Safe Programming With Rust

Slide credit today: Michael Hicks

Why Rust in a Software Engineering Course

- I've been telling you about the security implications of using unsafe languages
- But that advice is only actionable if you have a practical alternative!
- Enter: Rust.

What choice do programmers have today?

C/C++

- Low level
- More control
- Performance over safety
- Memory managed manually
- No periodic garbage collection
- …

Java, OCaml, Go, Ruby…

- High level
- Secure
- Less control
- Restrict direct access to memory
- Run-time management of memory via periodic garbage collection
- No explicit malloc and free
- Unpredictable behavior due to GC
- …

Rust: Type- and Thread-safe, and Fast

- Begun in 2006 by Graydon Hoare
- Sponsored as full-scale project and announced by Mozilla in 2010
  - Changed a lot since then; source of frustration
  - But now: most loved programming language in Stack Overflow annual surveys every year from 2016 through 2020
- Takes ideas from functional and OO languages, and recent research
- Key properties: Type safety, and no data races, despite use of concurrency and manual memory management

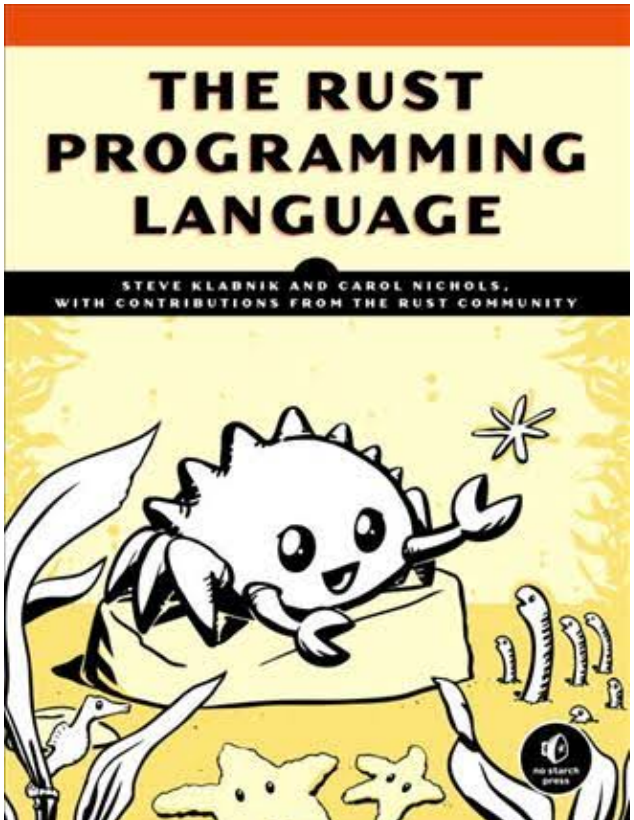
Features of Rust

- Lifetimes and Ownership
  - Key feature for ensuring safety
- Traits as core of object(-like) system
- Variable default is immutability
- Data types and pattern matching
- Type inference
  - No need to write types for local variables
- Generics (aka parametric polymorphism)
- First-class functions
- Efficient C bindings

Rust in the real world

- Firefox Quantum and Servo components
  - https://servo.org
- REmacs port of Emacs to Rust
  - https://github.com/Wilfred/remacs
- Amethyst game engine
  - https://www.amethyst.rs/
- Magic Pocket filesystem from Dropbox
  - https://www.wired.com/2016/03/epic-story-dropboxs-exodus-amazon-cloud-empire/
- OpenDNS malware detection components
- https://www.rust-lang.org/en-US/friends.html

Information on Rust



- Rust book free online
  - https://doc.rust-lang.org/book/
  - **We will follow it in these lectures**
- More references via Rust site
  - https://www.rust-lang.org/en-US/documentation.html
- Rust Playground (REPL)
  - https://play.rust-lang.org/

Installing Rust

- Instructions, and stable installers, here:

  https://www.rust-lang.org/en-US/install.html

- On a Mac or Linux (VM), open a terminal and run

  ```
  curl https://sh.rustup.rs -sSf | sh
  ```

