notes

March 26, 2022

1 Rust Notes

Pretty much a summary of https://doc.rust-lang.org/book/.

1.1 functional programming language

`main.main` (main.rs) is the main function of a program.

1.2 Basics

print with `println!` and `print!` functions.

`let` declares variables, `mut` makes variables mutable.

colons used for assigning types.

1.2.1 Loops

- loops are by default infinite
- break and continue to exit and skip
- while loops also exist
- for loops need to be made manually (the `start..end` syntax is available)
- loops can be named for specifying which loop to break and continue
- loops can return values

1.3 Structs

`struct` is similar to `interface` in typescript (used for typing objects).

[57]:
```
struct Point {
  x: i32,
  y: i32,
}

let mut point = Point { x: 0, y: 5 };
println!("point is ({}, {})", point.x, point.y);
point.x = 5;
println!("point is ({}, {})", point.x, point.y);
```

```
point is (0, 5)
point is (5, 5)
```

`..object` uses the fields from the given instance to fill in the rest.

[58]:
```
let mut point2 = Point {
  x: 0,
  ..point
};
println!("point is ({}, {})", point.x, point.y);
point.x = 5;
println!("point is ({}, {})", point.x, point.y);
```

```
point is (5, 5)
point is (5, 5)
```

tuple structs can be used to make a struct without named fields.

[59]:
```
struct PointT(i32, i32, i32);

let mut point = PointT(1, 2, 3);
println!("p is ({}, {}, {})", point.0, point.1, point.2);
point.1 = 5;
println!("p is ({}, {}, {})", point.0, point.1, point.2);
```

```
p is (1, 2, 3)
p is (1, 5, 3)
```

unit-like structs act similarly to a single field on an enum and are always equal to themselves

[60]:
```
struct AlwaysEqual;

let subject = AlwaysEqual;
```

structs can have implementation blocks

[61]:
```
struct Rectangle {
  width: u32,
  height: u32,
}

impl Rectangle {
  fn area(&self) -> u32 {
  self.width * self.height
  }
}
// the first parameter of a method is always a reference to the parent
```

1.4 Enums

`enum` is the same as in typescript

[62]:
```rust
#[derive(PartialEq)]
enum Direction {
  Up,
  Down,
  Left,
  Right,
}

let direction = Direction::Up;

fn go(dir: Direction) {
  if dir == Direction::Up {
    println!("Going up");
  } else if dir == Direction::Down {
    println!("Going down");
  } else if dir == Direction::Left {
    println!("Going left");
  } else if dir == Direction::Right {
    println!("Going right");
  }
}

go(direction);
go(Direction::Down);
```

```
Going up
Going down
```

above the `#[derive(PartialEq)]` allows the enum to be compared with `==`.

enums can be associated data types/values

[63]:
```rust
enum Direction2 {
  Up(i32),
  Down(i32),
  Left(i32),
  Right(i32)
}
```

enums can also have multiple types/values associated with them

[64]:
```rust
enum Direction3 {
  Up(i32),
  Down(i32),
  Left(i32),
  Right(i32),
  UpLeft(i32, i32),
  UpRight(i32, i32),
  DownLeft(i32, i32),
```

```
  DownRight(i32, i32),
}
```

enums can also have (multiple) named types associated with themselves

[65]:
```
enum Direction4 {
  Up(i32),
  Down(i32),
  Left(i32),
  Right(i32),
  UpLeft{ up: i32, left: i32 },
  UpRight{ up: i32, right: i32 },
  DownLeft{ down: i32, left: i32 },
  DownRight{ down: i32, right: i32 },
}
```

enums can alo have implementations similarly to structs

[66]:
```
enum Direction5 {
  Up(i32),
  Down(i32),
  Left(i32),
  Right(i32),
}

impl Direction5 {
  fn is_up(&self) -> bool {
    match self {
      Direction5::Up(_) => true,
      _ => false,
    }
  }
}

println!("{}", Direction5::Up(1).is_up());
println!("{}", Direction5::Down(1).is_up());
```

```
true
false
```

enums can also have options

[67]:
```
enum Direction6<T> {
  Up(T),
  Down(T),
  Left(T),
  Right(T),
  UpLeft{ up: T, left: T },
```

```
  UpRight{ up: T, right: T },
  DownLeft{ down: T, left: T },
  DownRight{ down: T, right: T },
}

// now all the values are of the type `T`

Direction6::<u32>::Up(1u32);
Direction6::<f32>::Up(1.1f32);
```

There is no null in rust, but there is a `Option` type.

