Safety in Concurrent Programming with Rust

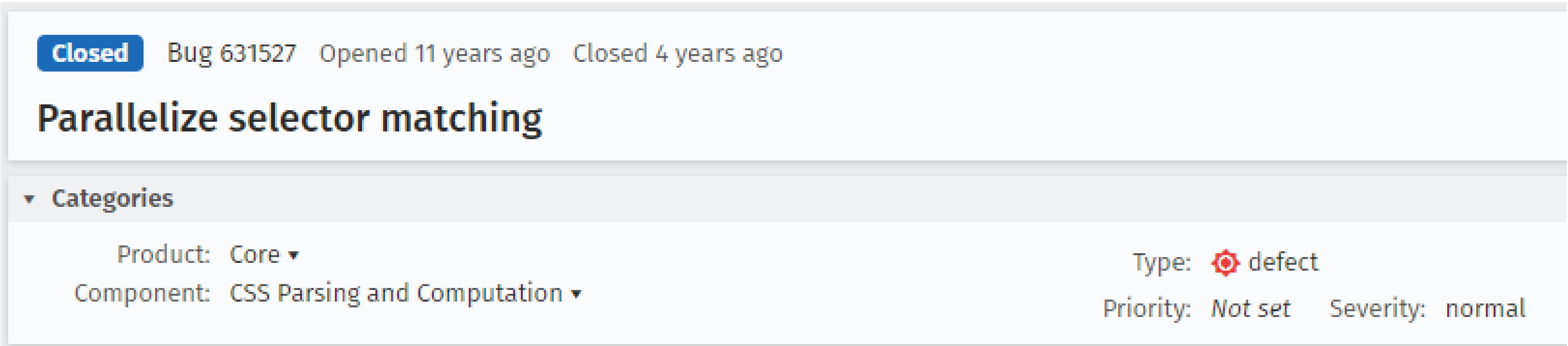
CS3211 Parallel and Concurrent Programming

Outline

- Challenges in concurrent programming
- Introducing Rust
  - Safety, ownership and borrowing
- Concurrency in Rust
  - Threads, Mutex, Atomic
  - Libraries: Crossbeam, Rayon

Challenges in concurrent programming

- Parallelizing … anything is a daunting task
  - The goal is to make things faster
  - Many times, parallelizing is done by just adding another instance that does the same work
- Race conditions, data races, deadlocks, starvation
- Unsafe usage of memory in C/C++
  - Use after free (UAF)
  - Double free



Fearless Concurrency in Rust

- Rust was initiated with two goals in mind:
  - Safety in system programming
  - Painless concurrency

Rust nowadays

- Strong safety guarantees
  - No seg-faults, no data races, expressive type system
- Without compromising on performance
  - No garbage collector, no runtime
  - Same level of performance as C/C++
- Goal
  - Confident, productive systems programming