- On Windows, download+run rustup-init.exe

  https://static.rust-lang.org/rustup/dist/i686-pc-windows-gnu/rustup-init.exe

Rust compiler, build system

- Rust programs can be compiled using rustc
  - Source files end in suffix .rs
  - Compilation, by default, produces an executable
    - No –c option
- Preferred: Use the cargo package manager
  - Will invoke rustc as needed to build files
  - Will download and build dependencies
  - Based on a .toml file and .lock file
    - You won’t have to mess with these for this class
  - Like ocamlbuild or dune

Using rustc

- Compiling and running a program

main.rs:

```
fn main() {
    println!("Hello, world!")
}
```

```
% rustc main.rs
% ./main
Hello, world!
%
```

Using cargo

- Make a project, build it, run it

Use `cargo` to run tests, too; will discuss later

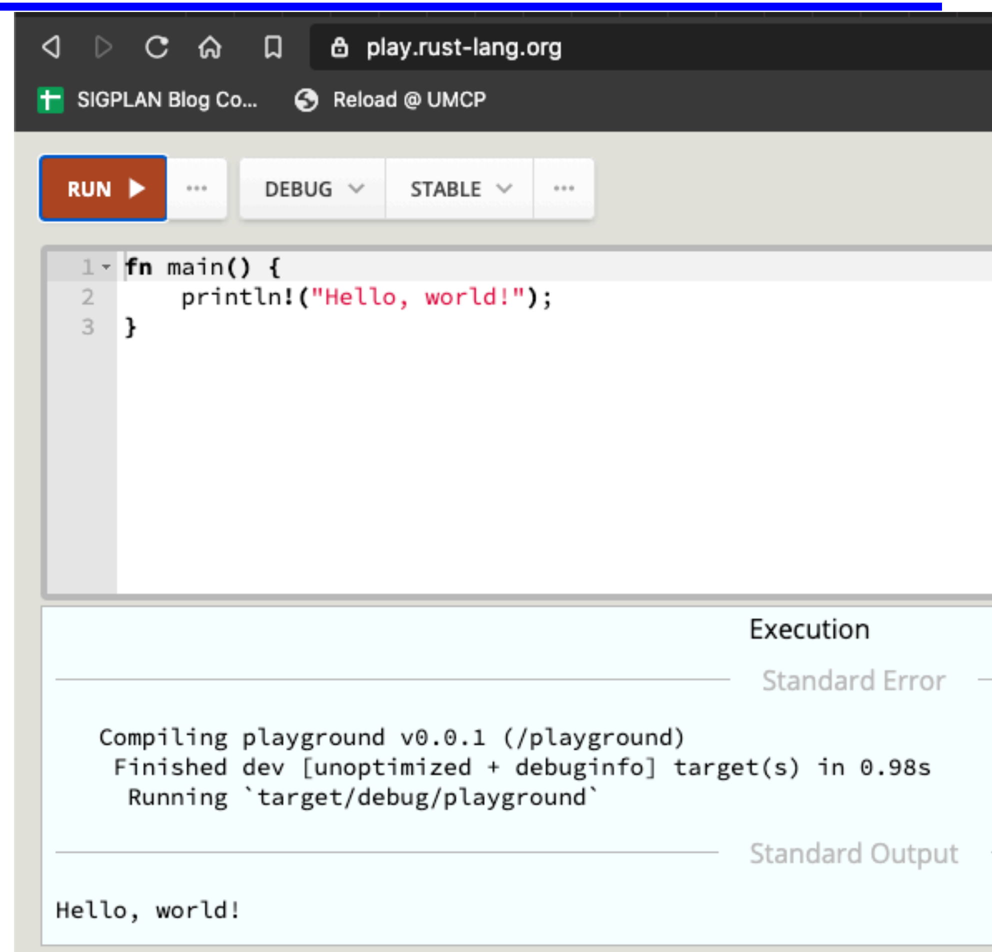
```
% cargo new hello_cargo --bin
% cd hello_cargo
% ls
Cargo.toml   src/
% ls src
main.rs
% cargo build
 Compiling hello_cargo v0.1.0 (file:///…)
 Finished dev [unoptimized + debuginfo] …
% ./target/debug/hello_cargo
Hello, world!
```

```
fn main() {
    println!("Hello, world!”)
}
```

More at https://doc.rust-lang.org/stable/cargo/getting-started/first-steps.html

Rust, interactively

- Rust has no top-level *a la* OCaml or Ruby
- There is an in-browser execution environment
  - https://play.rust-lang.org/



