Memory Safety in Rust

Ryan Eberhardt and Julio Ballista
April 6, 2021

Logistics

- Please make sure you're on Slack
- Week 1 exercises due on Thursday
  - Please let us know if you need additional time to complete them
  - A blog post can be substituted for weekly exercises
- Today: What is Rust's "ownership model," and how does it prevent common memory errors?
  - Specifically focusing on memory leaks, double frees, and use-after frees
  - Thursday will show how Rust prevents other sorts of memory errors

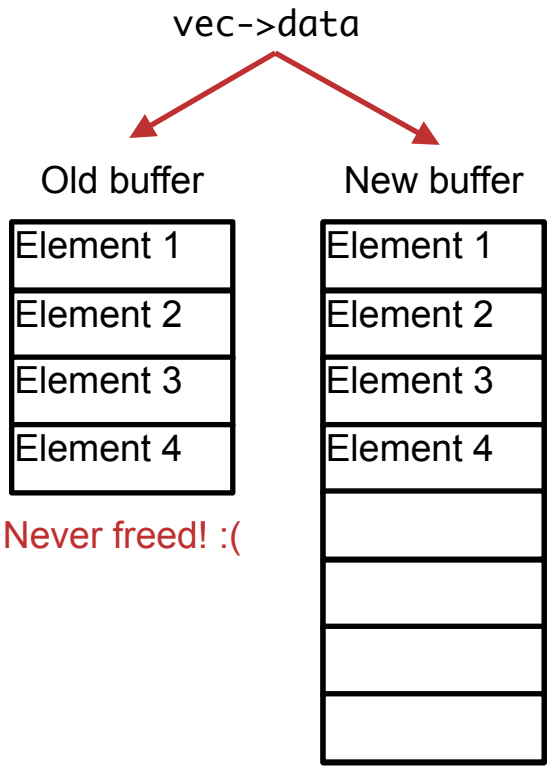
Identifying Memory Errors

A Memory Exercise

- 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)

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;
}
```



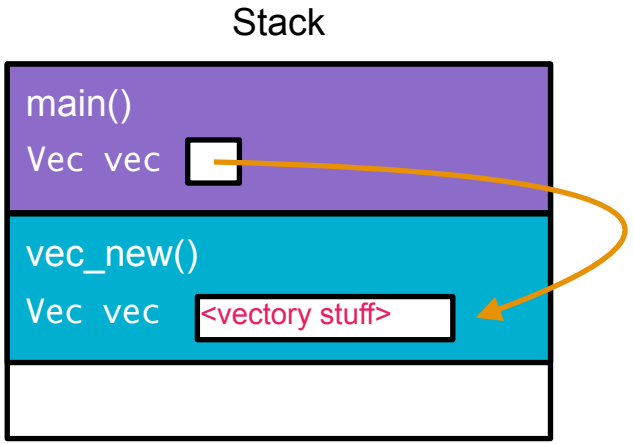
Double Frees