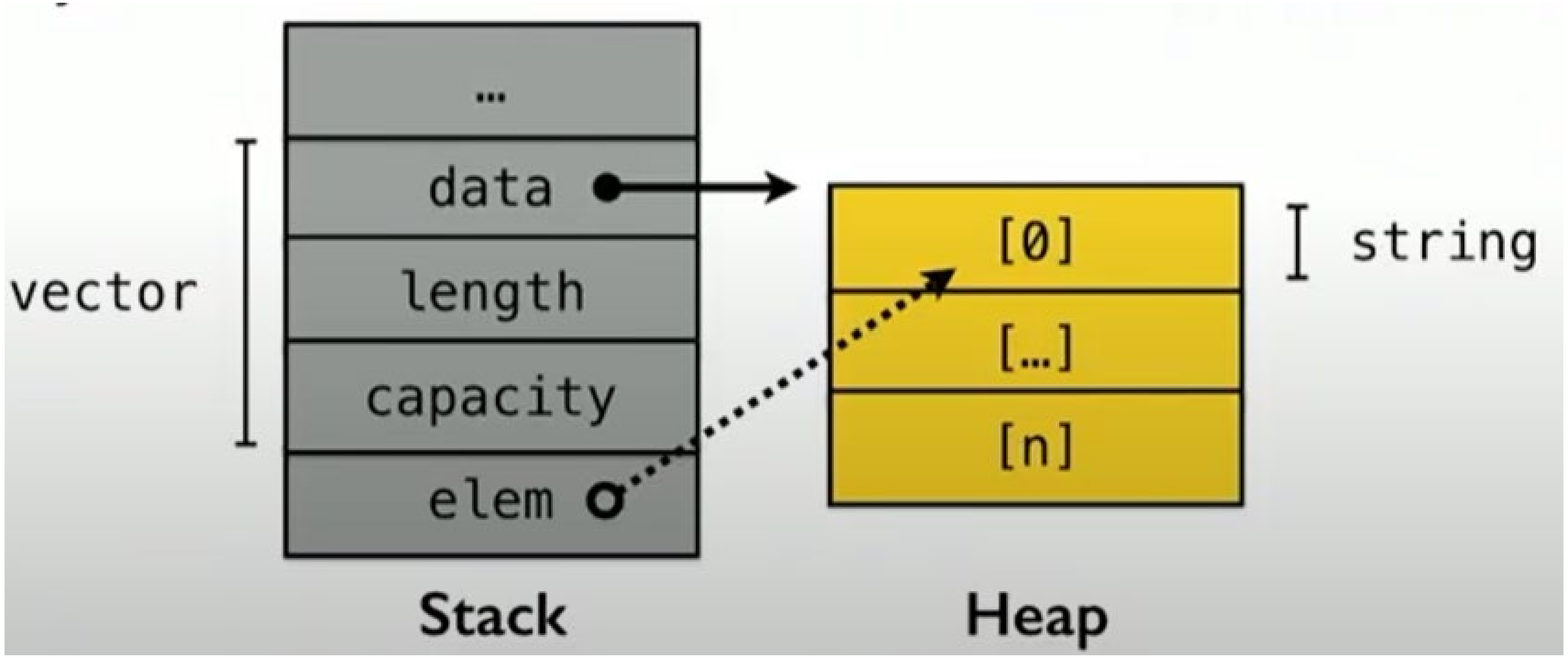
Rust

- Rustup – to install your rust tools
- Rustc – the Rust compiler
- Cargo
  - Calls the compiler – rustc
  - TOML (Tom’s Obvious, Minimal Language) format for the configuration file
- Packages, crates, modules
  - A package is one or more crates that provide a set of functionality
  - A crate is a binary or library
  - Modules are used to organize code within a crate into groups
    - Privacy control

C++ is unsafe

- Vector is freed when we exit the scope

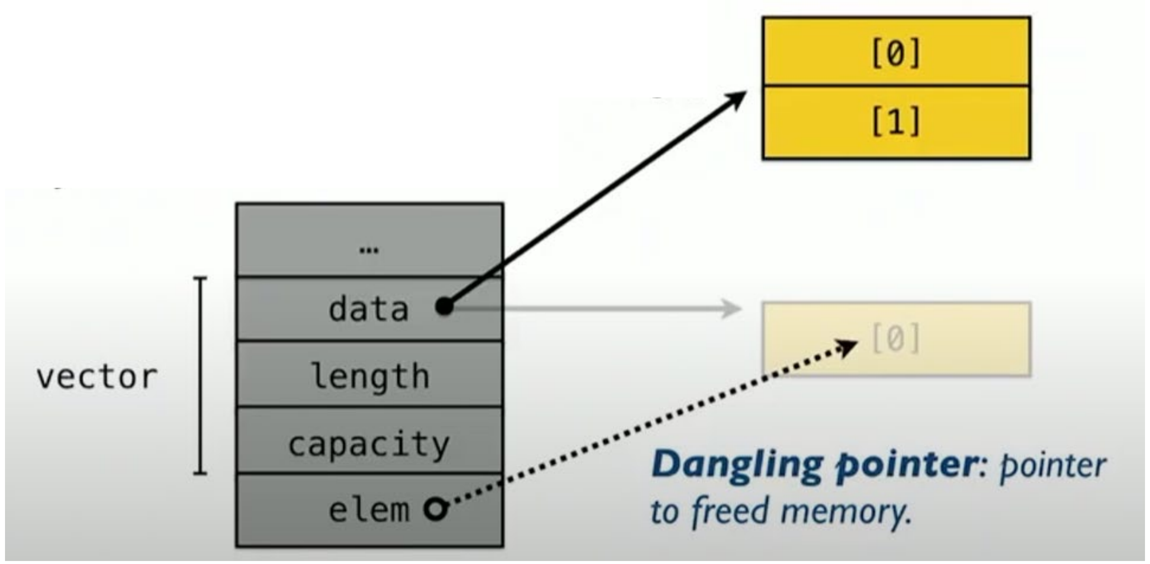
```cpp
void example() {
    vector<string> vector;
    ...
    auto& elem = vector[0];
    ...
}
```