Rust Documentation

- Rust documentation is a good reference, and way to learn
  – https://doc.rust-lang.org/stable/

- This contains links to
  – the Rust Book (on which most of our slides are based),
  – the reference manual, and
  – short manuals on the compiler, cargo, and more

Testing

- In any language, there is the need to test code
- In most languages, testing requires extra libraries:
  - Minitest in Ruby
  - Ounit in Ocaml
  - Junit in Java
- Testing in Rust is a first-class citizen!
  - The testing framework is built into cargo

Unit Testing In Rust

```
#[cfg(test)]
mod tests {
        #[test]
        fn it_works() {
                assert_eq!(my_function(), 4);
        }
}
```

Mark the *module* as containing tests

Mark this function as a test

Unit Testing In Rust

- Unit testing is for local or private functions
  - Put such tests in the same file as your code
- Use **`assert!`** to test that something is true
- Use **`assert_eq!`** to test that two things that implement the **`PartialEq`** trait are equal
  - E.g., integers, booleans, etc.

Integration Testing In Rust

- Integration testing is for APIs and whole programs
- Create a **`tests`** directory
- Create different files for testing major functionality
- Files don't need **`#[cfg(test)]`** or **`mod tests`**
  - But they do still need **`#[test]`** around each function
- Tests refer to code as if it were an external library
  - Declare it as an external library using **`extern crate`**
  - Include the functionality you want to test with **`use`**

Integration Testing In Rust

src/lib.rs

```
pub fn add(a: i32, b: i32) -> i32 {
  a + b
}
```

tests/test_add.rs

```
extern crate my-project-name;
use my-project-name::add;
#[test]
pub fn test_add() {
        assert_eq!(add(1,2), 3));
}
#[test]
pub fn test_negative_add() {
        assert_eq!(add(1,-2), -1));
}
```

Running Tests

- `cargo test` runs all of your tests
- `cargo test` ***s*** runs all tests that contain ***s*** in the name
- By default, console output is hidden
  - Use `cargo test -- --nocapture` to un-hide it

Fun Fact

- The original Rust compiler was written in OCaml
  - Betrays the sentiments of the language's designers!
- Now the Rust compiler is written in … Rust
  - How is this possible? Through a process called bootstrapping:
    - The first Rust compiler written in Rust is compiled by the Rust compiler written in OCaml
    - Now we can use the binary from the Rust compiler to compile itself
    - We discard the OCaml compiler and just keep updating the binary through self-compilation
    - So don't lose that binary! ☺

Ownership

Memory: the Stack and the Heap

- The stack
  - constant-time, automatic (de)allocation
  - Data size and lifetime must be known at compile-time
    - Function parameters and locals of known (constant) size
- The heap
  - Dynamically sized data, with non-fixed lifetime
    - Slightly slower to access than stack; i.e., via a pointer
  - **GC**: automatic deallocation, adds space/time overhead
  - **Manual** deallocation (C/C++): low overhead, but non-trivial opportunity for **devastating bugs**
    - Dangling pointers, double free – instances of memory corruption

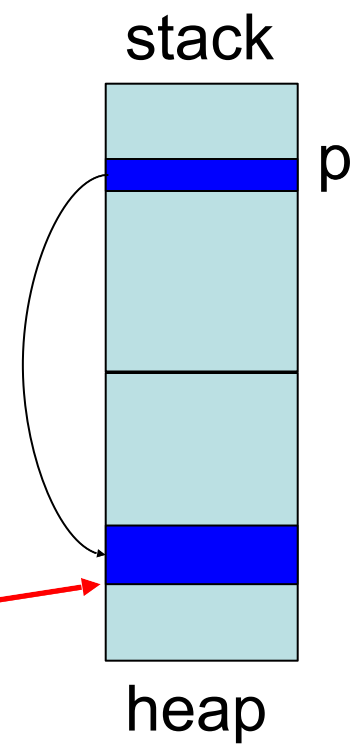
Memory: the Stack and the Heap

// C

```
char *p = malloc(10)
…
free(p);
```

// Java

```
String p = new String("rust");
…
p = null;
```
//GC will collect later



p is deleted from stack when the function terminates

Memory Management Errors

- May forget to free memory (memory leak)

```
{ int *x = (int *) malloc(sizeof(int)); }
```

