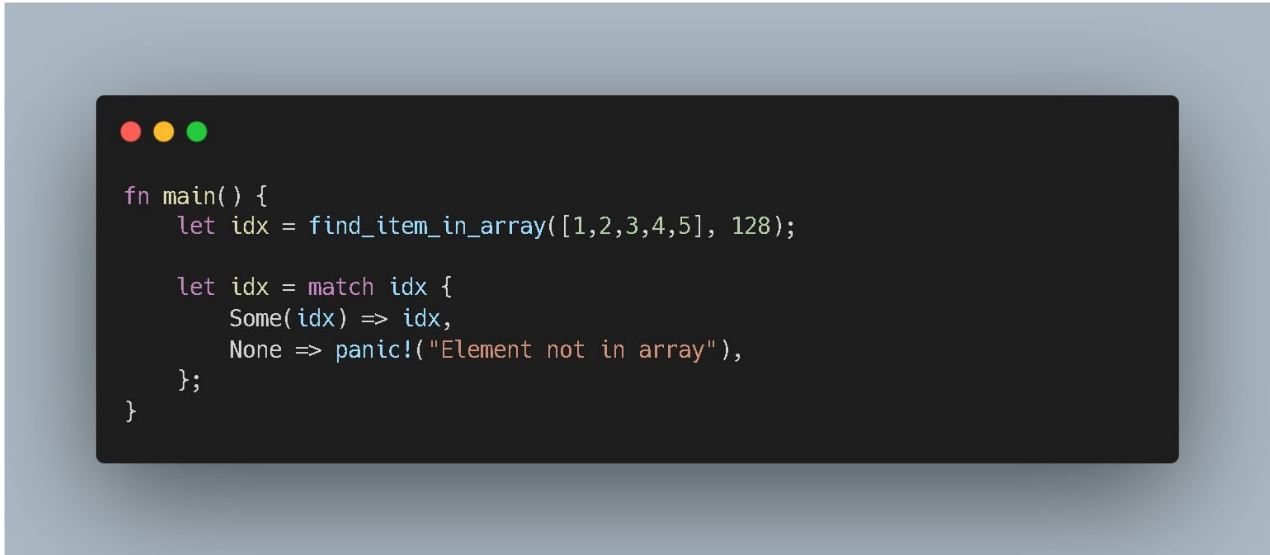


```rust
use std::fs::File;

fn main() {
    let f = File::open("hello.txt");

    let f = match f {
        Ok(file) => file,
        Err(error) => panic!("Problem opening the file: {:?}", error),
    };
}
```

Result/Option

The **Option** enum represents that some value may not exist

- You were expecting some user input, but they input nothing.
  - Or, you are searching some array for an element but that element is not in the array
- There are two cases for the **Option** enum
  - **Some(T)**
  - **None**



```rust
fn main() {
    let idx = find_item_in_array([1,2,3,4,5], 128);

    let idx = match idx {
        Some(idx) => idx,
        None => panic!("Element not in array"),
    };
}
```

Vec! / Vector

Recall:

Tuple	Array	
Fixed Length	Fixed Length	
(potentially) Multiple Types	Same type	

Vec! / Vector

Recall:

Tuple	Array	Vector
Fixed Length	Fixed Length	Variable Length
(potentially) Multiple Types	Same type	Same Type

Vec! / Vector

Recall:

We won't cover these yet… There's still some key concepts (ownership) that we need to cover first.

Tuple	Array	Vector
Fixed Length	Fixed Length	Variable Length
(potentially) Multiple Types	Same type	Same Type

Homework 1

Let’s look at it together.

That's all Folks!
SUPERB wallpapers

Vec! / Vector

Dolor sit amet consequat sit erat

CS128
Honors

Reminders:

extra/0 credit practice problems that are always open - mention in lecture when each are possible

add github repo with example code from lecture

add easy EC points to MPs - showcase interesting extensions during lecture

whenever we give lecture give them a chapter to follow along with

student interaction in lecture like steltzer- email students if they do well

mention common pitfalls of MPs after due date

partial credit until 1 week after 50%

emphasize early that you will get whatever you put in - lots of opportunities to do more