C++ is unsafe

- Dangling pointers issues

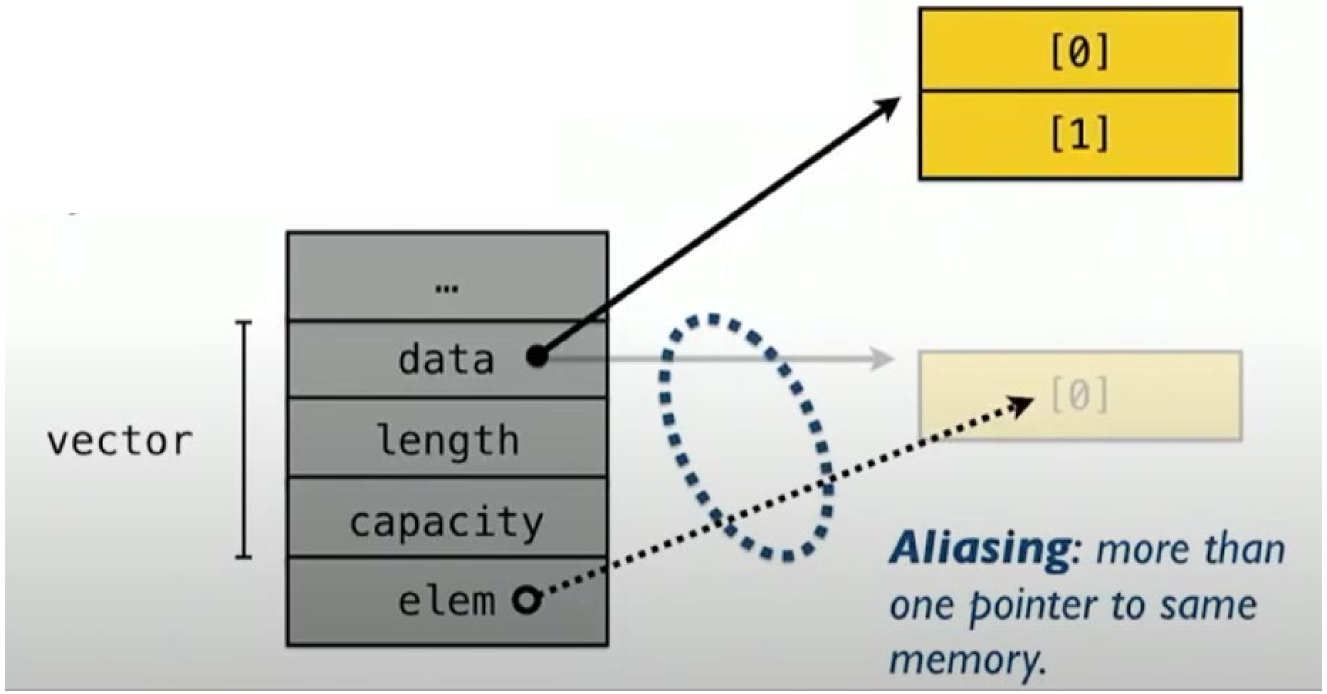
```
void example() {
    vector<string> vector;
    ...
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
}
```



C++ is unsafe

- Aliased pointers – pointers that point to the same chunk of memory
  - `elem` and `vector[0]`
- Mutation – changing a pointer
- Aliasing + mutation – changing (modifying) pointers that point same chunk of memory

```cpp
void example() {
    vector<string> vector;
    ...
    auto& elem = vector[0];
    vector.push_back(some_string);
    cout << elem;
}
```



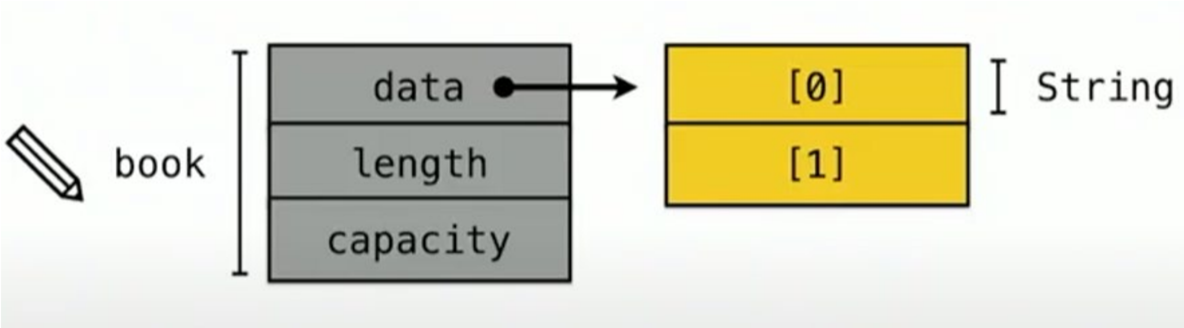
Solution

- Ownership and borrowing
  - Prevent simultaneous mutation and aliasing
- No runtime like in C++
- Memory safety like in garbage collected languages
- No data races like in …Rust

Ownership (1)

- Lines 2-4:
  - Vector book is initialized
  - Owner: main function

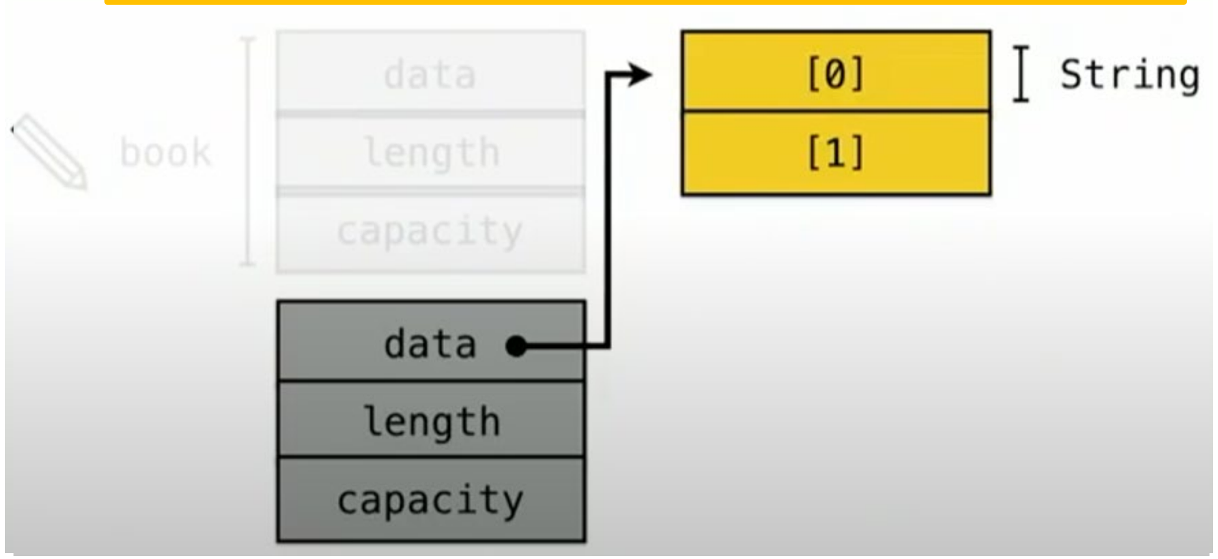
```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(book);
6      // a second call to publish would
7      // generate a compilation error
8      // publish(book);
9  }
10 fn publish(book: Vec<String>) {
11     ...
12 }
```



