Rust: Fearless Concurrency and Trivial Parallelism

Unique Ownership

- Values are owned by variables
- Only one owner for any value at a time
- Ownership can be moved from place to place
- Values are dropped when the owner goes out of scope

Mutability

- Immutable by default
- Can only change from immutable to mutable when ownership changes
- Declared with mut

```
// Variable declaration
let mut var foo = 1;

// Function definition
fn foo(bar: mut String) { .. }

// Closure
|mut arg1, mut arg2| { .. }

// Reference
&mut foo
```

Borrowing

- Many immutable references
- Only one mutable reference
- No mutable reference at all when any other reference exists
- Statically guaranteed: No dangling pointers!

Lifetimes

- Every reference has a lifetime tied to a scope
- References are only valid within the scope of that lifetime
- Lifetime annotations are generics
- Most of the time, explicit annotation is unnecessary
- Annotation is only required when the lifetime is ambiguous!

Moved String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3
 4     take(string);
 5
 6     println!("{}", string);
 7 }
 8
 9 fn take(string: String) {
10     println!("My string: {}", string);
11
12     // string would be dropped here
13 }
```

Moved String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3         move occurs because `string` has type `std::string::String`, which does not
   implement the `Copy` trait

 4     take(string);
 5          value moved here

 6     println!("{}", string);
 7                    borrow of moved value: `string` (value borrowed here after move)
   }
 8
 9 fn take(string: String) {
10     println!("My string: {}", string);
11
12     // string would be dropped here
13 }
```

Moved String

```rust
fn main() {
    let string = String::from("Hello, Rust!");

    take_ref(&string);

    println!("{}", string);
}

fn take_ref(string: &String) {
    println!("My string: {}", string);

    // string borrow ends here, no drop
}
```

Moved String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3
 4     take_ref(&string);
 5
 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &String) {
10     println!("My string: {}", string);
11
12     // string borrow ends here, no drop
13 }
```

```
-:---  owner.rs       All L12    (Rust Hi FlyC cargo God)
[~/talk/snippets]# rustc owner.rs && ./owner                    -> 0
My string: Hello, Rust!
Hello, Rust!
[~/talk/snippets]#                                              -> 0
```

```
U:%*-  *ansi-term*    All L4     (Term: char run)
```

Mutated String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3
 4     take_ref(&string);
 5
 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &String) {
10     string.push_str("\nI love you!");
11 }
```

Mutated String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3
 4     take_ref(&string);
 5
 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &String) {
10                     consider changing this to be a mutable reference: `&mut std::string::S
   tring`
       string.push_str("\nI love you!");
11     cannot borrow `*string` as mutable, as it is behind a `&` reference (`string` is a `&`
    reference, so the data it refers to cannot be borrowed as mutable)
   }
```

-:--- owner.rs All L10 (Rust FlyC:1/0 cargo God)

Mutated String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3
 4     take_ref(&string);
 5              expected type `&mut std::string::String`
                   found type `&std::string::String`
              mismatched types (types differ in mutability)

 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &mut String) {
10     string.push_str("\nI love you!");
11 }
```

Mutated String

```
 1 fn main() {
 2     let string = String::from("Hello, Rust!");
 3         consider changing this to be mutable: `mut string`

 4     take_ref(&mut string);
 5              cannot borrow `string` as mutable, as it is not declared as mutable (cannot b
   orrow as mutable)

 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &mut String) {
10     string.push_str("\nI love you!");
11 }
```

Mutated String

```
 1 fn main() {
 2     let mut string = String::from("Hello, Rust!");
 3
 4     take_ref(&mut string);
 5
 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &mut String) {
10     string.push_str("\nI love you!");
11 }
```

