Parallelism/Concurrency

Communication

Goals For Today

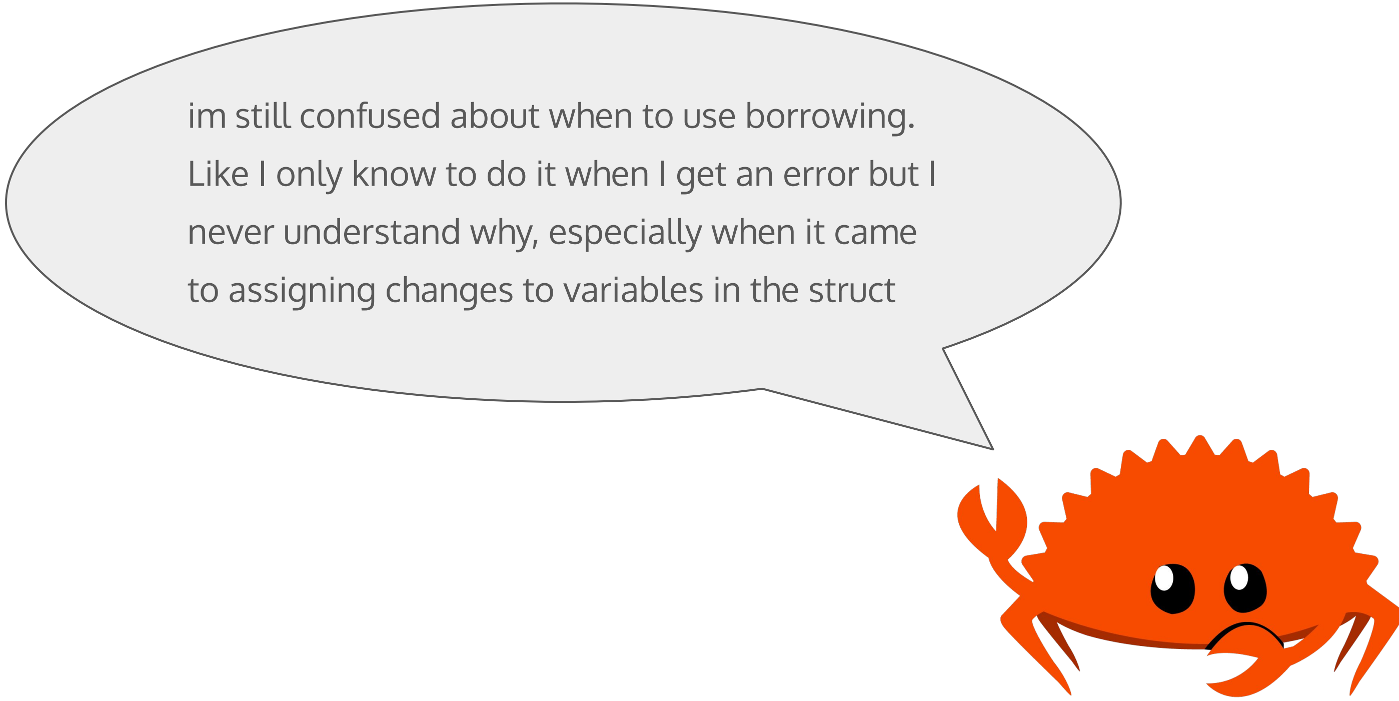
- Go Over MP2
- Review MPSC
- MPSC Example

Don’t Forget

- Nothing Due Soon
- HW10 releasing today

- No more HWs after HW10 :)

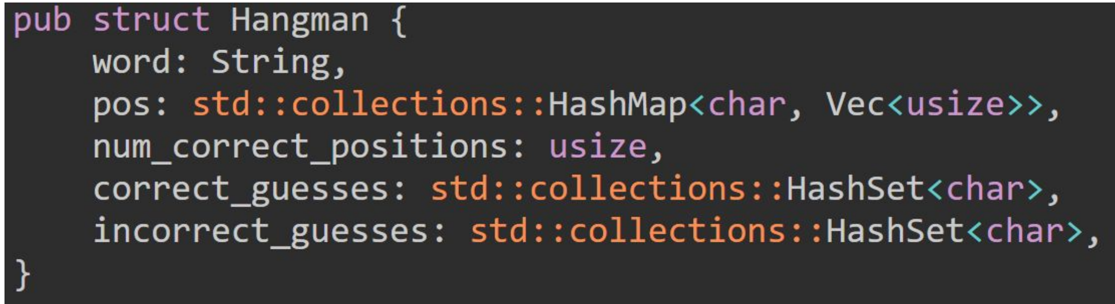
But First


im still confused about when to use borrowing. Like I only know to do it when I get an error but I never understand why, especially when it came to assigning changes to variables in the struct

MP2 - Overview

We want to create a Hangman game in Rust.