Ownership (2)

- Line 5: give ownership to publish
  - Pass the without &
- Runtime
  - Copy over the fields from `main`'s stack to `publish`'s stack
  - Forget about the first book in `main`
- Line 11: runs the destructor for book

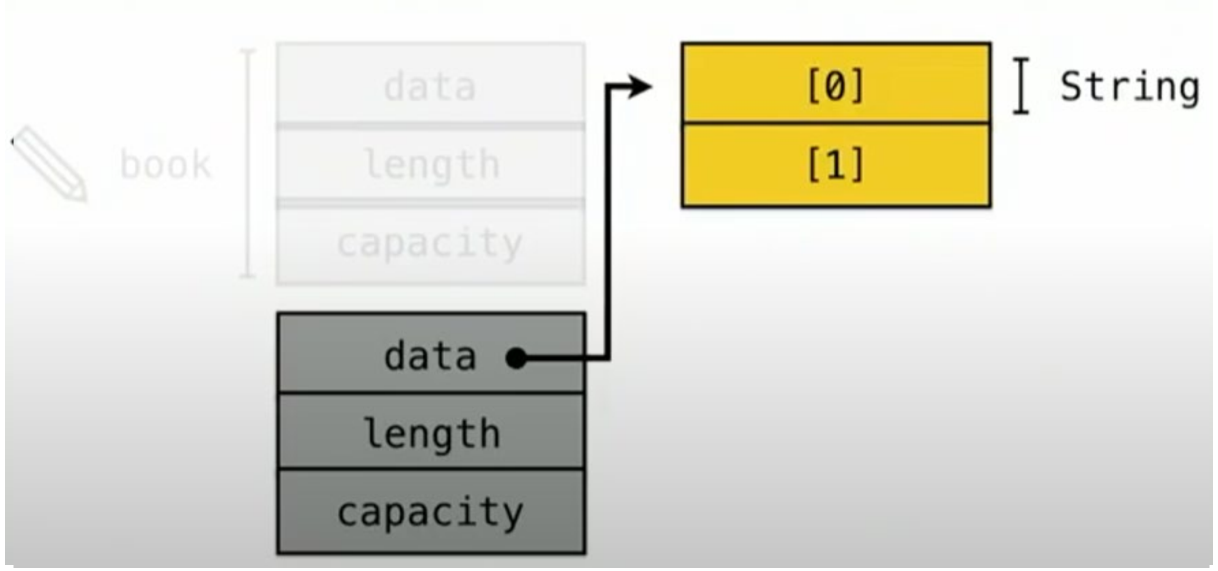
```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(book);
6      // a second call to publish would
7      // generate a compilation error
8      // publish(book);
9  }
10 fn publish(book: Vec<String>) {
11     ...
12 }
```



Ownership (3)

- Line 5: give ownership to publish
- Line 8: compilation error
  - Error: use of moved value book

```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(book);
6      // a second call to publish would
7      // generate a compilation error
8      // publish(book);
9  }
10 fn publish(book: Vec<String>) {
11     ...
12 }
```



Ownership does not allow aliasing!