Mutated String

```
 1 fn main() {
 2     let mut string = String::from("Hello, Rust!");
 3
 4     take_ref(&mut string);
 5
 6     println!("{}", string);
 7 }
 8
 9 fn take_ref(string: &mut String) {
10     string.push_str("\nI love you!");
11 }
```

```
-:---  owner.rs       All L2     (Rust FlyC cargo God)
[~/talk/snippets]# rustc owner.rs && ./owner                    -> 1
Hello, Rust!
I love you!
[~/talk/snippets]#                                              -> 0
```

```
U:%*-  *ansi-term*    All L4     (Term: char run)
```

Even in same function

```
1 fn main() {
2     let string = String::from("Hello, Rust!");
3     let string2 = string;
4
5     println!("{}", string2);
6     println!("{}", string);
7 }
```

Even in same function

```
1 fn main() {
2     let mut string = String::from("Hello, Rust!");
3         move occurs because `string` has type `std::string::String`, which do
  es not implement the `Copy` trait
      let string2 = string;
4                   value moved here

5     println!("{}", string2);
6     println!("{}", string);
7                   borrow of moved value: `string` (value borrowed here after
   move)
  }
```

Rust can figure this out

```rust
fn main() {
    let mut string = String::from("Rust can figure this out!");

    let first_ref = &string;
    println!("{}", first_ref);

    let second_ref = &string;
    println!("{}", second_ref);

    let third_ref = &mut string;
    println!("{}", third_ref);
}
```

```
U:---  multi-ref.rs   All L12    (Rust cargo)
[~/talk/snippets]# rustc multi-ref.rs && ./multi-ref                    -> 0
Rust can figure this out!
Rust can figure this out!
Rust can figure this out!
[~/talk/snippets]#                                                      -> 0
U:%*-  *ansi-term*    All L5     (Term: char run)
```

Rust can figure this out

```rust
fn main() {
    let mut string = String::from("Rust can figure this out!");

    {   // New scope
        let first_ref = &string;
        println!("{}", first_ref);   // Last use of reference

        let second_ref = &string;
        println!("{}", second_ref);  // Last use of reference
    }   // End scope

    let third_ref = &mut string; // Rust knows no other reference is in effect
    println!("{}", third_ref);
}
```

```
U:---  multi-ref.rs   All L1     (Rust cargo God)

[~/rede/snippets]# rustc multi-ref.rs && ./multi-ref                    -> 0
Rust can figure this out!
Rust can figure this out!
Rust can figure this out!
[~/rede/snippets]#                                                      -> 0
```

```
U:%*-  *ansi-term*    Bot L10    (Term: char run)
```

Rust can figure this out

```
 1 fn main() {
 2     let mut string = String::from("Rust can figure this out!");
 3
 4     let _first_ref = &string;
 5                      immutable borrow occurs here
       let _second_ref = &string;
 6     // References not used any more
 7
 8     // Rust knows no other reference is in effect
 9     let third_ref = &mut string;
10                     cannot borrow `string` as mutable because it is also borrowed as immut
   able (mutable borrow occurs here)
       println!("{}", third_ref);
11
12
13     println!("{}", _first_ref);   // This breaks
14                    immutable borrow later used here
   }
```

U:--- **multi-ref.rs** All L13 (Rust FlyC:1/0 cargo God)

Lifetime annotations

```rust
fn main() {
    let string1 = String::from("Hello");
    let string2 = String::from("Rust!");

    let str_vec = make_vec(&string1, &string2);
    println!("{:?}", str_vec);
}

fn make_vec(string1: &str, string2: &str) -> Vec<&str> {
    vec![string1, string2]
}
```

Lifetime annotations

```
 1 fn main() {
 2     let string1 = String::from("Hello");
 3     let string2 = String::from("Rust!");
 4
 5     let str_vec = make_vec(&string1, &string2);
 6     println!("{:?}", str_vec);
 7 }
 8
 9 fn make_vec(string1: &str, string2: &str) -> Vec<&str> {
10                                                 this function's return type
   contains a borrowed value, but the signature does not say whether it is borro
   wed from `string1` or `string2`
                                                   missing lifetime specifier (
   expected lifetime parameter)
     vec![string1, string2]
11 }
```