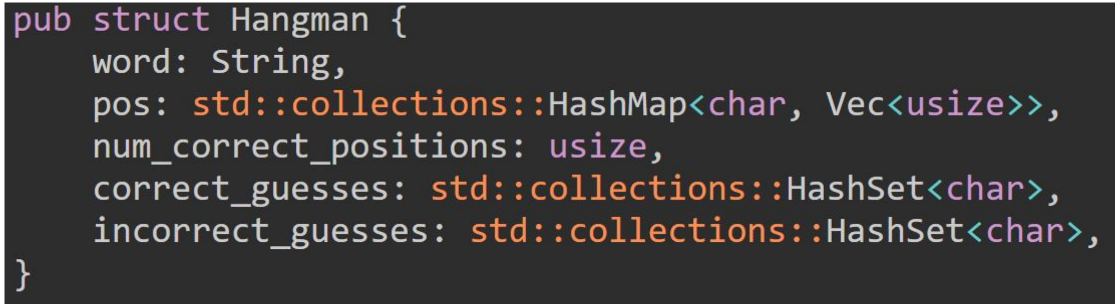
We provide you with an example of a Struct to use for this task.



```rust
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - New

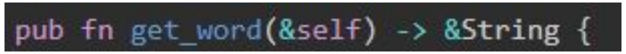
- Complete the new function. This should instantiate a new Hangman game object for the given word. You should return an Ok(Hangman) if the word is valid, and return an Err(HangmanError) with the HangmanErrorKind::InvalidWord enum value if the word is an empty string or contains non-alpha chars.
    a. Check if word is empty
    b. Ensure chars are alphabetic
    c. Return Hangman Struct



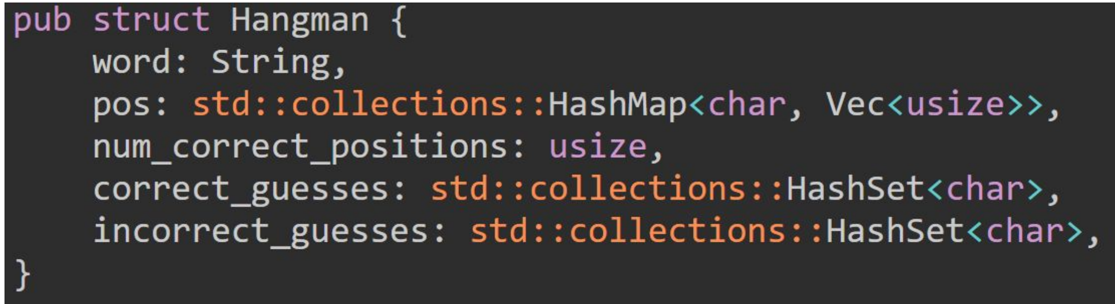
```
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - get_word

- Complete the `get_word` function. This should return a reference to the game word converted to lowercase.
    a. Return self.word
        i. It must be lowercase