Rust ownership compared to C++

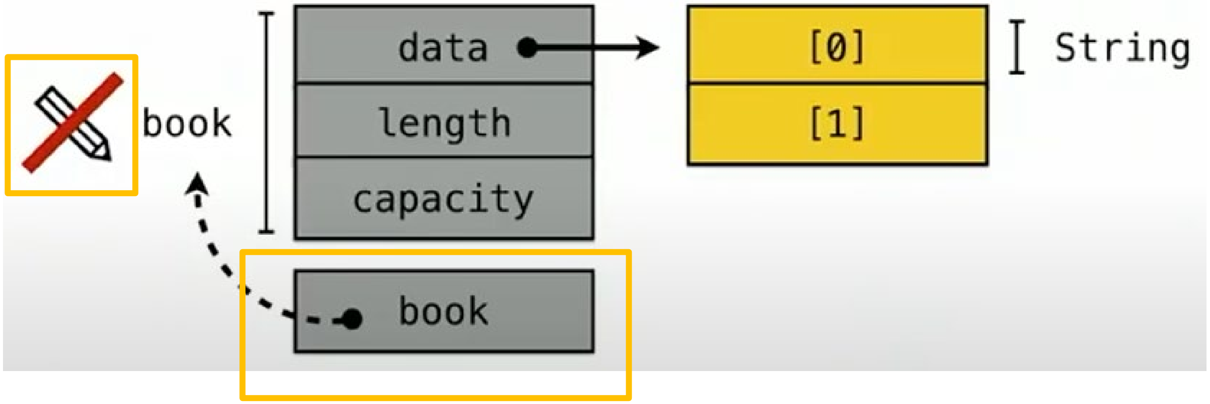
- Rust: giving ownership is the default
  - Not like the copy constructor in C++
  - A bit like a `move` in C++, but enforced at compilation time and no ownership is retained
- Rust: deep copy of data is explicit using `clone()`
  - In C++, the copy constructor does a deep copy

Shared borrow

- Line 12: type is a reference to a vector –> use &
- Line 5, 10: **borrow** the vector, creating a shared reference

```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(&book);
6      // a second call to publish
7      // borrows again the reference
8      // to book.
9      // compilation is successful
10     publish(&book);
11 }
12 fn publish(book: &Vec<String>) {
13     ...
14 }
```

A shared borrow allows aliasing, but no mutation!



Consequences of shared borrow

- Line 4: vector is (shared) borrowed here
  - Freezes the whole container vector
- Line 5: cannot mutate (compilation error)

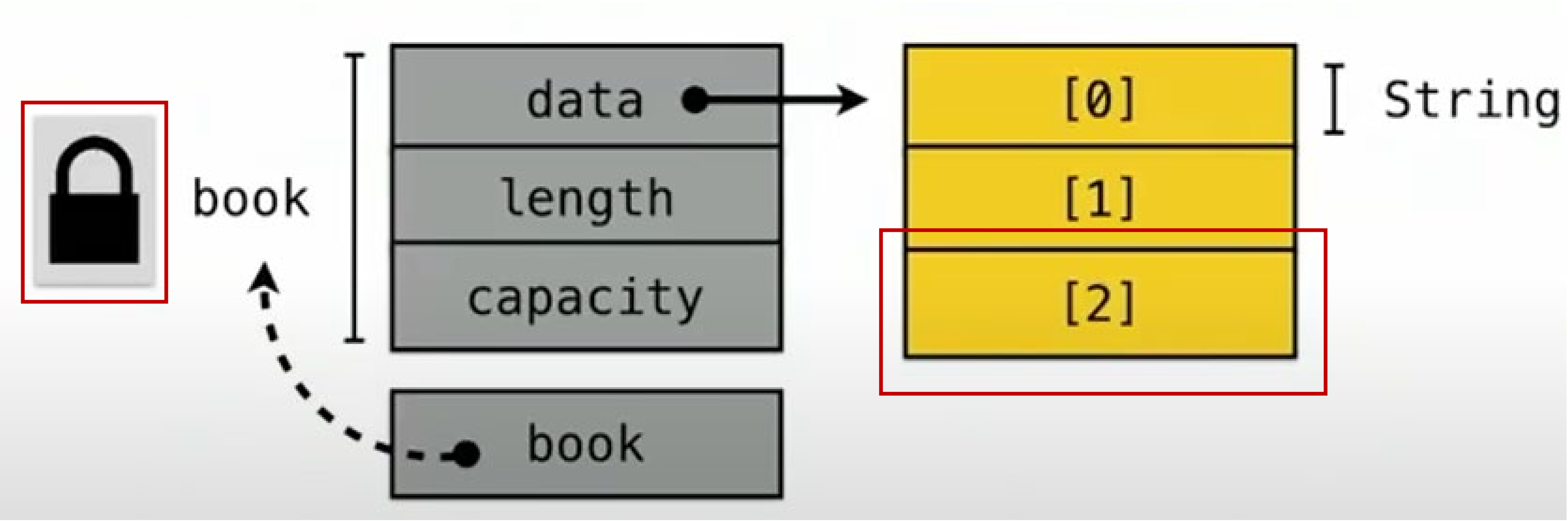
```
1  fn example() {
2      let mut vector = Vec::new();
3      ...
4      let elem = &vector[0];
5
6      // mutation is not allowed while
7      // a shared borrow exists for book.
8      // compilation error
9      vector.push(some_string);
10     ...
11 }
```

A shared borrow allows aliasing, but no mutation!