Lifetime annotations

```
1  fn main() {
2      let string1 = String::from("Hello");
3      let string2 = String::from("Rust!");
4
5      let str_vec = make_vec(&string1, &string2);
6      println!("{:?}", str_vec);
7  }
8
9  fn make_vec<'a>(string1: &'a str, string2: &'a str) -> Vec<&'a str> {
10     vec![string1, string2]
11 }
```

Lifetime annotations

```
 1 fn main() {
 2     let string1 = String::from("Hello");
 3     let string2 = String::from("Rust!");
 4
 5     let str_vec = make_vec(&string1, &string2);
 6     println!("{:?}", str_vec);
 7 }
 8
 9 fn make_vec<'a>(string1: &'a str, string2: &'a str) -> Vec<&'a str> {
10     vec![string1, string2]
11 }
```

```
-:---  lifetime.rs    All L12    (Rust cargo)
[~/talk/snippets]# rustc lifetime.rs                                    -> 0
[~/talk/snippets]# ./lifetime                                           -> 0
["Hello", "Rust!"]
[~/talk/snippets]#                                                      -> 0
```

Structs with References

```
1  struct StrStruct {
2      internal_ref: &str
3  }
4
5  fn main() {
6      let str_struct;
7      {
8          let string = String::from("The answer is 42!");
9
10         str_struct = make_str_struct(&string);
11     } // string dropped here!
12
13     // the reference is now invalid
14     println!("{}", str_struct.internal_ref);
15 }
16
17 // We don't need lifetime annotations here!
18 fn make_str_struct(string: &str) -> StrStruct {
19     StrStruct {
20         internal_ref: string
21     }
22 }
```

Structs with References

```
1  struct StrStruct {
2      internal_ref: &str
3                    missing lifetime specifier (expected lifetime parameter)
   }
4
5  fn main() {
6      let str_struct;
7      {
8          let string = String::from("The answer is 42!");
9
10         str_struct = make_str_struct(&string);
11     } // string dropped here!
12
13     // the reference is now invalid
14     println!("{}", str_struct.internal_ref);
15 }
16
17 // We don't need lifetime annotations here!
18 fn make_str_struct(string: &str) -> StrStruct {
19     StrStruct {
20         internal_ref: string
21     }
22 }
```

Structs with References

```
1  struct StrStruct<'a> {
2      internal_ref: &'a str
3  }
4
5  fn main() {
6      let str_struct;
7      {
8          let string = String::from("The answer is 42!");
9
10         str_struct = make_str_struct(&string);
11     } // string dropped here!
12
13     // the reference is now invalid
14     println!("{}", str_struct.internal_ref);
15 }
16
17 // We don't need lifetime annotations here!
18 fn make_str_struct(string: &str) -> StrStruct {
19     StrStruct {
20         internal_ref: string
21     }
22 }
```

Structs with References

```
 1 struct StrStruct<'a> {
 2     internal_ref: &'a str
 3 }
 4
 5 fn main() {
 6     let str_struct;
 7     {
 8         let string = String::from("The answer is 42!");
 9
10         str_struct = make_str_struct(&string);
11                                      `string` does not live long enough (borr
   owed value does not live long enough)
      } // string dropped here!
12    `string` dropped here while still borrowed

13     // the reference is now invalid
14     println!("{}", str_struct.internal_ref);
15                     borrow later used here
   }
16
17 // We don't need lifetime annotations here!
18 fn make_str_struct(string: &str) -> StrStruct {
19     StrStruct {
20         internal_ref: string
21     }
22 }
```

Concurrency vs Parallelism

- Concurrency
  - Doing things at different times or out of order
    - Async IO
    - Interrups
    - User Interface
- Parallelism
  - Doing multiple things at the same time
    - Multi-Threading
    - Multi-Processing
- Rust doesn't make much of a distiction here
- Because it deals with it the same way