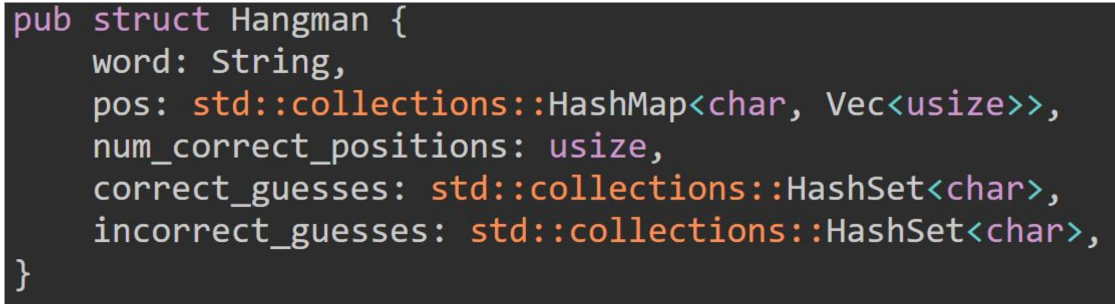
```rust
pub fn get_word(&self) -> &String {
```



```rust
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - get_num_guesses_left

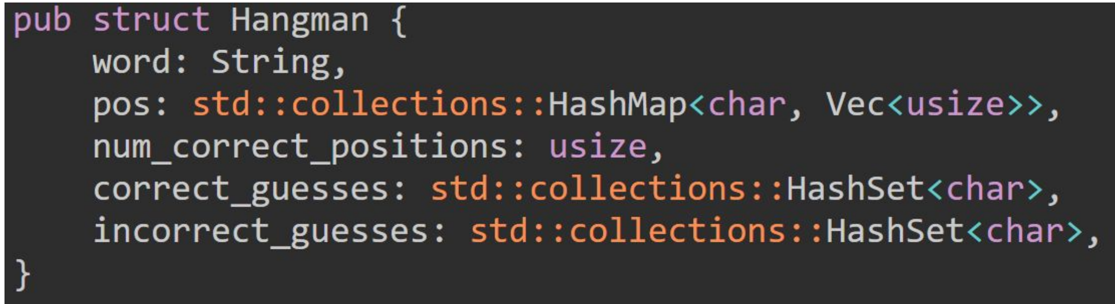
- Complete the `get_num_guesses_left` function. This should return the number of guesses left before the guesser loses.



```rust
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - get_(in)correct_guesses

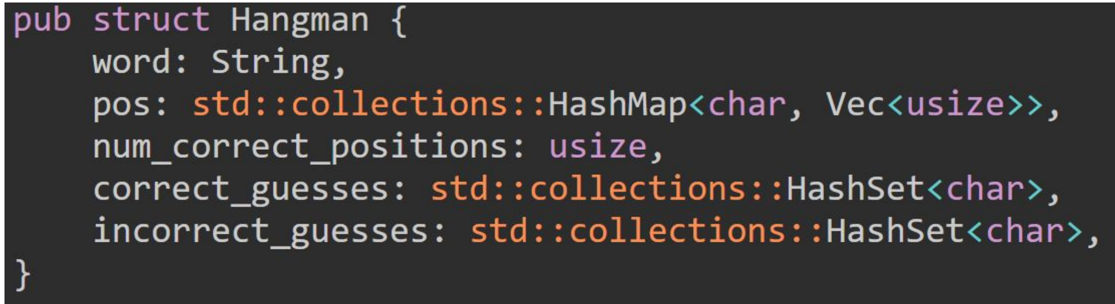
- Complete the `get_correct_guesses` function. This should return a reference to a HashSet of all correct guessed characters.
- Complete the `get_incorrect_guesses` function. This should return a reference to a HashSet of all incorrectly guessed characters.



```
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - get_game_result

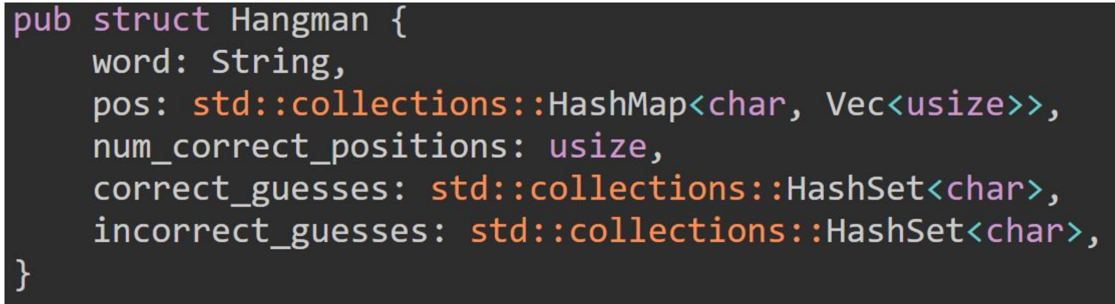
- Complete the `get_game_result` function. This should return the result of the game, obviously.
  You should return `Some(true)` if the user guessed all the characters in the word without exceeding the allowed number of guesses, and you should return `Some(false)` if the user made too many incorrect guesses. Finally, if the game is still in progress, then you should return `None`.