[68]:
```
// enum Option<T> {
//   None,
//   Some(T),
// }

Option::<u32>::None; // this is null but it is the same type as:
Option::<u32>::Some(10u32);

Option::<f32>::None; // this is null but it is the same type as:
Option::<f32>::Some(10.01f32);
```

`Option` is included in the standard library and does not need to be imported.

1.5 Matches

`match` is similar to `switch` in javascript but it can return a value not just execute a block.

[69]:
```
enum Coin {
  Penny,
  Nickel,
  Dime,
  Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
  match coin {
    Coin::Penny => 1,
    Coin::Nickel => 5,
    Coin::Dime => 10,
    Coin::Quarter => 25,
  }
}
```

fellow englishmen I agree the American coin system makes no sense

`match` can also take the value from the enum

[70]:
```rust
#[derive(Debug)] // this line is required to print the enum
enum UsState {
  Alabama,
  Alaska,
  // ...
}

enum Coin2 {
  Penny,
  Nickel,
  Dime,
  Quarter(UsState),
}

fn value_in_cents(coin: Coin2) -> u8 {
  match coin {
    Coin2::Penny => 1,
    Coin2::Nickel => 5,
    Coin2::Dime => 10,
    Coin2::Quarter(state) => {
      println!("State quarter from {:?}!", state);
      25
    }
  }
}
```

`match` with `Option<T>`

[71]:
```rust
fn plus_one(n: Option<i32>) -> Option<i32> {
  match n {
    None => None,
    Some(i) => Some(i + 1),
  }
}

let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

the `other` pattern is like the `default` pattern in javascript

[72]:
```rust
let n = 10;
println!(
  "{}",
  match n {
    3 => 1,
    7 => 3,
    other => other,
```

```
  }
)
```

```
10
```

[72]:
```
()
```

the _ pattern is like the `other` pattern but it doesn't bind the value

[73]:
```
let n = 10;
println!(
  "{}",
  match n {
    3 => 1,
    7 => 3,
    _ => 7,
  }
)
```

```
7
```

[73]:
```
()
```

1.6 If Let

`if let` combines `if` and `let`

it simplifies

[74]:
```
let some_option = Some(3);
match some_option {
  Some(amount) => println!("number {}!", amount),
  _ => (),
}
```

```
number 3!
```

[74]:
```
()
```

to

[75]:
```
let some_option = Some(3);
if let Some(amount) = some_option {
  println!("number {}!", amount);
}
```

```
number 3!
```

[75]:
```
()
```

`if let` can also be used to match on multiple values

[76]:
```
enum Direction7 {
  Up(i32),
  Down(i32),
  Left(i32),
  Right(i32),
}

let some_option = Direction7::Up(3);
match some_option {
  Direction7::Up(amount) => println!("up: {}!", amount),
  Direction7::Down(amount) => println!("down: {}!", amount),
  Direction7::Left(amount) => println!("left: {}!", amount),
  Direction7::Right(amount) => println!("right: {}!", amount),
  _ => (),
}
```

```
up: 3!
```

[76]:
```
()
```

to

[77]:
```
enum Direction8 {
  Up(i32),
  Down(i32),
  Left(i32),
  Right(i32),
}

let some_option = Direction8::Up(3);
if let Direction8::Up(amount) = some_option {
  println!("up: {}!", amount);
} else if let Direction8::Down(amount) = some_option {
  println!("down {}!", amount);
} else if let Direction8::Left(amount) = some_option {
  println!("left {}!", amount);
} else if let Direction8::Right(amount) = some_option {
  println!("right {}!", amount);
}
```

```
up: 3!
```

[77]:
```
()
```

1.7 Vectors

`Vec<T>` (aka vectors) is a collection type or list - in rust like most other languages, the first element in a list is at index 0 - `Vec` is simply a growable array

[78]:
```rust
// create an empty vector
let v: Vec<i32> = Vec::new();
println!("create an empty vector: {:?}", v);