- May retain ptr to freed memory (dangling pointer)

```
{ int *x = ...malloc();
  free(x);
  *x = 5; /* oops! */
}
```

- May try to free something twice (double free)

```
{ int *x = ...malloc(); free(x); free(x); }
```

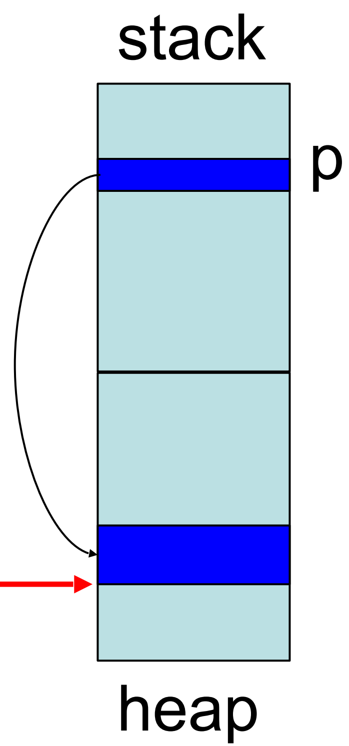
  - This may corrupt the memory management data structures
    - E.g., the memory allocator maintains a free list of space on the heap that’s available

GC-less Memory Management, Safely

- Rust's heap memory managed **without GC**
- Type checking ensures **no dangling pointers** or **double frees**
  - unsafe idioms are disallowed
  - memory leaks *not* prevented (not a safety problem)
- Key features of Rust that ensure safety: **ownership** and **lifetimes**
  - Data has a single owner. Immutable aliases OK, but mutation only via owner or single mutable reference
  - How long data is alive is determined by a lifetime

Memory: the Stack and the Heap

```
// Rust
let p = String::from("hello");
...
```



- Deleted when the owner p is out of scope.
- No manual free, no GC

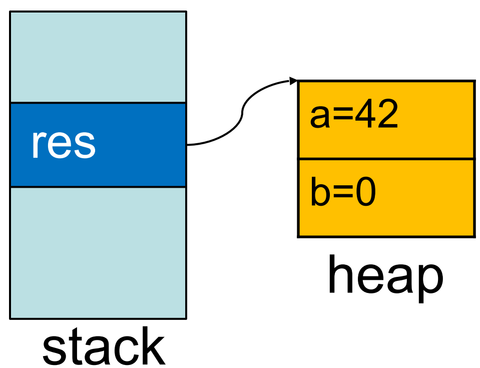
p is deleted from stack when the function terminates

Ownership

Only one "owner" of an object

- When the "owner" of the object goes out of scope, its data is automatically freed. No Garbage collection
- Can not access object beyond its lifetime (checked at compile-time)

```
fn foo() {
   let mut res = Box::new(Pair {
                                     a: 0,
                                     b: 0
                 });
  res.a = 42;
}
```



Rules of Ownership

1. Each value in Rust has a variable that's its owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped (freed)

`String`: Dynamically sized, mutable data

```
{
 let mut s = String::from("hello");

 s.push_str(", world!"); //appends to s

 println!("{}", s);
} //s's data is freed by calling s.drop()
```

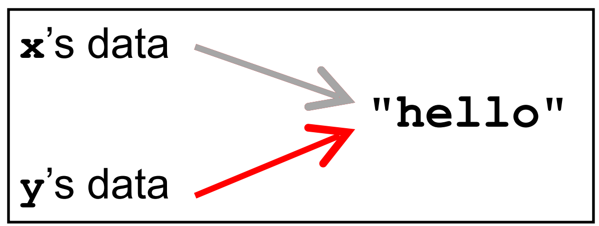
- **`s`** is the *owner* of this data
  - When **`s`** goes out of scope, its **`drop`** method is called, which frees the data

Assignment Transfers Ownership

- Heap allocated data is copied by reference

```
let x = String::from("hello");
let y = x; //x moved to y
```

  - Both **x** and **y** point to the same underlying data



- A move leaves only one owner: **y**

*Prevents double **free()**!*

```
println!("{}, world!", y); //ok
println!("{}, world!", x); //fails
```

Deep Copying Retains Ownership

- Make clones (copies) to avoid ownership loss

```
let x = String::from("hello");
let y = x.clone(); //x no longer moved
println!("{}, world!", y); //ok
println!("{}, world!", x); //ok
```