```rust
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - guess

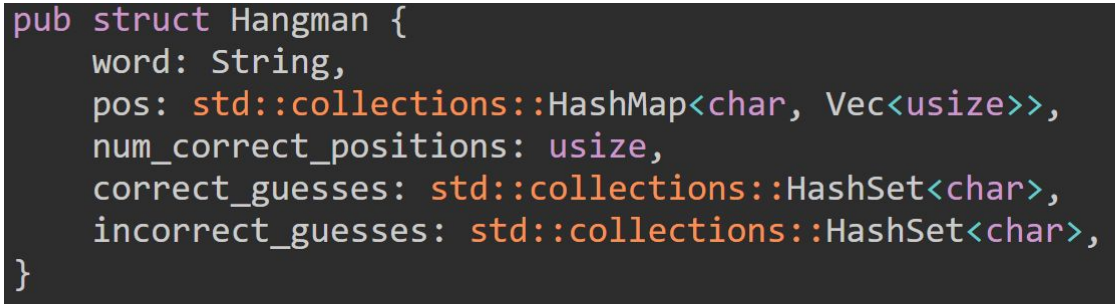
- Complete the `guess` function. This guesses a character in the hangman game and updates the game state. You should return `Ok(true)` if the guess was valid and correct, and return `Ok(false)` if the guess was valid but incorrect. If the guess was invalid, then you should return an `Err(HangmanError)`. `HangmanError` is defined in the `hangman/error.rs` file. Importantly, `HangmanError` implements the `std::error::Error` trait. The `HangmanError` struct has a factory method called `new` which allows you to instantiate a `HangmanError` with a `HangmanErrorKind` enum value, and the user input which was invalid. There are a variety of different `HangmanError` enum values you should use. You should use a `GameAlreadyOver` error kind if the game was already finished before the guess, an `InvalidCharacter` error kind if the character is not alphabetic, and an `AlreadyGuessedCharacter` error kind if the character was already guessed (either correctly or incorrectly). For example: if the user's input is stored in the variable `user_input`, and the user_input is an invalid character, you can return the appropriate HangmanError using
  `return HangmanError::new(HangmanErrorKind::InvalidCharacter, user_input);`.
  Note: the guess is case INSENSITIVE (e.g., if the word is "abc," both 'A' and 'a' are correct guesses).



```
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - guess

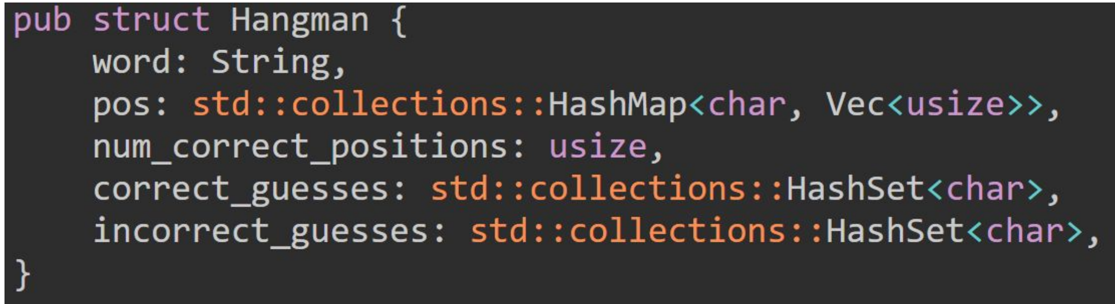
- Complete the `guess` function. This guesses a character in the hangman game and updates the game state. You should return `Ok(true)` if the guess was valid and correct, and return `Ok(false)` if the guess was valid but incorrect. If the guess was invalid, then you should return an `Err(HangmanError)`. `HangmanError` is defined in the `hangman/error.rs` file. Importantly, `HangmanError` implements the `std::error::Error` trait. The `HangmanError` struct has a factory method called `new` which allows you to instantiate a `HangmanError` with a `HangmanErrorKind` enum value, and the user input which was invalid. There are a variety of different `HangmanError` enum values you should use. **You should use a `GameAlreadyOver` error kind if the game was already finished before the guess**, an `InvalidCharacter` error kind if the character is not alphabetic, and an `AlreadyGuessedCharacter` error kind if the character was already guessed (either correctly or incorrectly). For example: if the user's input is stored in the variable `user_input`, and the user_input is an invalid character, you can return the appropriate HangmanError using
`return HangmanError::new(HangmanErrorKind::InvalidCharacter, user_input);`.
Note: the guess is case INSENSITIVE (e.g., if the word is "abc," both 'A' and 'a' are correct guesses).