Mutable borrow

- Line 9: mutable reference to a vector
- Line 5, 6: mutable borrow

```
1  fn main() {
2      let mut book = Vec::new();
3      book.push(...);
4      book.push(...);
5      publish(&mut book);
6      publish(&mut book);
7   }
8
9   fn publish(book: &mut Vec<String>) {
10      book.push(...);
11  }
```



Shared borrow: “Don’t break your friend’s toys”

```
let mut book = Vec::new();
book.push(...); //success: book is mutable
{
    let r = &book; // shared borrow of book
    book.push();   // compilation error: cannot mutate
                   //   while shared
    r.push(...);   // compilation error: cannot mutate
                   //   while shared
}   // shared borrow ends
book.push(...);    // success: book can be mutated
```

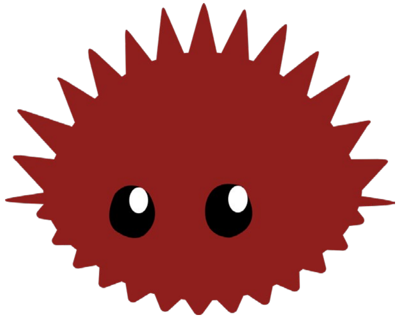
A shared borrow allows aliasing, but no mutation!

Mutable borrow: “No, it’s my turn now!”

```rust
let mut book = Vec::new();
book.push(...);          //success: book is mutable
{
    let r = &mut book; // mutable borrow of book
    book.len();        // compilation error: cannot access
                       //   while mutable borrow exists
    r.push(...);       // success: reference can be mutated
}   // mutable borrow ends
book.push(...);        // success: book can be mutated
```

A mutable borrow allows mutation, but no aliasing!

Memory safety in Rust



- The borrow checker *statically* prevents **aliasing + mutation**
  - Compile time
  - Fighting the borrow checker!
    - Don’t give up! Don’t use unsafe!
- Ownership prevents **double-free**
  - The owner frees
- Borrowing prevents **use-after-free**

- No segfaults!

Type	Ownership	Alias?	Mutate?
T	Owned		Yes
&T	Shared reference	Yes	
&mut T	Mutable reference		Yes

No data races in Rust

- Data race = sharing   +   mutation   +   no ordering
- Sharing + mutation are prevented in Rust

Library-based concurrency

- Not build into the language
  - Rust had message passing build into the language – removed
- Library-based – in std or other libraries
  - Multi-paradigm
  - Leverage on ownership/borrowing

Create a thread

- Line 1: create a thread
  - `loc` is a JoinHandle
  - If `loc` is dropped, the spawned thread is detached
- Line 5: join a thread

```
1 let loc = thread::spawn(|| {
2     "world"
3 });
4 println!("Hello, {}!",
5     loc.join().unwrap());
```

Transfer the vector to a thread

- `move` converts any variables captured by reference or mutable reference to variables captured by *value*
  - `move` keyword: the closure will take ownership of the values it uses from the environment, thus transferring ownership of those values from one thread to another
- Line 5: error: use after move

```
1 let mut dst = Vec:: new();
2 thread::spawn(move || {
3     dst.push(3);
4 });
5 dst.push(3);
```

Remove the **`move`**

- Line 3: error: value doesn't live long enough
  - Possible memory issues: UAF

- Spawn a thread
  - Capture everything as a borrow
  - The close captures `dst` (mutable borrow)
    - Rust *infers* how to capture `dst`

```
1 let mut dst = Vec:: new();
2 thread::spawn(|| {
3     dst.push(3);
4 });
5 dst.push(3);
```

Reference counting (RC)

- Line 4: error: Rc<T> can't be sent across threads
  - RC type is not atomically managed
  - **No Send trait**

```
1 let v = Rc:: new(vec![1 ,2, 3]);
2 let v2 = v.clone();
3 thread::spawn(move || {
4     println!("{}", v.len());
5 });
6 another_fn(v2.clone);
```

Traits

- Like an interface that you can implement for a given type
- It might have methods
- Example
  - Clone
- Marker traits:
  - Send – transferred across thread boundaries
    - String, u32, Arc<String>
  - Sync – safe to share references between threads
    - Type T is Sync if and only if &T is Send
  - Copy – safe to memcpy (for built-in types)
    - u32, f32
    - Not for Strings

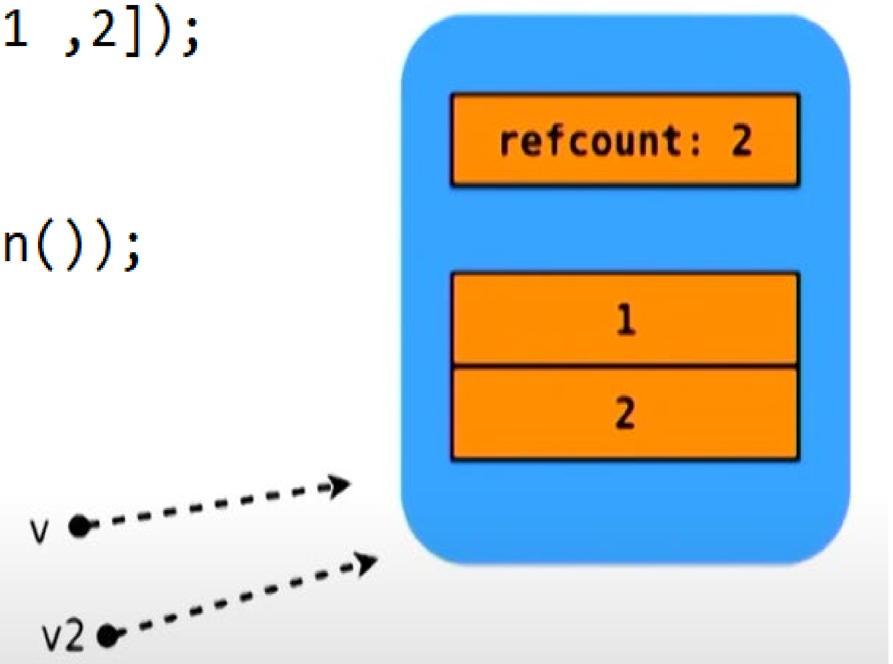
```
trait Clone {
  fn clone(&self) -> Self;
}

impl<T: Clone> Clone for Vec<T> {
  fn clone(&self) -> Vec<T> {
    let mut v = Vec::new();
    for elem in self {
        v.push(elem.clone());
    }
    return v;
  }
}
```

