# Antioxidants for Rust

## WD-40 is Child's Play

uid=1000(Jay Angra) gid=1000(bynx)

Security Innovation

October 2020

# Outline

# Introduction

# What will be covered in this talk?

- Streamlined introduction to Rust
- Common Vulnerability Patterns
- Examples? Examples!

# What will be covered in this talk?

- Streamlined introduction to Rust
- Common Vulnerability Patterns
- Examples? Examples!

# What will be covered in this talk?

- Streamlined introduction to Rust
- Common Vulnerability Patterns
- Examples? Examples!

# Refresher on Memory Corruption and Safety

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks
  - Double Free

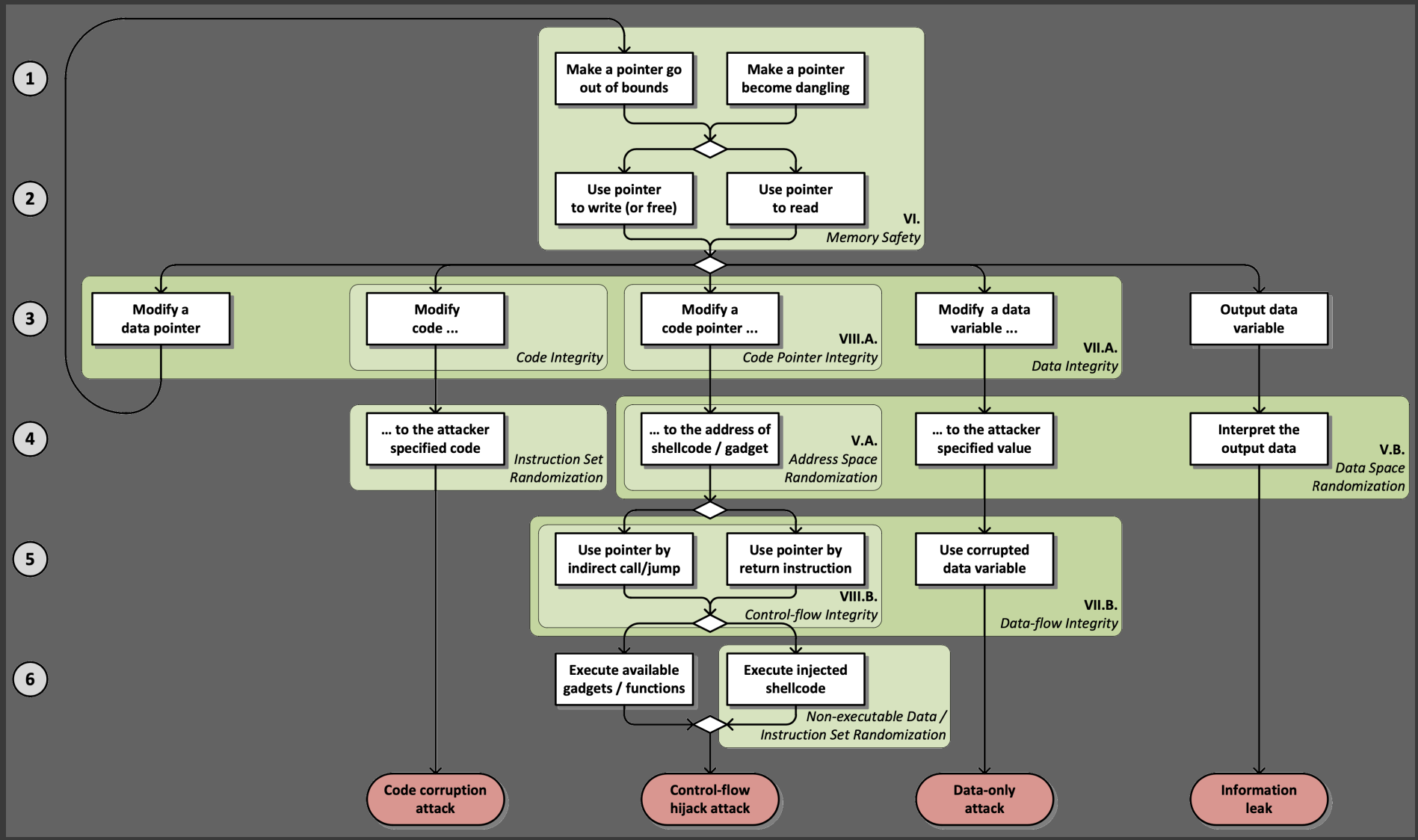
# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks
  - Double Free

# Type of Memory Erros

- Access errors
  - Buffer Overflows
  - Race conditions
  - UAF
  - Segmentation Faults
- Uninitialized variables
- Memory leaks
  - Double Free

# Memory Corruption Attack Model



# What Causes Memory Corruption?

*Aliasing* + ~~*Mutation*~~

~~*Aliasing*~~ + *Mutation*

*Aliasing* + *Mutation*

# The Hotel Analogy

Imagine a heap space and heap allocator as rooms in a hotel and it's front desk, respectively