```
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
}
```

Double free: a buffer is `free`d twice. (Sounds innocuous, but can actually lead to Remote Code Execution: take CS 155)
(Here, we `free(vec->data)`, and then call `vec_free`, which does the same thing)

Dangling Pointers

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

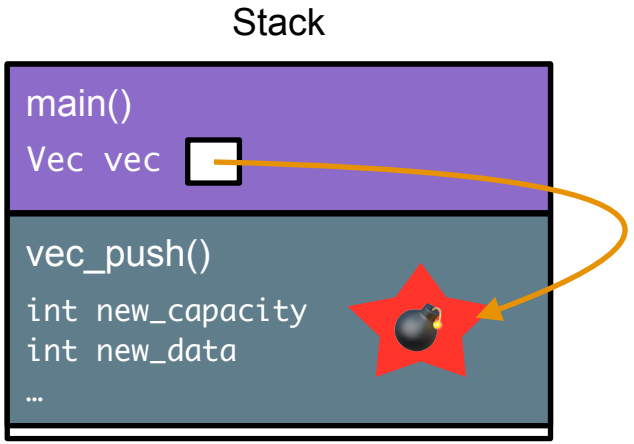


Dangling pointer: A pointer that is referencing memory that isn't there anymore

(Here, `vec` points into the stack frame of `vec_new`, but as soon as `vec_new` returns, that memory is gone)

Dangling Pointers

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

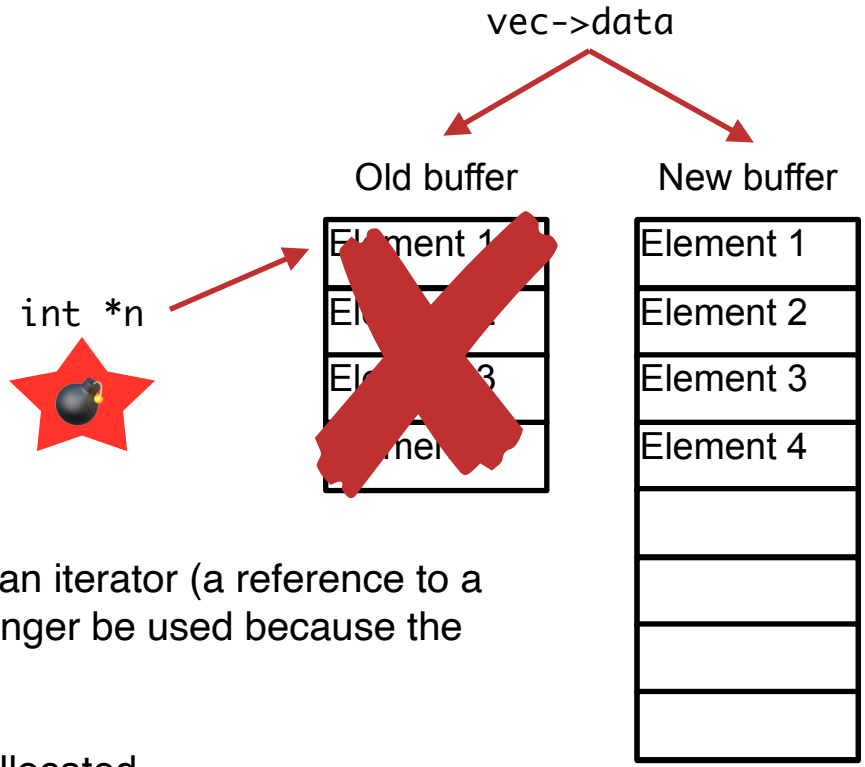


Dangling pointer: A pointer that is referencing memory that isn't there anymore

(Here, `vec` points into the stack frame of `vec_new`, but as soon as `vec_new` returns, that memory is gone)

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);
}
```



Iterator invalidation: a type of dangling pointer where an iterator (a reference to a certain position within an iterable container) can no longer be used because the container was modified

Here, vec_push can cause the vector buffer to be reallocated

What’s wrong with this code?

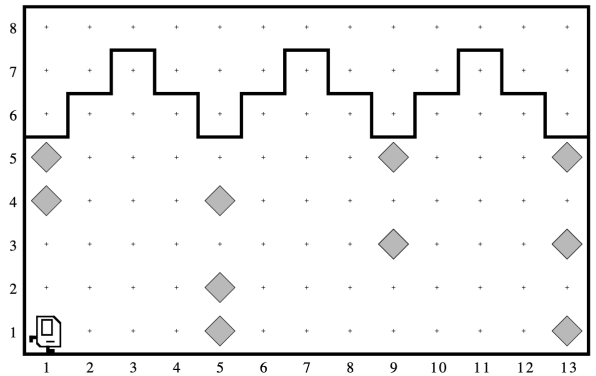
- This code is really bad code (obviously it has a lot of bugs)
- But… Why? What about it *makes* it bad code?

Taking a step back

What makes good code?

Problem 2

Karel has been hired to repair the damage done to the Quad in the 1989 earthquake. In particular, Karel is to repair a set of arches where some of the stones (represented by beepers, of course) are missing from the columns supporting the arches, as follows:



- A natural decomposition for StoneMasonKarel is to implement a repairColumn function, then write:

```
while (frontIsClear()) {
    repairColumn();
    moveToNextColumn();
}
repairColumn();
```

- Many 106A students write repairColumn functions that sometimes end with Karel facing south, and other times end with Karel facing east
- Why is this bad?

What makes good code?