Rc

- Reference Counted Pointer
- Takes ownership of the value
- Handles descruction when ref-count becomes zero
- Only hands out read-only references usually
- !Send + !Sync: Not thread safe!

```
struct Rc<T> {
    strong: usize,
    weak: usize
    value: T
}
```

RefCell

- Run-time Borrow Checker
- Gives out many readable XOR one writable references to interior value
- References wrapped in a guard and can't escape
- Send + !Sync: Sendable, but no shared references

```
pub struct RefCell<T: ?Sized>
    borrow: Cell<BorrowFlag>,
    value: UnsafeCell<T>,
}
```

Callbacks

```
 1 struct Callbacks {
 2     on_click: Box<dyn FnMut()>,
 3     on_dbl_click: Box<dyn FnMut()>
 4 }
 5
 6 fn main()  {
 7     let mut click_string = String::new();
 8
 9     let callbacks = Callbacks {
10         on_click:
11         Box::new(|| {                                  // This is a closusure
12             let click_string = &mut click_string;i     //
13             click_string.push_str("Clicked! ")         //
14         } ),                                           // End of closure
15
16         on_dbl_click:
17         Box::new(|| {                                    // This is a closure
18             let click_string = &mut click_string;        //
19             click_string.push_str("Doubleclicked! ");    //
20         } ),                                             // End of closure
21     };
22
23 }
```

U:--- **callback.rs**   All L24   (Rust cargo God)

Callbacks

```
 6 fn main()  {
 7     let mut click_string = String::new();
 8
 9     let callbacks = Callbacks {
10         on_click:
11         Box::new(|| {                                  // This is a closusure
12                 value captured here
          cast requires that `click_string` is borrowed for `'static`
              let click_string = &mut click_string;    //
13                                                       `click_string` does not live long enough (borrowed val
   ue does not live long enough)
              click_string.push_str("Clicked! ")         //
14         } ),                                          // End of closure
15
16         on_dbl_click:
17         Box::new(|| {                                    // This is a closure
18                 value captured here
          cast requires that `click_string` is borrowed for `'static`
              let click_string = &mut click_string;          //
19                                                       `click_string` does not live long enough (borrowed val
   ue does not live long enough)
              click_string.push_str("Doubleclicked! ");    //
20         } ),                                          // End of closure
21     };
22
23 }
   `click_string` dropped here while still borrowed
```

U:--- **callback.rs** Bot L23 (Rust FlyC:5/1 cargo God)

Callbacks

```
 6 fn main()  {
 7     let mut click_string = String::new();
 8
 9     let callbacks = Callbacks {
10         on_click:
11         Box::new(|| {                                  // This is a closusure
12             let click_string = &mut click_string;      //
13             click_string.push_str("Clicked! ")         //
14         } ),                                           // End of closure
15
16         on_dbl_click:
17         Box::new(|| {                                      // This is a closure
18                   value captured here
           cast requires that `click_string` is borrowed for `'static`
               let click_string = &mut click_string;          //
19                                    `click_string` does not live long enough (borrowed val
   ue does not live long enough)
               click_string.push_str("Doubleclicked! ");      //
20         } ),                                               // End of closure
21     };
22
23 }
   `click_string` dropped here while still borrowed
```

Callbacks

```
 1 struct Callbacks {
 2     on_click: Box<dyn FnMut()>,
 3     on_dbl_click: Box<dyn FnMut()>
 4 }
 5
 6 fn main() {
 7     let mut click_string = String::new();
       move occurs because `click_string` has type `std::string::String`, which does not
   implement the `Copy` trait

 9     Callbacks {
10         on_click:
11         Box::new(move || {                          // This is a closusure
12                 value moved into closure here
               let click_string = &mut click_string;   //
13                                                     variable moved due to use in closure
               click_string.push_str("Clicked! ")      //
14         } ),                                        // End of closure
15
16         on_dbl_click:
17         Box::new(move || {                              // This is a closure
18                 use of moved value: `click_string` (value used here after move)
               let click_string = &mut click_string;   //
19                                                     use occurs due to use in closure
               click_string.push_str("Doubleclicked! ");   //
20         } ),                                            // End of closure
21     };
```