// create a vector with some values
let v = vec![1, 2, 3];
println!("create a vector with some values: {:?}", v);

// create a vector and add values dynamically
let mut v = Vec::new();
v.push(1);
v.push(2);
v.push(3);

// get an item from a vector
let v = vec![1, 2, 3];
println!("get an item from a vector: {:?}", &v[2]);

// iterate over a vector
let v = vec![100, 32, 57];
for i in &v {
  println!("iterate over a vector: {}", i);
}

// get an item that might not exist from a vector
let v = vec![100, 32, 57];
// println!("get an item that might not exist from a vector: {:?}", &v[100]);
// this will cause a panic
// because the index is out of bounds
// instead use the get method
println!("get an item that might not exist from a vector: {:?}", v.get(100));

// iterate over a vector and mutate the values
let mut v = vec![100, 32, 57];
for i in &mut v {
  *i += 50;
}
println!("iterate over a vector and mutate the values: {:?}", v);

// storing multiple values with enums
#[derive(Debug)]
enum SpreadsheetCell {
  Int(i32),
```

```
  Float(f64),
  Text(String),
}

let row = vec![
  SpreadsheetCell::Int(3),
  SpreadsheetCell::Text(String::from("blue")),
  SpreadsheetCell::Float(10.12),
];
println!("storing multiple values with enums: {:?}", row);

// create a vector and remove values dynamically
let mut v = vec![0, 1, 2, 3, 4, 5];
// `pop` also returns the value that was removed
v.pop(); // returns Some(5)
v.pop(); // returns Some(4)
v.pop(); // returns Some(3)
println!("create a vector and remove values dynamically: {:?}", v);
```

```
create an empty vector: []
create a vector with some values: [1, 2, 3]
get an item from a vector: 3
iterate over a vector: 100
iterate over a vector: 32
iterate over a vector: 57
get an item that might not exist from a vector: None
iterate over a vector and mutate the values: [150, 82, 107]
storing multiple values with enums: [Int(3), Text("blue"), Float(10.12)]
create a vector and remove values dynamically: [0, 1, 2]
```

1.8 Hashmaps

`HashMap<K, V>` is like an object in javascript (it stores key-value pairs) - `HashMap` is a collection type - you must always import `HashMap` before using it

[79]:
```
use std::collections::HashMap;
```

[80]:
```
// create an empty hashmap
let scores = HashMap::<String, u8>::new();

println!("create an empty hashmap: {:?}", scores);

// create a hashmap with some values
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

println!("create a hashmap with some values: {:?}", scores);
```

```rust
// create a `HashMap` from iterables
let teams = vec![String::from("Blue"), String::from("Yellow")];
let initial_scores = vec![10, 50];
let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();

println!("create a `HashMap` from iterables: {:?}", &scores);

// get a value from a hashmap
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

println!("get a value from a hashmap: {}", scores.get(&String::from("Blue")).
 ↪unwrap());

// iterate over a hashmap
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in &scores {
  println!("iterate over a hashmap: {}: {}", key, value);
}

// iterate over a hashmap and mutate the values
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (_, value) in &mut scores {
  *value += 10;
}

println!("iterate over a hashmap and mutate the values: {:?}", scores);

// overwrite a value in a hashmap
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
scores.insert(String::from("Blue"), 20);

println!("overwrite a value in a hashmap: {:?}", scores);

// only insert a value if the key is not already present
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
```

```
scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("only insert a value if the key is not already present: {:?}", scores);

// update a value based on the old value
let text = "hello world wonderful world";
let mut map = HashMap::new();

for word in text.split_whitespace() {
  let count = map.entry(word).or_insert(0);
  *count += 1;
}

println!("update a value based on the old value: {:?}", map);
```

```
create an empty hashmap: {}
create a hashmap with some values: {"Blue": 10, "Yellow": 50}
create a `HashMap` from iterables: {"Yellow": 50, "Blue": 10}
get a value from a hashmap: 10
iterate over a hashmap: Blue: 10
iterate over a hashmap: Yellow: 50
iterate over a hashmap and mutate the values: {"Yellow": 60, "Blue": 20}
overwrite a value in a hashmap: {"Blue": 20, "Yellow": 50}
only insert a value if the key is not already present: {"Blue": 10, "Yellow":
50}
update a value based on the old value: {"wonderful": 1, "hello": 1, "world": 2}
```

1.9 Error Handling

`panic!` throws an exception and stops the program

[81]:
```
panic!("crash and burn");
```