- Pre/postconditions are essential to breaking code into small pieces with well-defined interfaces in between
  - We want to be able to reason about each small piece in isolation
  - Then, if we can verify that preconditions/postconditions are upheld in isolation, we can string together a bunch of components and simply verify that the preconditions/postconditions all fit together without needing to keep the *entire program* in our heads
- It's the programmer's responsibility to make sure the pre/postconditions are upheld

Good memory management

- In any complex program, you'll allocate memory and pass it around the codebase. Where should that memory be freed?
  - If you free too early, other parts of your code might still be using pointers to that memory
  - If you don't free anywhere (or you free in a function that only gets called sometimes), you'll have a memory leak
- Good C/C++ code will clearly define how memory is passed around and "who" is responsible for cleaning it up
- If you read C/C++ code, you'll see notions of "ownership" in the comments, where the "owner" is responsible for the memory

```
/* Get status of the virtual port (ex. tunnel, patch).
 *
 * Returns '0' if 'port' is not a virtual port or has no errors.
 * Otherwise, stores the error string in '*errp' and returns positive errno
 * value. The caller is responsible for freeing '*errp' (with free()).
 *
 * This function may be a null pointer if the ofproto implementation does
 * not support any virtual ports or their states.
 */
int (*vport_get_status)(const struct ofport *port, char **errp);
```

Open vSwitch

```c
/**
 * @note Any old dictionary present is discarded and replaced with a copy of the new one. The
 * caller still owns val is and responsible for freeing it.
 */
int av_opt_set_dict_val(void *obj, const char *name, const AVDictionary *val, int search_flags);
```

ffmpeg

```
/**
 * iscsi_boot_create_target() - create boot target sysfs dir
 * @boot_kset: boot kset
 * @index: the target id
 * @data: driver specific data for target
 * @show: attr show function
 * @is_visible: attr visibility function
 * @release: release function
 *
 * Note: The boot sysfs lib will free the data passed in for the caller
 * when all refs to the target kobject have been released.
 */
struct iscsi_boot_kobj *
iscsi_boot_create_target(struct iscsi_boot_kset *boot_kset, int index,
                         void *data,
                         ssize_t (*show) (void *data, int type, char *buf),
                         umode_t (*is_visible) (void *data, int type),
                         void (*release) (void *data))
{
        return iscsi_boot_create_kobj(boot_kset, &iscsi_boot_target_attr_group,
                                      "target%d", index, data, show, is_visible,
                                      release);
}
EXPORT_SYMBOL_GPL(iscsi_boot_create_target);
```

Linux kernel

Sometimes, custom cleanup functions must be used to free memory. Calling free() on this memory would be a bug!

```
/* Looks up a port named 'devname' in 'ofproto'.  On success, returns 0 and
 * initializes '*port' appropriately. Otherwise, returns a positive errno
 * value.
 *
 * The caller owns the data in 'port' and must free it with
 * ofproto_port_destroy() when it is no longer needed. */
int (*port_query_by_name)(const struct ofproto *ofproto,
                          const char *devname, struct ofproto_port *port);
```

Open vSwitch

Sometimes, custom cleanup functions must be used to free memory. Calling free() on this memory would be a bug!

```
/**
 * dvb_unregister_frontend() - Unregisters a DVB frontend
 *
 * @fe: pointer to &struct dvb_frontend
 *
 * Stops the frontend kthread, calls dvb_unregister_device() and frees the
 * private frontend data allocated by dvb_register_frontend().
 *
 * NOTE: This function doesn't frees the memory allocated by the demod,
 * by the SEC driver and by the tuner. In order to free it, an explicit call to
 * dvb_frontend_detach() is needed, after calling this function.
 */
int dvb_unregister_frontend(struct dvb_frontend *fe);
```

Linux kernel

Ownership can sometimes get extremely complicated, where one part of the codebase is responsible for freeing part of a data structure and a different part of the codebase is responsible for freeing a different part

```
static void mapper_count_similar_free(mapper_t* pmapper, context_t* _) {
    mapper_count_similar_state_t* pstate = pmapper->pvstate;
    slls_free(pstate->pgroup_by_field_names);

    // lhmslv_free will free the keys: we only need to free the void-star values.
    for (lhmslve_t* pa = pstate->pcounts_by_group->phead; pa != NULL; pa = pa->pnext) {
        unsigned long long* pcount = pa->pvvalue;
        free(pcount);
    }
    lhmslv_free(pstate->pcounts_by_group);

    ...
}
```

Miller

Pre/postconditions must be consistently upheld

- It's up to the programmer to make sure to get this right. If you don't uphold the interface, your program is broken
  - Consequences: anything from denial of service (e.g. memory leak) to remote code execution (e.g. double free, use-after free, buffer overflow)
- The compiler cannot help you out
  - Static analyzers can help *sometimes*, but not always (see week 1 exercises)
- Key point: *compiler does not know what your postconditions are,* ***because it's not possible to express in the C language***