Atomically Reference Counted - **`Arc`**

- Arc: allows only shared references
  - References cannot be mutated
- Line 3: move reference
- Line 4: it's safe to access v

```
1 let v = Arc:: new(vec![1 ,2]);
2 let v2 = v.clone();
3 thread::spawn(move || {
4     println!("{}", v.len());
5 });
6 another_fn(&v2);
```



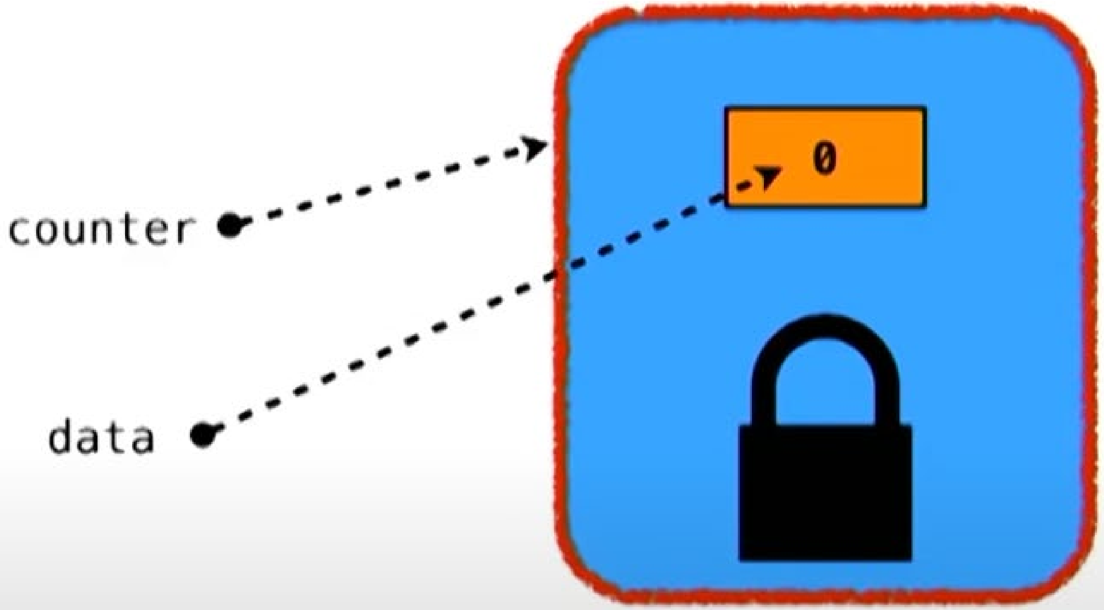
Synchronization in Rust

- Shared memory
  - Mutex
  - Atomics
- Message-passing
  - Channels: MPSC (multi-producer, single-consumer FIFO queue)

Mutex

- Based on ownership
- Data protected by the mutex
- Lock returns a guard - a proxy through which we can access the data

```
fn sync_inc(counter: &Mutex<i32>) {
    let mut data Guard<i32> = counter.lock();
    *data += 1;
}
```



Atomics

- Similar to modern C++
  - Same memory model
- Ordering of memory operations - SeqCst

```rust
let number = AtomicUsize::new(10);
let prev = number.fetch_add(1, SeqCst);
assert_eq!(prev, 10);
let prev = number.swap(2, SeqCst);
assert_eq!(prev, 11);
assert_eq!(number.load(SeqCst), 2);
```

Multi-Producer, Single-Consumer FIFO queue

- Channel with a reading and writing reference
  - Accepts one reader and multiple writers

```rust
let (tx, rx) = mpsc::channel ();
let tx2 = tx.clone();
thread::spawn(move || tx.send(5));
thread::spawn(move || tx2.send(4));

//prints 4 and 5 in an unspecified order
println!("{:?}", rx.recv());
println!("{:?}", rx.recv());
```

Outline

- Challenges in concurrent programming
- Introducing Rust
  - Safety, ownership and borrowing
- Concurrency in Rust
  - Threads, Mutex, Atomic
  - Libraries: Crossbeam, Rayon