```
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - guess

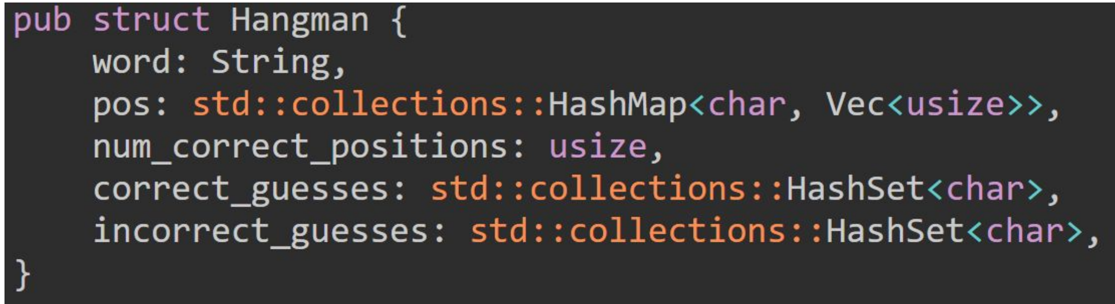
- Complete the `guess` function. This guesses a character in the hangman game and updates the game state. You should return `Ok(true)` if the guess was valid and correct, and return `Ok(false)` if the guess was valid but incorrect. If the guess was invalid, then you should return an `Err(HangmanError)`. `HangmanError` is defined in the `hangman/error.rs` file. Importantly, `HangmanError` implements the `std::error::Error` trait. The `HangmanError` struct has a factory method called `new` which allows you to instantiate a `HangmanError` with a `HangmanErrorKind` enum value, and the user input which was invalid. There are a variety of different `HangmanError` enum values you should use. You should use a `GameAlreadyOver` error kind if the game was already finished before the guess**, an `InvalidCharacter` error kind if the character is not alphabetic**, and an `AlreadyGuessedCharacter` error kind if the character was already guessed (either correctly or incorrectly). For example: if the user's input is stored in the variable `user_input`, and the user_input is an invalid character, you can return the appropriate HangmanError using
`return HangmanError::new(HangmanErrorKind::InvalidCharacter, user_input);`.
Note: the guess is case INSENSITIVE (e.g., if the word is "abc," both 'A' and 'a' are correct guesses).



```
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - guess

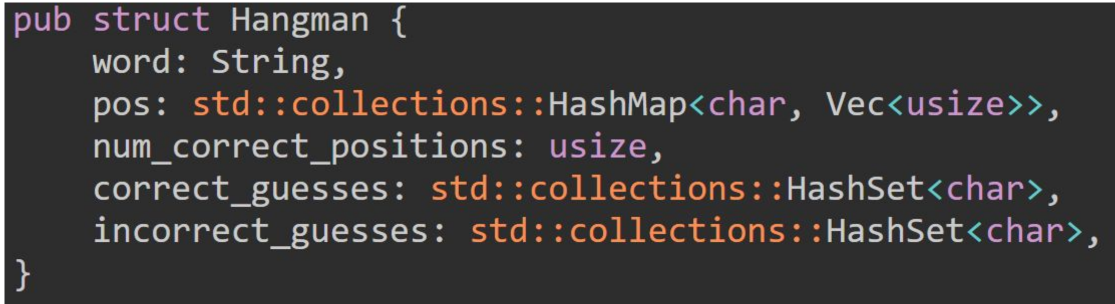
- Complete the `guess` function. This guesses a character in the hangman game and updates the game state. You should return `Ok(true)` if the guess was valid and correct, and return `Ok(false)` if the guess was valid but incorrect. If the guess was invalid, then you should return an `Err(HangmanError)`. `HangmanError` is defined in the `hangman/error.rs` file. Importantly, `HangmanError` implements the `std::error::Error` trait. The `HangmanError` struct has a factory method called `new` which allows you to instantiate a `HangmanError` with a `HangmanErrorKind` enum value, and the user input which was invalid. There are a variety of different `HangmanError` enum values you should use. You should use a `GameAlreadyOver` error kind if the game was already finished before the guess, an `InvalidCharacter` error kind if the character is not alphabetic, **and an `AlreadyGuessedCharacter` error kind if the character was already guessed (either correctly or incorrectly)**. For example: if the user's input is stored in the variable `user_input`, and the user_input is an invalid character, you can return the appropriate HangmanError using
`return HangmanError::new(HangmanErrorKind::InvalidCharacter, user_input);`.
Note: the guess is case INSENSITIVE (e.g., if the word is "abc," both 'A' and 'a' are correct guesses).