U:--- **callback.rs** Top L18 (Rust FlyC:1/0 cargo)

Callbacks

```
 3 struct Callbacks {
 4     on_click: Box<dyn FnMut()>,
 5     on_dbl_click: Box<dyn FnMut()>
 6 }
 7
 8 fn main()  {
 9     let mut click_string = Rc::new(String::new());
10     // Clone Rc-pointer, increases ref-count
11     let mut dbl_click_string = click_string.clone();
12
13     Callbacks {
14         on_click:
15         Box::new(move || {                          // This is a closusure
16             let click_string = &mut click_string;   //
17             click_string.push_str("Clicked! ")      //
18             cannot borrow data in a `&` reference as mutable (cannot borrow as mutable)
           } ),                                        // End of closure
19
20         on_dbl_click:
21         Box::new(move || {                            // This is a closure
22             let click_string = &mut dbl_click_string;       //
23             click_string.push_str("Doubleclicked! ");   //
24         } ),                                            // End of closure
25     };
26
27 }
```

Callbacks

```
 5     on_click: Box<dyn FnMut()>,
 6     on_dbl_click: Box<dyn FnMut()>
 7 }
 8
 9 fn main()  {
10     let click_string: Rc<RefCell<String>> = Rc::new(RefCell::new(String::new()));
11     // Clone Rc-pointer, increases ref-count
12     let dbl_click_string = click_string.clone();
13
14     Callbacks {
15         on_click:
16         Box::new(move || {
17             // borrow_mut() returns a wrapped mutable reference
18             let mut string_ref: RefMut<_> = click_string.borrow_mut();
19             string_ref.push_str("Clicked! ")
20             // reference wrapper dropped here, RefCell usable again
21         } ),
22
23         on_dbl_click:
24         Box::new(move || {
25             // borrow_mut() returns a wrapped mutable reference
26             let mut string_ref: RefMut<_> = dbl_click_string.borrow_mut();
27             string_ref.push_str("Doubleclicked! ");
28             // reference wrapper dropped here, RefCell usable again
29         } ),
30     };
31
32 }
```

Send/Sync Traits

- Marker Traits
- Send: Allows moving values to other threads
- Sync: Allows sharing references with other threads
- Most types are actually Send + Sync!

Sharing & Locking

Arc and Mutex

- Arc: Atomically counted Rc
  - Like Rc, can be cloned without copying data
  - Like Rc, only allows read access
  - Unlike Rc, can be moved and shared among threads
  - Send + Sync
- Mutex: Atomically locked RefCell
  - Like RefCell, takes ownership of the contained value
  - Like RefCell, ensures exclusive access to data
  - Unlike RefCell, doesn't panic when locking a locked value
  - Unlike RefCell, can be moved and shared among threads
  - Send + Sync

Channels

- Sending values to other parts of your code
- Sender never blocks, unless you use synchronous channels
- Receiver always blocks as long as Senders exist
- Great for simple event systems when combined with enums
- The stdlib mpsc channels allow one Receiver and multiple Senders

Channels

```rust
use std::sync::mpsc::channel;


fn main() {
    let (tx, rx) = channel();

    std::thread::spawn(move || {
        for i in 0..100 {
            tx.send(i).ok();
        }
    });

    std::thread::spawn(move || {
        for number in rx.iter() {
            println!("{}", number);
        }
    });
}
```

