Memory Safety in Rust

Ryan Eberhardt and Armin Namavari
April 7, 2020

Last lecture Ryan told us the bad news about C and C++…

This lecture, I’m going to tell you how Rust addresses some of those issues

Disclaimer: you can still write buggy Rust programs! Rust just makes it harder to make certain kinds of mistakes!

Why is it so easy to screw up in C?

A Memory Exercise

- You should have completed this before class today!
  - We thank Will Crichton for this exercise and for giving us permission to use it in this class!
- Discuss your answers to the exercise in groups (we'll assign you to different breakout rooms in Zoom)

Dangling Pointers

```
Vec* vec_new() {
  Vec vec;
  vec.data = NULL;
  vec.length = 0;
  vec.capacity = 0;
  return &vec; // OOF
}
```

Double Frees

```
void main() {
  Vec* vec = vec_new();
  vec_push(vec, 107);

  int* n = &vec->data[0];
  vec_push(vec, 110);
  printf("%d\n", *n);

  free(vec->data);
  vec_free(vec); // YIKES
}
```

Iterator Invalidation

```
void main() {
  Vec* vec = vec_new();
  vec_push(vec, 107);

  int* n = &vec->data[0];
  vec_push(vec, 110);
  printf("%d\n", *n); // :(

  free(vec->data);
  vec_free(vec);
}
```

Memory Leaks

```
void vec_push(Vec* vec, int n) {
  if (vec->length == vec->capacity) {
    int new_capacity = vec->capacity * 2;
    int* new_data = (int*) malloc(new_capacity);
    assert(new_data != NULL);

    for (int i = 0; i < vec->length; ++i) {
      new_data[i] = vec->data[i];
    }

    vec->data = new_data; // OOP: we forget to free the old data
    vec->capacity = new_capacity;
  }

  vec->data[vec->length] = n;
  ++vec->length;
}
```

It is Incredibly Hard to Reason about Programs

- Sometimes impossible (see CS 103, 154)
- Sometimes more than impossible*
- How do we get around this?
- A: The language and the compiler!

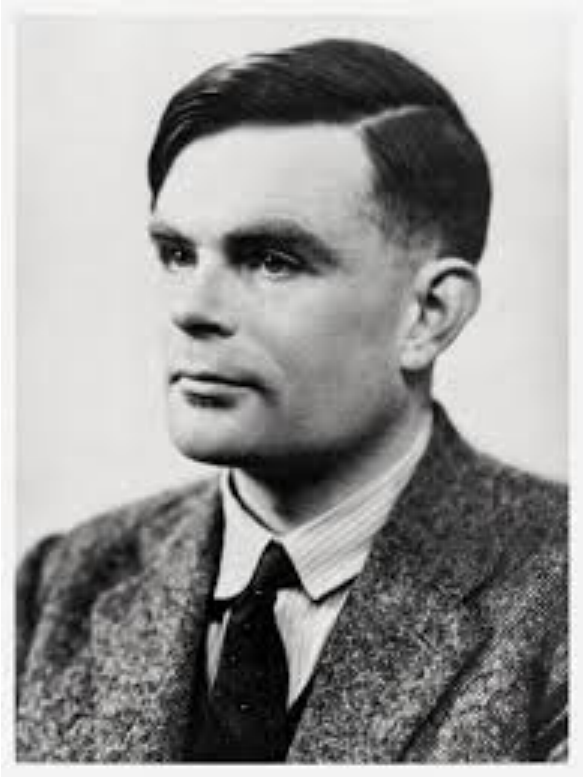

Image Source: https://www.newyorker.com/culture/culture-desk/living-in-alan-turings-future

The Language and the Compiler

- In order to make it easier to reason about programs, Rust needs to place some restrictions on the programs you can write.
    - This makes it difficult (sometimes impossible) to write certain programs in **safe** Rust (we will talk about unsafe Rust later in the course).
- A lot of the cool guarantees we get from Rust come the **checks its compiler performs**
- Rust can sometimes exceed the performance of C because of **compiler optimizations.**
- If you want to delve deeper into these topics, be sure to take CS 242 (Programming Languages) and CS 143 (Compilers) as well as their follow-ons — these particular topics are outside of the scope of CS110L, but let us know if you'd like us to point you to relevant resources for learning more.