```
thread '<unnamed>' panicked at 'crash and burn', src/lib.rs:404:1
stack backtrace:
   0: std::panicking::begin_panic
   1: run_user_code_64
   2: <unknown>
   3: <unknown>
   4: <unknown>
   5: <unknown>
   6: <unknown>
   7: <unknown>
   8: <unknown>
   9: <unknown>
  10: BaseThreadInitThunk
  11: RtlUserThreadStart
```

```
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose
backtrace.
```

```
Child process terminated with status: exit code: 0xc0000005
```

`Result<T, E>` is the type used if something can be an error or a value

[ ]:
```rust
// enum Result<T, E> {
//   Ok(T),
//   Err(E),
// }


// `Ok` is a successful result
let x: Result<i32, &str> = Ok(5i32);

// `Err` is an error result
let x: Result<i32, &str> = Err("Error");

// `match` statements are used to handle results
let x: Result<i32, &str> = Ok(5i32);
let y: Result<i32, &str>  = Err("Error");

match x {
  Ok(val) => println!("handle results: success {}", val),
  Err(e) => println!("handle results: error {}", e),
}

match y {
  Ok(val) => println!("handle results: success {}", val),
  Err(e) => println!("handle results: error {}", e),
}

// match on different errors
use std::fs::File;
use std::io::ErrorKind;

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

let f = match f {
  Ok(file) => file,
  Err(error) => match error.kind() {
    ErrorKind::NotFound => match File::create("hello.txt") {
      Ok(fc) => fc,
      Err(e) => panic!("Problem creating the file: {:?}", e),
    },
    other_error => {
```

```
      panic!("Problem opening the file: {:?}", other_error)
    }
  },
};

println!("match on different errors: {:?}", f);

// the `expect` function is used to handle errors
use std::fs::File;

let f = File::open("hello.txt").expect("Failed to open hello.txt");

// propagate errors
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
  let f = File::open("hello.txt");

  let mut f = match f {
    Ok(file) => file,
    Err(e) => return Err(e),
  };

  let mut s = String::new();

  match f.read_to_string(&mut s) {
    Ok(_) => Ok(s),
    Err(e) => Err(e),
  }
}

// propagate errors with `?` (this does the same as above)
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
  let mut f = File::open("hello.txt")?;
  let mut s = String::new();
  f.read_to_string(&mut s)?;
  Ok(s)
}

// even shorter
use std::fs::File;
use std::io::{self, Read};
```

```rust
fn read_username_from_file() -> Result<String, io::Error> {
  let mut s = String::new();
  File::open("hello.txt")?.read_to_string(&mut s)?;
  Ok(s)
}

// `?` can only be used if the return type is `Result`, `Option` or another␣
 ↪type that implements `std::ops::Try`

// the main function can be given another return type to get around this
use std::error::Error;
use std::fs::File;

fn main() -> Result<(), Box<dyn Error>> {
  let f = File::open("hello.txt")?;

  Ok(())
}
```

```
handle results: success 5
handle results: error Error
match on different errors: File { handle: 0x1c4, path:
"\\\\?\\C:\\Users\\Joseph\\code\\learning-rust\\hello.txt" }
```

`Result<T, E>` is included in the standard library and does not need to be imported.

1.10 Traits

`trait` can be used to make a list of functions an object must have, this is a bit similar to inheritance in javascript - this makes it so that we can have a group of different types that can be used under the same name - `impl` can be used to implement a trait for an object

[ ]:
```rust
trait Draw {
  fn draw(&self);
}

struct Button {
  width: u32,
  height: u32,
  label: String,
}

impl Draw for Button {
  fn draw(&self) {
    println!("drawing button: ({}, {}) {}", self.width, self.height, self.
 ↪label);
  }
}
```

```rust
struct Paragraph {
  width: u32,
  height: u32,
  text: String,
}

impl Draw for Paragraph {
  fn draw(&self) {
    println!("drawing paragraph: ({}, {}) {}", self.width, self.height, self.
 ↪text);
  }
}