Channels

```
 7     std::thread::spawn(move || {
 8         for i in 0..100 {
 9             tx.send(i).ok();
10         }
11     });
12
13     std::thread::spawn(move || {
14         for number in rx.iter() {
15             println!("{}", number);
16         }
17     });
18 }
```

```
U:---  channels.rs     Bot L19     (Rust FlyC cargo God)
[~/rede/snippets]# ./channels                                    -> 0
[~/rede/snippets]#                                               -> 0
```

```
U:%*-  *ansi-term*     All L2      (Term: char run)
```

Channels

```
 1 use std::sync::mpsc::channel;
 2
 3
 4 fn main() {
 5     let (tx, rx) = channel();
 6
 7     std::thread::spawn(move || {
 8         for i in 0..100 {
 9             tx.send(i).ok();
10         }
11     });
12
13     let handle = std::thread::spawn(move || {
14         for number in rx.iter() {
15             println!("{}", number);
16         }
17     });
18
19     handle.join().ok();
20 }
```

Channels

```
10          }
11      });
12
13      let handle = std::thread::spawn(move || {
14          for number in rx.iter() {
15              println!("{}", number);
16          }
17      });
18
19      handle.join().ok();
20  }
```

```
U:---  channels.rs     Bot L21     (Rust FlyC cargo God)
89
90
91
92
93
94
95
96
97
98
99
[~/rede/snippets]#                                          -> 0
U:%*-  *ansi-term*     Bot L13     (Term: char run)
```

Channels

```
 1 use std::sync::mpsc::channel;
 2
 3
 4 fn main() {
 5     let (tx, rx) = channel();
 6     let tx2 = tx.clone();
 7
 8     std::thread::spawn(move || {
 9         for i in 0..100 {
10             tx.send(i).ok();
11         }
12     });
13
14     std::thread::spawn(move || {
15         for i in (0..100).rev() {
16             tx2.send(i).ok();
17         }
18     });
19
20     let handle = std::thread::spawn(move || {
21         for number in rx.iter() {
22             println!("{}", number);
23         }
24     });
25
26     handle.join().ok();
27 }
```

U:--- **channels.rs**   All L27   (Rust FlyC cargo God)

Channels

```
 5      let (tx, rx) = channel();
 6      let tx2 = tx.clone();
 7
 8      std::thread::spawn(move || {
 9          for i in 0..100 {
10              tx.send(i).ok();
11          }
12      });
13
14      std::thread::spawn(move || {
15          for i in (0..100).rev() {
16              tx2.send(i).ok();
17          }
18      });
```

```
U:---   channels.rs     10% L16     (Rust FlyC cargo God)
4
3
94
2
95
1
96
0
97
98
99
[~/rede/snippets]#                                          -> 0

U:%*-   *ansi-term*     Bot L58     (Term: char run)
```

rayon/crossbeam

- par_iter
  - Can parallelize most iterations by just replacing .iter() with .par_iter()
  - Be careful about small loops, there's overhead due to locking/synchronization!
- Thread Pools
  - Global and local pools
- Scoped Threads
  - Share values on the stack with threads
  - Readable references in multiple threads
  - Writeable reference in one thead

Scoped Threads

```
fn main() {
    let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack

    let handle = std::thread::spawn(|| {
        let found_primes = (0..100)
            .into_iter()
            .filter(|num| {
                primes.contains(num)      // Access stack data
            }).collect::<Vec<usize>>();

        println!("{:?}", found_primes);
    });

    let handle2 = std::thread::spawn(|| {
        let found_primes = (0..100).rev()
            .into_iter()
            .filter(|num| {
                primes.contains(num)      // Access stack data
            }).collect::<Vec<usize>>();

        println!("{:?}", found_primes);
    });

    handle.join();    // Wait until thread is done, so primes isn't dropped
    handle2.join();   // Wait until thread is done, so primes isn't dropped
} // <- Dropping primes here
```