- Primitives copied automatically
  - **`i32`**, **`char`**, **`bool`**, **`f32`**, tuples of these types, etc.

```
let x = 5;
let y = x;
println!("{} = 5!", y); //ok
println!("{} = 5!", x); //ok
```

    - These have the **`Copy`** trait; more on traits later

Ownership Transfer in Function Calls

```
fn main() {
  let s1 = String::from("hello");
  let s2 = id(s1); //s1 moved to arg
  println!("{}",s2); //id's result moved to s2
  println!("{}",s1); //fails
}

fn id(s:String) -> String {
 s // s moved to caller, on return
}
```

- On a call, ownership passes from:
  - argument to called function's parameter
  - returned value to caller's receiver

References and Borrowing

- Create an alias by making a reference
  - An explicit, non-owning pointer to the original value
  - Called borrowing. Done with `&` operator
- References are immutable by default

```
fn main() {
 let s1 = String::from("hello");
 let len = calc_len(&s1); //lends pointer
 println!("the length of '{}' is {}",s1,len);
}
fn calc_len(s: &String) -> usize {
 s.push_str("hi"); //fails! refs are immutable
 s.len()     // s dropped; but not its referent
}
```

Rules of References

1. At any given time, you can have *either* but not both of
   - One mutable reference
   - Any number of immutable references

2. References must always be valid (pointed-to value not dropped)

Borrowing and Mutation

- Make immutable references to mutable values
  - Shares read-only access through owner and borrowed references
    - Same for immutable values
  - Mutation disallowed on original value until borrowed reference(s) dropped

```
{ let mut s1 = String::from("hello");
  { let s2 = &s1;
    println!("String is {} and {}",s1,s2); //ok
    s1.push_str(" world!"); //disallowed
    println!("{}", s2);
  } //drops s2
  s1.push_str(" world!"); //ok
  println!("String is {}",s1);}//prints updated s1
```

Mutable references

- To permit mutation via a reference, use **`&mut`**
  – Instead of just **`&`**
  – But only OK for mutable variables

```
let mut s1 = String::from(“hello”);
{ let s2 = &s1;
  s2.push_str(“ there”);//disallowed; s2 immut
} //s2 dropped
let s3 = &mut s1; //ok since s1 mutable
s3.push_str(“ there”); //ok since s3 mutable
println!(”String is {}”,s3); //ok
```

Quiz 1: What does this evaluate to?

```
{ let mut s1 = String::from("Hello!");
    {
      let s2 = &s1;
      s2.push_str("World!");
      println!("{}", s2)
    }
}
```

A. "Hello!"

B. "Hello! World!"

C. Error

D. "Hello!World!"

Quiz 1: What does this evaluate to?

```
{ let mut s1 = String::from("Hello!");
    {
      let s2 = &s1;
      s2.push_str("World!");
      println!("{}", s2)
    }
}
```

A. "Hello!"

B. "Hello! World!"

C. **Error; s2 is not mut**

D. "Hello!World!"

Quiz 2: What is printed?

```
fn foo(s: &mut String) -> usize{
    s.push_str("Bob");
    s.len()
}
fn main() {
    let mut s1 = String::from("Alice");
    println!("{}",foo(&mut s1))
}
```

A. 0

B. 8

C. Error

D. 5

Quiz 2: What is printed?

```
fn foo(s: &mut String) -> usize{
    s.push_str("Bob");
    s.len()
}
fn main() {
    let mut s1 = String::from("Alice");
    println!("{}",foo(&mut s1))
}
```

A. 0

B. 8

C. Error

D. 5

Ownership and Mutable References

- Can make only one mutable reference
- Doing so blocks use of the original
  - Restored when reference is dropped

```
let mut s1 = String::from("hello");
{ let s2 = &mut s1; //ok
  let s3 = &mut s1; //fails: second borrow
  s1.push_str(" there"); //fails: second borrow
} //s2 dropped; s1 is first-class owner again
s1.push_str(" there"); //ok
println!("String is {}",s1); //ok
```

implicit borrow
(**`self`** is a reference)