Dangling Pointers

```
Vec* vec_new() {
  Vec vec;
  vec.data = NULL;
  vec.length = 0;
  vec.capacity = 0;
  return &vec; // OOF
}
```

Wouldn't it be nice if the compiler realized that vec "lives" within those two curly braces and therefore its address shouldn't be returned from the function?

Double Frees

```
void main() {
  Vec* vec = vec_new();
  vec_push(vec, 107);

  int* n = &vec->data[0];
  vec_push(vec, 110);
  printf("%d\n", *n);

  free(vec->data);
  vec_free(vec); // YIKES
}
```

Wouldn't it be nice if the compiler enforced that once free is called on a variable, that variable can no longer be used?

Iterator Invalidation

```
void main() {
  Vec* vec = vec_new();
  vec_push(vec, 107);

  int* n = &vec->data[0];
  vec_push(vec, 110);
  printf("%d\n", *n); // :(

  free(vec->data);
  vec_free(vec);
}
```

Wouldn't it be nice if the compiler stopped us from modifying the data n was pointing to (as it does in vec_push)?

Memory Leaks

```
void vec_push(Vec* vec, int n) {
  if (vec->length == vec->capacity) {
    int new_capacity = vec->capacity * 2;
    int* new_data = (int*) malloc(new_capacity);
    assert(new_data != NULL);

    for (int i = 0; i < vec->length; ++i) {
      new_data[i] = vec->data[i];
    }

    vec->data = new_data; // OOP
    vec->capacity = new_capacity;
 }

  vec->data[vec->length] = n;
  ++vec->length;
}
```

Wouldn't it be nice if the compiler noticed when a piece of heap data no longer had anything pointing to it? (and so then it could safely be freed?)

Pause

How does Rust prevent us from making the errors we just saw?

Ownership

- The reason you ran into trouble when decomposing your code!
- From the Rust Book:

Ownership Rules

First, let's take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate them:

- Each value in Rust has a variable that's called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

Controlling references to resources is a broader idea in systems programming that isn’t unique to Rust

Ownership in Context

```
fn main() {
    let s: String = "im a lil string”.to_string();
    let u = s;
    println!("{}", s); // println!(“{}”, u) compiles just fine!
}
```

Note: you can copy/paste this code and run it in your browser @ https://play.rust-lang.org/ !

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:7:20
  |
5 |     let s: String = "im a lil string".to_string();
  |         - move occurs because `s` has type `std::string::String`, which does not implement the `Copy` trait
6 |     let u = s;
  |             - value moved here
7 |     println!("{}", s);
  |                    ^ value borrowed here after move
```

Ownership in Context

```
fn om_nom_nom(s: String) {
    println!("I have consumed {}", s);
}

fn main() {
    let s: String = "im a lil string".to_string();
    om_nom_nom(s);
    println!("{}", s);
}
```

```
error[E0382]: borrow of moved value: `s`
 --> src/main.rs:8:20
  |
6 |     let s: String = "im a lil string".to_string();
  |         - move occurs because `s` has type `std::string::String`, which does not implement the `Copy` trait
7 |     om_nom_nom(s);
  |                - value moved here
8 |     println!("{}", s);
  |                    ^ value borrowed here after move
```

With great power comes great responsibility

```
fn om_nom_nom(s: String) {
    println!("I have consumed {}", s);
}

fn main() {
    let s: String = "im a lil string".to_string();
    om_nom_nom(s);
    println!("{}", s);
}
```

- Each “owner” has the responsibility to clean up after itself
- When you move `s` into `om_nom_nom`, `om_nom_nom` becomes the owner of `s`, and it will free `s` when it’s no longer needed in that scope
    - Technically the `s` parameter in `om_nom_nom` become the owner
- That means you can no longer use it in main!

An Exception to the Syntax: Copying

Given what we just saw, how can the following be valid syntax?