Crossbeam

- Ability to create scoped threads – now in std
  - Scope is like a little container we are going to put our threads in
  - You cannot borrow variables mutably into two threads in the same scope
- Message passing using multiple-producer-multiple-consumer channel
  - With exponential backoff

Scoped threads

- Line 5: create the scope
- Line 6: spawn threads

```
1  fn main() {
2      let v = vec![1, 2, 3];
3      println!("main thread has id {}", thread_id::get());
4
5      std::thread::scope(|scope| {
6          scope.spawn(|inner_scope| {
7              println!("Here's a vector: {:?}", v);
8              println!("Now in thread with id {}", thread_id::get());
9          });
10     }).unwrap();
11
12     println!("Vector v is back: {:?}", v);
13 }
```

Producer consumer

- Line 12: use a bounded channel

```
11 fn main() {
12     let (send_end, receive_end) = bounded(CHANNEL_CAPACITY);
13 
14     let mut threads = vec![];
15     for _i in 0 .. NUM_THREADS {
16         let send_end = send_end.clone();
17         threads.push(
18             thread::spawn(move || {
19                 for _k in 0 .. ITEMS_PER_THREAD {
20                     let produced_value = produce_item();
21                     send_end.send(produced_value).unwrap();
22                 }
23             })
24         );
25     }
26 
27     for _j in 0 .. NUM_THREADS {
28         // create consumers
29         let receive_end = receive_end.clone();
30         threads.push(
31             thread::spawn(move || {
32                 for _k in 0 .. ITEMS_PER_THREAD {
33                     let to_consume = receive_end.recv().unwrap();
34                     consume_item(to_consume);
35                 }
36             })
37         );
38     }
39 
40     for t in threads {
41         let _ = t.join();
42     }
43     println!{"Done!"}
44 }
```

Exponential backoff

- Resources might not be available right now?
  - Retry later
- It’s unhelpful to have a tight loop that simply retries as fast as possible
- Wait a little bit and try again
  - If the error occurs, next time wait a little longer
- Rationale: if the resource is overloaded right now, the reaction of requesting it more will make it even more overloaded and makes the problem worse!

Backoff with scoped threads

- Line 12: backoff in lock-free loop
- Line 20: wait for another thread to take its turn first

```
1  use crossbeam_utils::Backoff;
2  use std::sync::atomic::AtomicUsize;
3  use std::sync::atomic::Ordering::SeqCst;
4
5  fn fetch_mul(a: &AtomicUsize, b: usize) -> usize {
6      let backoff = Backoff::new();
7      loop {
8          let val = a.load(SeqCst);
9          if a.compare_and_swap(val, val.wrapping_mul(b), SeqCst) == val {
10             return val;
11         }
12         backoff.spin();
13     }
14 }
```

```
17 fn spin_wait(ready: &AtomicBool) {
18     let backoff = Backoff::new();
19     while !ready.load(SeqCst) {
20         backoff.snooze();
21     }
22 }
```

Rayon

- A data parallelism library
  - Parallelize some spots without full/major rewrite
- Similar in functionality with OpenMP
  - But OpenMP uses compiler directives
- Rationale: reasonably common that computationally-intensive parts of the program happen in a loop, so parallelizing loops is likely to be quite profitable

Example: Sequential maximum of a vector

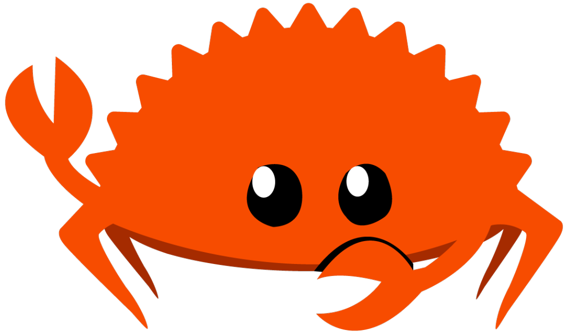
```
fn main() {
    let vec = init_vector();
    let max = MIN;
    vec.iter().for_each(|n| {
        if *n > max {
            max = *n;
        }
    });

    println!("Max value in the array is{}",max);
    if max == MAX {
        println!("This is the max value for an i64.")
    }
}
```

Example: Maximum of a vector with Rayon

```
fn main() {
    let vec = init_vector();
    let max = AtomicI64::new(MIN);
    vec.par_iter().for_each(|n| {
        loop {
            let old = max.load(Ordering::SeqCst);
            if *n <= old {
                break;
            }
            let returned = max.compare_and_swap(old, *n, Ordering::SeqCst);
            if returned == old {
                println!("Swapped {} for {}.", n, old);
                break;
            }
        }
    });
    println!("Max value in the array is {}", max.load(Ordering::SeqCst));
    if max.load(Ordering::SeqCst) == MAX {
        println!("This is the max value for an i64.")
    }
}
```

Rust in a nutshell



- Zero-cost abstraction (like C/C++)
- Memory safety & data-race freedom
- Results in
  - Confident, productive systems programming

- References:
  - https://www.youtube.com/watch?v=SiUBdUE7xnA
  - https://www.youtube.com/watch?v=L0dEE2IqbD8
  - https://www.youtube.com/watch?v=6BYKw0Y758Q