Immutable and Mutable References

- Cannot make a mutable reference if immutable references exist
  - Holders of an immutable reference assume the object will not change from under them!

```
let mut s1 = String::from("hello");
{ let s2 = &s1; //ok: s2 is immutable
  let s3 = &s1; //ok: multiple imm. refs allowed
  let s4 = &mut s1; //fails: imm ref already
} //s2-s4 dropped; s1 is owner again
s1.push_str(" there"); //ok
println!("String is {}",s1); //ok
```

Aside: Generics and Polymorphism

- Rust has support like that of Java and OCaml
  - Example: The `std` library defines `Vec<T>` where `T` can be instantiated with a variety of types
    - `Vec<char>` is a vector of characters
    - `Vec<&str>` is a vector of string slices

- You can define polymorphic functions, too
  - Rust:

```
fn id<T>(x:T) -> T { x }
```

  - Java:

```
static <T> T id(T x) { return x; }
```

  - Ocaml:

```
let id x = x
```

- More later…

Dangling References

- References must always be to valid memory
  - Not to memory that has been dropped

```
fn main() {
  let ref_invalid = dangle();
  println!("what will happen … {}",ref_invalid);
}
fn dangle() -> &String {
  let s1 = String::from("hello");
  &s1
} // bad! s1's value has been dropped
```

  - Rust will disallow this using a concept called lifetimes
    - A lifetime is a type-level parameter that names the scope in which the data is valid

Lifetimes: Preventing Dangling Refs

- Another way to view our prior example

```
{
  let r; // deferred init
  {
    let x = 5;
    r = &x;
  }
  println!("r: {}",r); //fails
}
```

r's lifetime *'a*

x's lifetime *'b*

Issue:
r $\longleftarrow$ x but *'a* $\not\leq$ *'b*

- The Rust type checker observes that **x** goes out of scope while **r** still exists
  - A **lifetime** is a *type variable* that identifies a scope
  - **r**'s lifetime 'a exceeds **x**'s lifetime 'b

Lifetimes and Functions

String slice
(more later)

- Lifetime of a reference not always explicit
  - E.g., when passed as an argument to a function

```
fn longest(x:&str, y:&str) -> &str {
  if x.len() > y.len() { x } else { y }
}
```

  - What could go wrong here?

```
{ let x = String::from("hi");
  let z;
  { let y = String::from("there");
    z = longest(&x,&y); //will be &y
  } //drop y, and thereby z
  println!("z = {}",z);//yikes!
}
```

Quiz 3: What is printed?

```
{ let mut s = &String::from("s");
      {
          let y = String::from("y");
          s = &y;
      }
      println!("s: {}",s);
}
```

A. dog

B. hi

C. Error – y is immutable

D. Error – y dropped while still borrowed

Quiz 3: What is printed?

```
{ let mut s = &String::from("s");
      {
            let y = String::from("y");
            s = &y;
      }
      println!("s: {}",s);
}
```

A. dog
B. hi
C. Error – y is immutable
D. Error – y dropped while still borrowed

Lifetime Parameters

- Each reference to a value of type *t* has a lifetime parameter
  - `&t` (and `&mut t`) – lifetime is implicit
  - `&'a t` (and `&'a mut t`) – lifetime `'a` is explicit
- Where do the lifetime names come from?
  - When left implicit, they are generated by the compiler
  - Global variables have lifetime `'static`
- Lifetimes can also be generic

```
fn longest<'a>(x:&'a str, y:&'a str) -> &'a str {
  if x.len() > y.len() { x } else { y }
}
```

  - Thus: **x** and **y** must have the same lifetime, and the returned reference shares it

Lifetimes FAQ

- When do we use explicit lifetimes?
  - When more than one var/type needs the same lifetime (like the **`longest`** function)
- How does lifetime subsumption work?
  - If lifetime **`'a`** is longer than **`'b`**, we can use **`'a`** where **`'b`** is expected; can require this with **`'b: 'a`**.
    - Permits us to call **`longest(&x,&y)`** when **`x`** and **`y`** have different lifetimes, but one outlives the other
  - Just like subtyping/subsumption in OO programming
- Can we use lifetimes in data definitions?
  - Yes; we will see this later when we define **`structs`**, **`enums`**, etc.

Lifetimes FAQ

- How do I tell the compiler exactly which lines of code 'a covers?
  - You can't. The compiler will figure it out.

Recap: Rules of References

1. At any given time, you can have *either* but not both of
   – One mutable reference
   – Any number of immutable references

2. References must always be valid
   – A reference must never outlive its referent