```
fn om_nom_nom(n: u32) {
    println!("{} is a very nice number", n);
}

fn main() {
    let n: u32 = 110;
    let m = n;
    om_nom_nom(n);
    om_nom_nom(m);
    println!("{}", m + n);
}
```

```
Output:
110 is a very nice number
110 is a very nice number
220
```

Wait a minute… that seems restrictive and must make it really hard to write code!

Thought experiment

- Say you have a group of lawyers that are reviewing and signing a contract over Google Docs
  - Is this realistic? Nope :) But just pretend!
- What are some ground rules we'd need to set in order to avoid chaos?
  - If someone modifies the contract before everyone else reviews/signs it, that's fine
  - But if someone modifies the contract *while* others are reviewing it, people might miss changes and think they're signing a contract that says something else
  - We should allow a single person to modify, *or* everyone to read, but not both

Borrowing Intuition

- I should be able to have as many “const” pointers to a piece of data that I like
- However if I have a “non-const” pointer to a piece of data at the same time, this could invalidate what the other const pointers are viewing (e.g. they can become dangling pointers…)
- If I have at most one “non-const” pointer at any given time, this should be OK.

Borrowing

- We can have multiple shared (immutable) references at once (with no mutable references) to a value.
- We can have only one mutable reference at once (no shared references to it)
- This is a paradigm that pops up a lot in systems programming, especially when you have “readers” and “writers.” In fact, you’ll see it in CS110 once you start talking about threading and concurrency.

Lifetimes

- The lifetime of a value starts when it’s created and ends the last time it’s used
- Rust doesn’t let you have a reference to a value that lasts longer than the value’s lifetime
- Rust computes lifetimes at compile time using static analysis (this is often an over-approximation)
- Rust calls the special “drop” function on a value once its lifetime ends (this is essentially a destructor).

Borrowing Example

```
fn change_it_up(s: &mut String) {
    *s = "goodbye".to_string();
}

fn make_it_plural(word: &mut String) {
    word.push('s');
}

fn let_me_see(s: &String) {
    println!("{}", s);
}

fn main() {
    let mut s = "hello".to_string();
    change_it_up(&mut s);
    let_me_see(&s);
    make_it_plural(&mut s);
    let_me_see(&s);
    // let's make it even more plural
    s.push('s'); // does this seem strange?
    let_me_see(&s);
}
```

[-] `pub fn push(&mut self, ch: char)`

Appends the given `char` to the end of this `String`.

Borrowing Example: Vectors

```
fn main() {
    let v = vec![1, 2, 3];
    for i in v.iter_mut(){
        *i = 5;
    }
    for i in v.iter() {
        println!("{}", i);
    }
}
```

```
pub fn iter_mut(&mut self) -> IterMut<T>
```

Returns an iterator that allows modifying each value.

```
error[E0596]: cannot borrow `v` as mutable, as it is not declared as mutable
 --> src/main.rs:3:14
  |
2 |     let v = vec![1, 2, 3];
  |         - help: consider changing this to be mutable: `mut v`
3 |     for i in v.iter_mut(){
  |              ^ cannot borrow as mutable

error: aborting due to previous error
```

Reminder: The ownership and borrowing rules are enforced at compile time!

So what?

- This is a big deal — you only compile the program once, but you can run the executable as many times as you like afterward
- This is essentially making a fixed cost investment in our preprocessing.
- It’s generally desirable to shift checks from runtime to compile time.
  - Generally, there is a tension between security and performance. Rust tries to give you both.
  - Just don’t screw up your compiler :(
  - Many security vulnerabilities pop up from making fancy optimizations

A Reminder: The First Assignment

- Again we just want you to get familiar with the basic syntax so we can talk about fancier concepts next week.
- You'll definitely see ownership and borrowing in action — hopefully seeing it in this context and wrestling with the compiler/borrow checker will solidify your understanding (there's only so much you can get from the lecture by itself)
- Please ask questions on Slack and help each other out!

Additional Resources/Readings

- Ownership and borrowing for visual learners!
- A great resource on iterating over vectors in Rust
- A Medium article about ownership, borrowing, and lifetimes
- CS242 lecture notes — shout out to Will Crichton to providing advice on explaining some of these concepts!
- The Rust book
- Check out sections 4.1 and 4.2 (deeper explanation of lifetimes)