Type systems

- The *types* of a programming language are the *nouns* of a spoken language
    - When you talk, what do you talk *about*?
- C type system: numbers, pointers, structs… not much else
    - Extremely simple: can learn most of the C language in half a quarter of CS 107
    - Simple != easy

strdup definition:

```
char *strdup(const char *s);
```

Bad strdup usage:

```
const char *hello = "hello world";
char *duplicate = strdup(hello);
return;
```

Compiler's analysis:

Passes a char to strdup* ✅

*Stores the return value in a char** ✅

Everything looks good! ✅

strdup manpage:

> The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc(3), and can be freed with free(3).

Experienced programmer's analysis

Receives a heap-allocated string from strdup ✅

Returns before freeing the string! 🚫

Type systems

- The *types* of a programming language are the *nouns* of a spoken language
    - When you talk, what do you talk *about*?
- C type system: numbers, pointers, structs… not much else
    - Extremely simple: can learn most of the C language in half a quarter of CS 107
    - Simple != easy

strdup definition:

```
char *strdup(const char *s);
```

strdup manpage:

> The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc(3), and can be freed with free(3).

- The pre/postconditions may be written in comments, but they are not present in the actual code, because the C language does not have a way for them to be expressed
- Consequently: the compiler is unaware of what you're trying to do

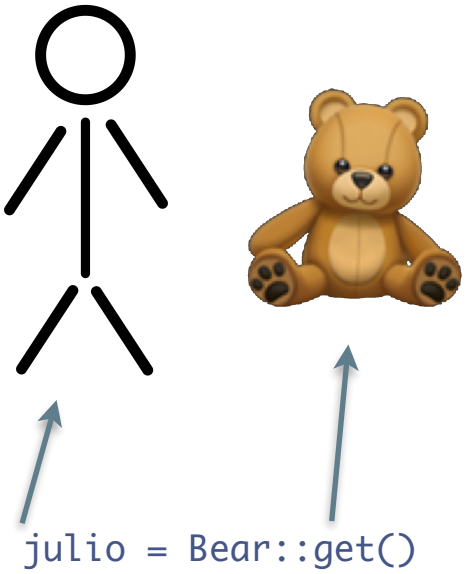
Are there better type systems that we can use to specify our preconditions/postconditions *in the code*?

(implication: if the compiler can understand your pre/postconditions, it can verify that they are met)

(Meet Rust 🦀)

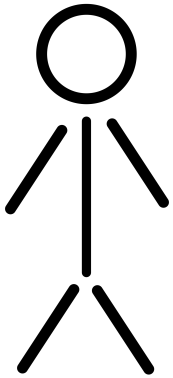
What if Ownership lived in the programming language?

Ownership Visualized



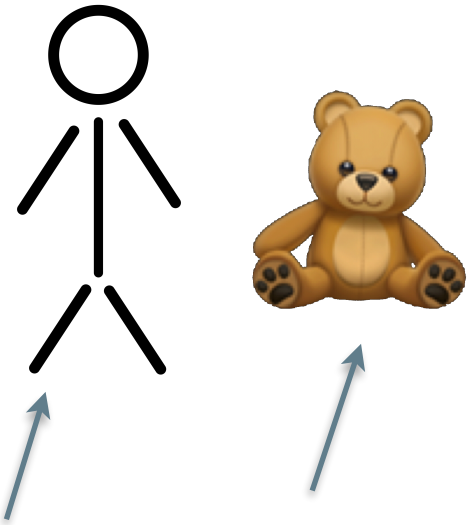
Ownership Visualized

Toys need to be put back when we're done!




```
fn main() {
     let julio = Bear::get();
}
```

Ownership Visualized

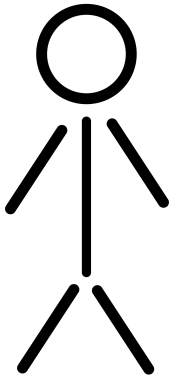


```
let julio = Bear::get();
```

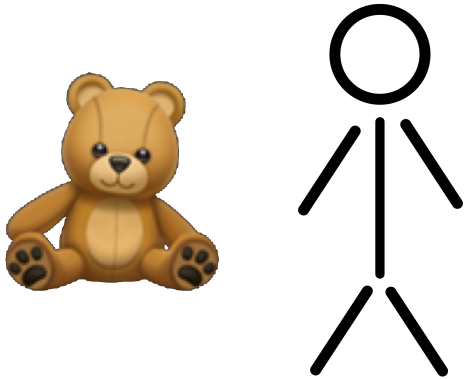
- This 'julio' is the owner of the bear. (I own it)
- I (julio) can do anything I want with the toy, like call functions wrapped within it
- This person is responsible for putting the gift back where they found it before leaving (free the memory!)