Scoped Threads

```
 1 fn main() {
 2     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
 3
 4     let handle = std::thread::spawn(|| {
 5                                     to force the closure to take ownership of
   `primes` (and any other referenced variables), use the `move` keyword: `move
   ||`
                                       closure may outlive the current function,
   but it borrows `primes`, which is owned by the current function (may outlive
   borrowed value `primes`)
                     function requires argument type to outlive `'static`
         let found_primes = (0..100)
 6           .into_iter()
 7           .filter(|num| {
 8               primes.contains(num)     // Access stack data
 9               `primes` is borrowed here
             }).collect::<Vec<usize>>();
10
11         println!("{:?}", found_primes);
12     });
13
14     let handle2 = std::thread::spawn(|| {
15         let found_primes = (0..100).rev()
16           .into_iter()
17           .filter(|num| {
18               primes.contains(num)     // Access stack data
```

Scoped Threads

```
 1 fn main() {
 2     let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
 3     let primes_ref = &primes;
 4                      `primes` does not live long enough (borrowed value does
   not live long enough)
       let primes_ref2 = &primes;
 5
 6     let handle = std::thread::spawn(move || {
 7                  argument requires that `primes` is borrowed for `'static`
           let found_primes = (0..100)
 8             .into_iter()
 9             .filter(|num| {
10                 primes_ref.contains(num)      // Access stack data
11             }).collect::<Vec<usize>>();
12
13         println!("{:?}", found_primes);
14     });
15
16     let handle2 = std::thread::spawn(move || {
17         let found_primes = (0..100).rev()
18             .into_iter()
19             .filter(|num| {
20                 primes_ref2.contains(num)      // Access stack data
21             }).collect::<Vec<usize>>();
22
23         println!("{:?}", found_primes);
```

Scoped Threads

```
1  use rayon;
2
3  fn main() {
4      let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
5
6      rayon::scope(|s| {
7          s.spawn(|_| {
8              let found_primes = (0..100)
9                  .into_iter()
10                 .filter(|num| {
11                     primes.contains(num)      // Access stack data
12                 }).collect::<Vec<usize>>();
13
14             println!("{:?}", found_primes);
15         });
16
17         s.spawn(|_| {
18             let found_primes = (0..100).rev()
19                 .into_iter()
20                 .filter(|num| {
21                     primes.contains(num)      // Access stack data
22                 }).collect::<Vec<usize>>();
23
24             println!("{:?}", found_primes);
25         });
26     });
```

Scoped Threads

```
19                  .into_iter()
20                  .filter(|num| {
21                      primes.contains(num)      // Access stack data
22                  }).collect::<Vec<usize>>();
23
24              println!("{:?}", found_primes);
25          });
26      });
27      // Rayon automatically joins all threads
28  } // <- Dropping primes here
```

```
-:---  scoped-threads.rs   Bot L27    (Rust FlyC cargo God)
[~/talk/snippets]# cargo run --bin scoped-threads                    -> 0
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/scoped-threads`
[15, 13, 11, 9, 7, 5, 3, 2]
[2, 3, 5, 7, 9, 11, 13, 15]
[~/talk/snippets]#                                                   -> 0
```

Scoped Threads

```
1  use rayon;
2  use rayon::prelude::*;
3
4  fn main() {
5      let primes = [2,3,5,7,9,11,13,15]; // Some data on the stack
6
7      rayon::scope(|s| {
8          s.spawn(|_| {
9              let found_primes = (0..100_usize)
10                 .into_par_iter()
11                 .filter(|num| {
12                     primes.contains(num)     // Access stack data
13                 }).collect::<Vec<usize>>();
14
15             println!("{:?}", found_primes);
16         });
17
18         s.spawn(|_| {
19             let found_primes = (0..100_usize)
20                 .into_par_iter()
21                 .rev()
22                 .filter(|num| {
23                     primes.contains(num)     // Access stack data
24                 }).collect::<Vec<usize>>();
25
26             println!("{:?}", found_primes);
```