// so as long as the type implements the trait, it can be used as a trait object
pub struct Screen {
  pub components: Vec<Box<dyn Draw>>,
}

impl Screen {
  pub fn run(&self) {
    for component in self.components.iter() {
      component.draw();
    }
  }
}

// as you can see the Paragraph and Button types can be used under the Draw␣
 ↪trait
let screen = Screen {
  components: vec![
    Box::new(Paragraph {
      width: 75,
      height: 10,
      text: String::from("Hello World!"),
    }),
    Box::new(Button {
      width: 50,
      height: 10,
      label: String::from("OK"),
    }),
  ],
};

screen.run();
```

```
drawing paragraph: (75, 10) Hello World!
drawing button: (50, 10) OK
```

1.11 Threading

`std::thread`, similar to python's `threading` module, allows you to spawn a new thread to run asynchronously

[ ]:
```rust
use std::thread;
use std::time::Duration;

thread::spawn(|| {
  for i in 1..10 {
    println!("hi number {} from the spawned thread!", i);
    thread::sleep(Duration::from_millis(1));
  }
});

for i in 1..5 {
  println!("hi number {} from the main thread!", i);
  thread::sleep(Duration::from_millis(1));
}
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the main thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
```

[ ]:
```rust
()
```

it is important to note that the thread is spawned in the background, so the main thread will continue to run and the spawned thread will stop running when the main thread is done

wait for a thread to finish with `join`

[82]:
```rust
use std::thread;
use std::time::Duration;

let handle = thread::spawn(|| {
  for i in 1..10 {
    println!("hi number {} from the spawned thread!", i);
    thread::sleep(Duration::from_millis(1));
  }
});
```

```
for i in 1..5 {
  println!("hi number {} from the main thread!", i);
  thread::sleep(Duration::from_millis(1));
}

handle.join().unwrap();
```

```
hi number 1 from the main thread!
hi number 1 from the spawned thread!
hi number 2 from the main thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 4 from the main thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

use the `move` keyword to move ownership of a value into a new thread

[ ]:
```
let v = vec![1, 2, 3];
let t = thread::spawn(|| { println!("{:?}", v); });
drop(v);
t.join().unwrap();
```

```
let t = thread::spawn(|| { println!("{:?}", v); });
                                              ^ `v` is borrowed here
let t = thread::spawn(|| { println!("{:?}", v); });
                         ^^ may outlive borrowed value `v`
closure may outlive the current function, but it borrows `v`, which is owned by
 ↪the current function
help: to force the closure to take ownership of `v` (and any other referenced␣
 ↪variables), use the `move` keyword

move
```

```
let t = thread::spawn(|| { println!("{:?}", v); });
                         ^^ borrow of `v` occurs here
drop(v);
     ^ move out of `v` occurs here
let t = thread::spawn(|| { println!("{:?}", v); });
                                              ^ borrow occurs due to use in closu
let t = thread::spawn(|| { println!("{:?}", v); });
```

```
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ argument requires that `v` i
 ↪borrowed for `'static`
cannot move out of `v` because it is borrowed
```

[ ]:
```
let v = vec![1, 2, 3];
let t = thread::spawn(move || { println!("{:?}", v); });
drop(v);
t.join().unwrap();
```

```
let t = thread::spawn(move || { println!("{:?}", v); });
                      ^^^^^^^ value moved into closure here
let t = thread::spawn(move || { println!("{:?}", v); });
                                                 ^ variable moved due to use in
 ↪closure
drop(v);
     ^ value used here after move
let v = vec![1, 2, 3];
    ^ move occurs because `v` has type `Vec<i32>`, which does not implement the
 ↪`Copy` trait
use of moved value: `v`
```

[ ]:
```
let v = vec![1, 2, 3];
let t = thread::spawn(move || { println!("{:?}", v); });
t.join().unwrap();
```

```
[1, 2, 3]
```

> "Do not communicate by sharing memory; instead, share memory by communicating."

https://golang.org/doc/effective_go#concurrency

this is a good example of why you should not communicate by sharing memory, instead communicate by communicating