```
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

MP2 - guess

- Complete the `guess` function. This guesses a character in the hangman game and updates the game state. You should return `Ok(true)` if the guess was valid and correct, and return `Ok(false)` if the guess was valid but incorrect.



```rust
pub struct Hangman {
    word: String,
    pos: std::collections::HashMap<char, Vec<usize>>,
    num_correct_positions: usize,
    correct_guesses: std::collections::HashSet<char>,
    incorrect_guesses: std::collections::HashSet<char>,
}
```

Review MPSC

- One form of communication is to use message passing
    - We create an MPSC channel (multiple producer, single consumer)
        - *Does MPMC exist?*

Let’s do a new example...

Review MPSC

- One form of communication is to use message passing
  - We create an MPSC channel (multiple producer, single consumer)
    - *Does MPMC exist?*

Let’s do a new example…

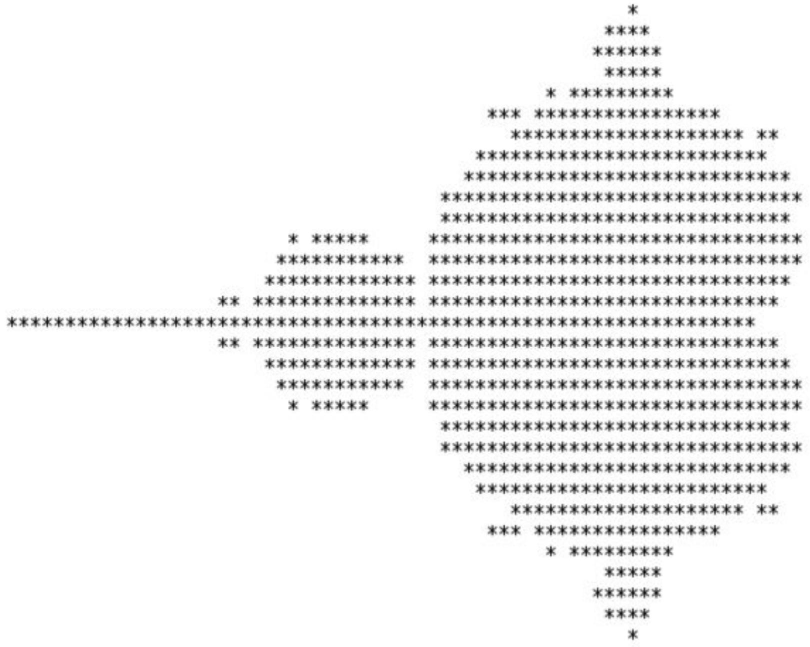
$$Z_{n+1} = Z_n^2 + C$$

Review MPSC

- One form of communication is to use message passing
  - We create an MPSC channel (multiple producer, single consumer)
    - *Does MPMC exist?*