Scoped Threads

```
13                  }).collect::<Vec<usize>>();
14
15              println!("{:?}", found_primes);
16          });
17
18          s.spawn(|_| {
19              let found_primes = (0..100_usize)
20                  .into_par_iter()
21                  .rev()
22                  .filter(|num| {
23                      primes.contains(num)     // Access stack data
24                  }).collect::<Vec<usize>>();
25
26              println!("{:?}", found_primes);
-:---  scoped-threads.rs   40% L18    (Rust FlyC cargo God)
[~/talk/snippets]# cargo run --bin scoped-threads                    -> 0
    Finished dev [unoptimized + debuginfo] target(s) in 0.03s
     Running `target/debug/scoped-threads`
[2, 3, 5, 7, 9, 11, 13, 15]
[15, 13, 11, 9, 7, 5, 3, 2]
[~/talk/snippets]#                                                   -> 0
```

Hunter's Async type

- Why not use Futures?
  - Futures forget: Values can only be returned once
  - No asynchronous API to read directory contents and metadata
  - It seemed interesting and fun
- Advantages:
  - Easy to wrap any synchronous operation
  - Flexible ad-hoc mutation of value from anywhere, anytime
- Disadvantage: Not zero-cost

Async Type Definitions

```rust
// Final value is temporarily stored here
pub type AsyncValue<T> = Arc<Mutex<Result<T>>>;

// The closure that computes the value we want
pub type AsyncValueFn<T> = FnOnce() -> Result<T> + Send + 'static;

// This closure runs when the value is ready
pub type OnReadyFn<T> = FnOnce(Result<&mut T, AError>)
                              -> Result<()> + Send + 'static;

// This stores all the state (final values, closures, state, etc)
// When it's done running, the unnecessary Arcs cause a few bytes overhead :(
pub struct Async<T: Send + 'static>
{
    // Value is moved here later, so further access doesn't require locking
    pub value: AResult<T>,

    async_value: AsyncValue<T>,
    async_closure: Arc<Mutex<Option<Box<dyn AsyncValueFn<T>>>>>,

    // A Vec so that multiple closures can be registered
    on_ready: Arc<Mutex<Vec<Box<dyn AsyncReadyFn<T>>>>>,
}
```

Async Usage

```rust
fn read_dir_async() -> Result<Async<Files>, Error> {
    let mut async_files = Async::new(|| {
        // Can be slow!!
        let files: Files = Files::read_dir("/foo")?;

        Ok(files)
    });

    // Send event to main loop when ready, so it redraws screen
    async_files.on_ready(|files| send_event_files_ready(&files.dir))?;

    // Run in default thread pool
    async_files.run_pooled(None)?;

    // Return async dir entries
    Ok(files)
}

fn draw_files(async_files: Async<Files>) -> Result<(), Error> {
    // Optionally do this once to move async value out of Mutex
    files.pull_async().ok();

    let files: &Files = files.get()?;

    for file in files.iter() {
        println!("{}", file.name);
```

Async File Selection

```
// Run some program that prints out the path of a file to stdout
let path = run_external_selector()?;

// Create File structure from that path
let file = File::new_from_path(&path, None)?;

// Get the parent directory of that file
let dir = file.parent_as_file()?;

// Does some bookkeeping and asynchronously loads new directory
self.main_widget_goto(&dir)?;

// We don't want to wait for the directory to load here!


// This adds a new callback to be run when the directory is loaded
// Doesn't overwrite any previously registered callbacks
self.main_async_widget_mut()?
    .widget
    .on_ready(move |w, _| {
        w?.select_file(&file);
        Ok(())
    })?;


// Execution goes on immediately
```

Other Optimizations

- 3-Phase directory loading
  1. Load file paths/names with read_dir()
  2. Load metadata for visible files
  3. Calculate number of files in visible directories
- Mostly Useless
  - Async bookmark saving
  - Async tag loading/saving
  - Async config loading