Ownership Visualized - What happens now?

```
let julio = Bear::get();
let ryan = julio;
```

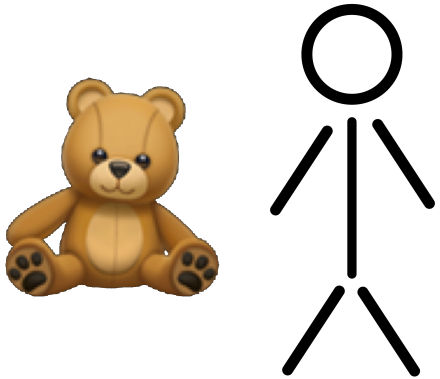


`let julio = ...`



`let ryan = julio;`

Ownership Visualized - What happens now?

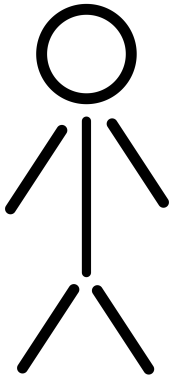


```
let ryan = "julio";
```

- Now, Ryan is the owner of the toy!
- Ryan can do anything he wants with the toy, such as call functions on it.
- Ryan is now responsible for putting the toy back where they found it before leaving (free the memory!)

What about Julio?

Ownership Visualized - What happens now?

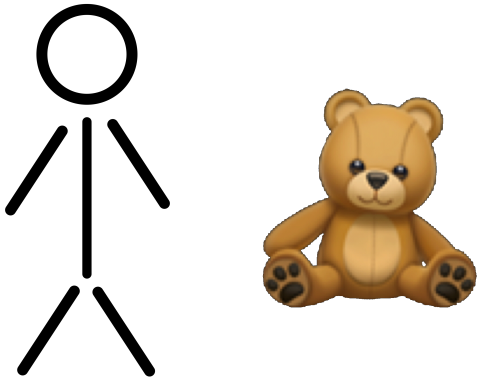


```
let julio = ...
```

- Julio has given the toy to Ryan!!
- No ownership of the toy anymore :(
- Can't do ANYTHING with this string anymore :(
- sad.
- But no longer responsible for putting the toy back :D

Let's see it run!

When else is ownership transferred?



- Function calls can take ownership of variables as well!
- This means that at the end of the function execution, they will be responsible for freeing the toy in memory
- It also means you can no longer use your toy back when the function returns!

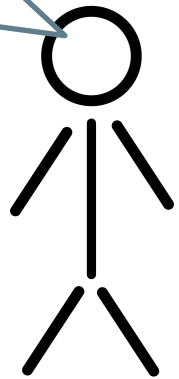
```
let julio = Bear::get()
my_cool_bear_function(julio); <-- This is letting the function own the julio!
/* julio no longer owns the toy D: Compiler wont let you use it! */
```

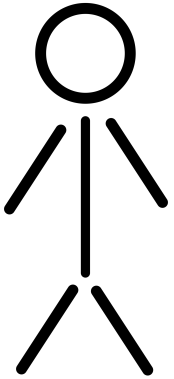
How will I ever decompose code????

Borrowing

```
let julio = Bear::get();
my_cool_bear_function(&julio)
/* The julio variable can still be used here! */
```

Hey, my_cool_bear_function, you could BORROW this toy. Just give it back when you're done!





Thank you, this means you'll have to put the toy back when you're done though!

`let julio = ...`

`my_cool_bear_function(Bear: &Bear)`

Ownership (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.

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

Takeaways

- In Rust, every piece of memory is “owned” by a variable/function
  - This ownership is explicit in the code (as opposed to C/C++, where ownership is usually described in function comments)
  - When the owner goes out of scope, the compiler inserts code to free the memory
- Because of the ownership model, you can’t have:
  - Memory leaks
  - Double frees
  - Use-after-frees
  - Other memory errors — next class!

Next Time + Resources [End]

- What other kinds of references / variables can we create in Rust?
- What does ownership transferring look like in memory?
- More code examples :D
- 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)