[ ]:
```
use std::thread;
use std::time::Duration;

let v = vec![1, 2, 3];
let handle = thread::spawn(move || {
  println!("(sub-thread) Here's a vector: {:?}", v);
});
println!("Here's a vector: {:?}", v);
handle.join().unwrap();
```

```
let handle = thread::spawn(move || {
                           ^^^^^^^ value moved into closure here
  println!("(sub-thread) Here's a vector: {:?}", v);
```

```
                                                    ^ variable moved due to use in
 ↪closure
println!("Here's a vector: {:?}", v);
                                    ^ value borrowed here after move
let v = vec![1, 2, 3];
    ^ move occurs because `v` has type `Vec<i32>`, which does not implement the
 ↪`Copy` trait
borrow of moved value: `v`
```

instead of sharing variables between threads, pass messages between threads

[ ]:
```
use std::sync::mpsc;
use std::thread;

let (tx, rx) = mpsc::channel();

let spawned = thread::spawn(move || {
  tx.send(String::from("hello")).unwrap();
});

let received = rx.recv().unwrap();
println!("Got: {}", received);

spawned.join().unwrap();
```

```
Got: hello
```

send multiple messages and iterate over them

[ ]:
```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

let (tx, rx) = mpsc::channel();

let spawned = thread::spawn(move || {
  let vals = vec![
    String::from("hi"),
    String::from("from"),
    String::from("the"),
    String::from("thread"),
  ];

  for val in vals {
    tx.send(val).unwrap();
    thread::sleep(Duration::from_secs(1));
  }
});
```

```
for received in rx {
  println!("{} ", received);
}

spawned.join().unwrap();
```

```
hi
from
the
thread
```

multiple producers

[ ]:
```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
  let vals = vec![
    String::from("hi"),
    String::from("from"),
    String::from("the"),
    String::from("thread"),
  ];

  for val in vals {
    tx1.send(val).unwrap();
    thread::sleep(Duration::from_secs(1));
  }
});

thread::spawn(move || {
  let vals = vec![
    String::from("more"),
    String::from("messages"),
    String::from("for"),
    String::from("you"),
  ];

  for val in vals {
    tx.send(val).unwrap();
    thread::sleep(Duration::from_secs(1));
  }
});
```

```
for received in rx {
    println!("Got: {}", received);
}
```

```
Got: hi
Got: more
Got: messages
Got: from
Got: for
Got: the
Got: you
Got: thread
```

[ ]: ()

1.12 Miscellaneous

to import a local module first use the mod key word followed by the name of the module

[ ]:
```
mod my_mod;
use my_mod::hello;
// there will be an error because I have not made the file "my_mod.rs"
```

```
mod my_mod;
^^^^^^^^^^^

file not found for module `my_mod`
help: to create the module `my_mod`, create file "src\my_mod.rs" or␣
 ↪"src\my_mod\mod.rs"
```

```
no `hello` in `my_mod`
unresolved import `my_mod::hello`
```

`std::env` includes the `args` function to get the args and the `var` function to get an environment variable

`eprintln!` and `eprint!` are macros that print to stderr

1.12.1 Templates

[ ]:
```
struct Message<T> {
  content: T,
  from: String,
  to: String,
}

let x = Message::<String> {
```

```
  content: "hello".to_string(),
  from: "Boris".to_string(),
  to: "Joe".to_string(),
};
let y = Message::<u32> {
  content: 123u32,
  from: "Joe".to_string(),
  to: "Boris".to_string(),
};
```

[ ]:
```
fn largest<T: PartialOrd>(list: &[T]) -> &T {
  let mut largest = &list[0];

  for item in list.clone() {
    if &item > &largest {
      largest = &item;
    }
  }

  largest
}

let number_list = vec![34, 50, 25, 100, 65];

println!("The largest number is {}", largest(&number_list));

let char_list = vec!['y', 'm', 'a', 'q'];

println!("The largest char is {}", largest(&char_list));
```

```
The largest number is 100
The largest char is y
```

[ ]:
```
// the ` + Copy` ensures that which ever type is given it must implement copy
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
  let mut largest = list[0];

  for &item in list {
    if item > largest {
      largest = item;
    }
  }

  largest
}

let number_list = vec![34, 50, 25, 100, 65];
```

```
println!("The largest number is {}", largest(&number_list));

let char_list = vec!['y', 'm', 'a', 'q'];

println!("The largest char is {}", largest(&char_list));
```

```
The largest number is 100
The largest char is y
```