Let’s do a new example…

$$Z_{n+1} = Z_n^2 + C$$

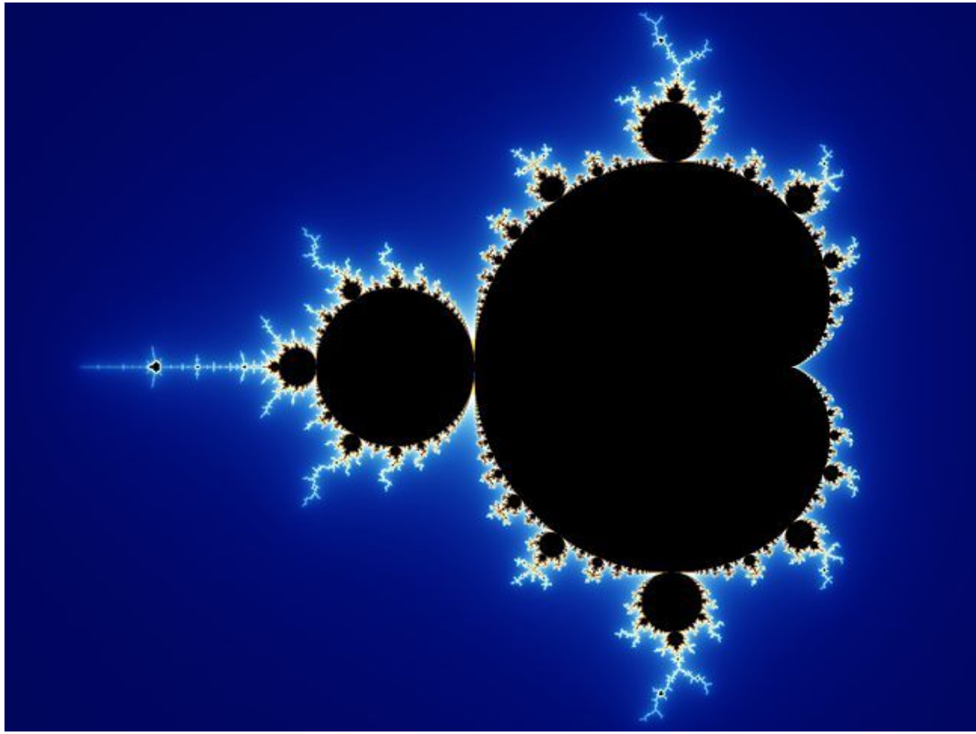


Review MPSC

- One form of communication is to use message passing
  - We create an MPSC channel (multiple producer, single consumer)
    - *Does MPMC exist?*

Let’s do a new example…

$$f(z) = z^2 + c$$

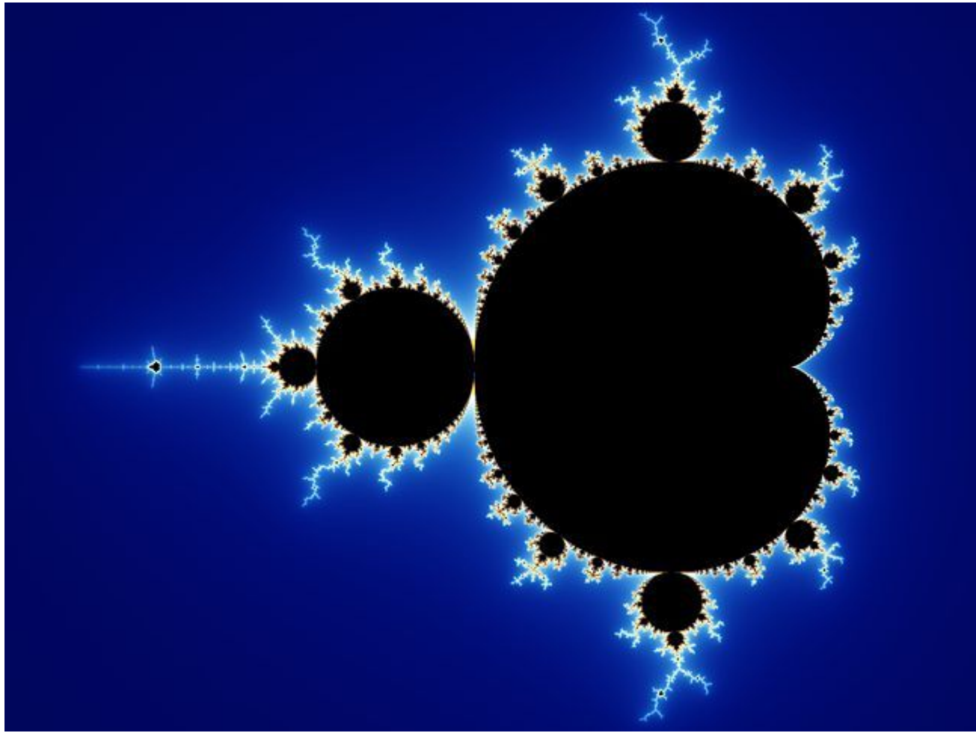


Review MPSC

- One form of communication is to use message passing
    - We create an MPSC channel (multiple producer, single consumer)
        - *Does MPMC exist?*

Let's do a new example...

$$f(z) = z^2 + c$$



That’s all for now!

See you next episode.