$\rightarrow$ *pointers are key cards for the rooms*

- We checkout from a room, but *keep the working room key*
  $\rightarrow$ **The key is our dangling pointer**
- The front desk will eventually check-in another person into the room we just left (i.e., allocating new objects)
  $\rightarrow$ **Our dangling key can still access their room**

# The Rust Programming Language

# Hello, Rust!

- Focuses on *safety, speed, and concurrency*
- Combines *low-level control* over performance with *high-level convenience* and safety guarantees
- `rustc`, the rust compiler, won't do anything with a value unless explicitly instructed to do so
  - unclear / uncertain logic results in a `painc!`

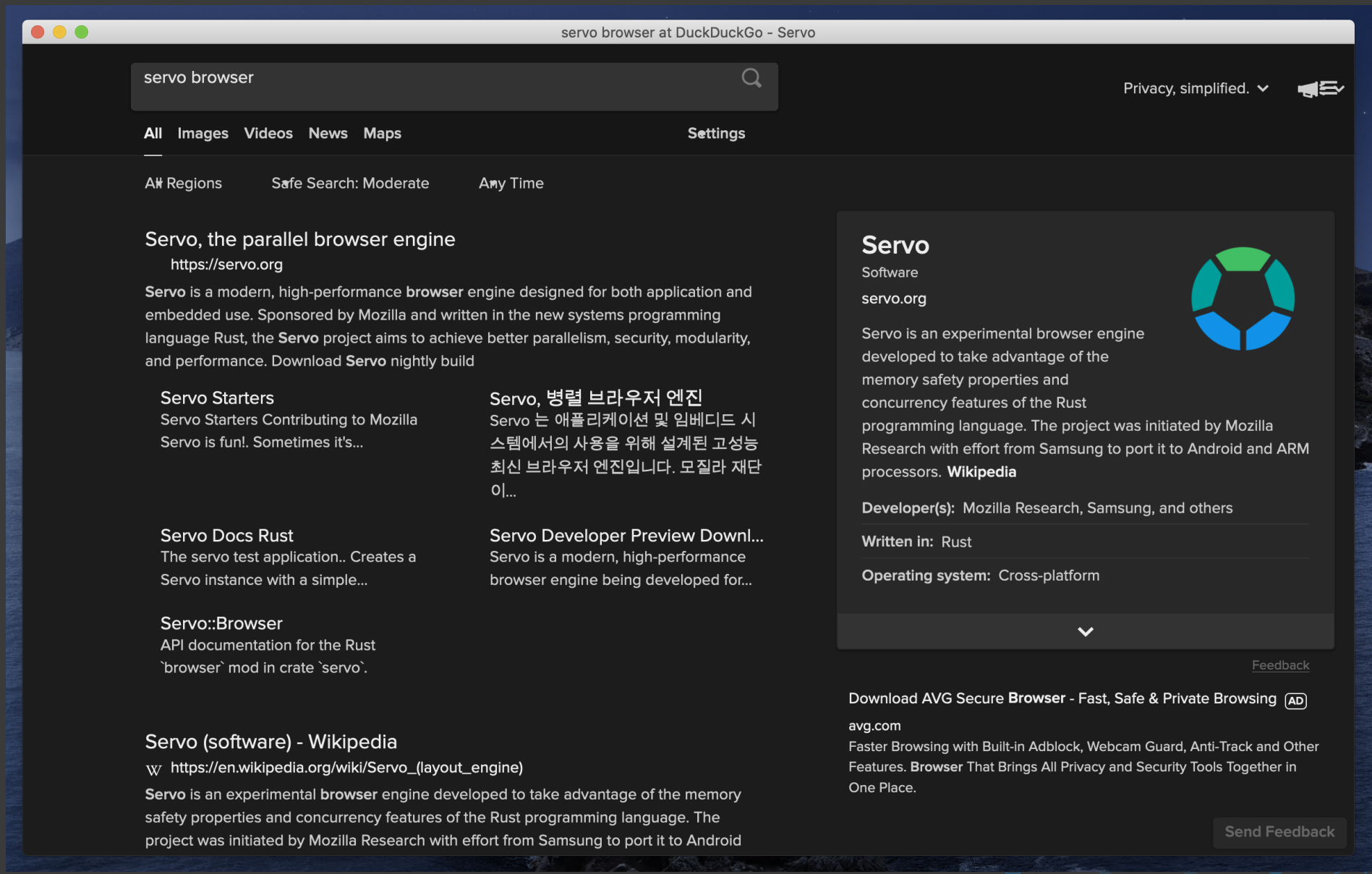
# Hello, Rust!

- `cargo` is rust's swiss army package manager:
  - manages dependencies, compilation, and publication
- `rust-analyzer` and `rls`
  - Fundamentally diferent language server implementations
  - `rls` runs a compiler on the whole project and dumps a huge JSON file with facts derived during the compilation process
  - `rust-analyzer` maintains a persistent compiler pid, providing on-demand analysis
- `rustc` and `rustup` are the rust compiler and toolchain manager, respectively

# Some Cool Projects

- **Servo** – A modern, high-performance browser engine designed for both application and embedded use
- **Redox OS** – A modern microkernel operating system (with optional GUI)
- **Alacritty** – a GPU accelerated terminal emulator ("the fastest in existence")

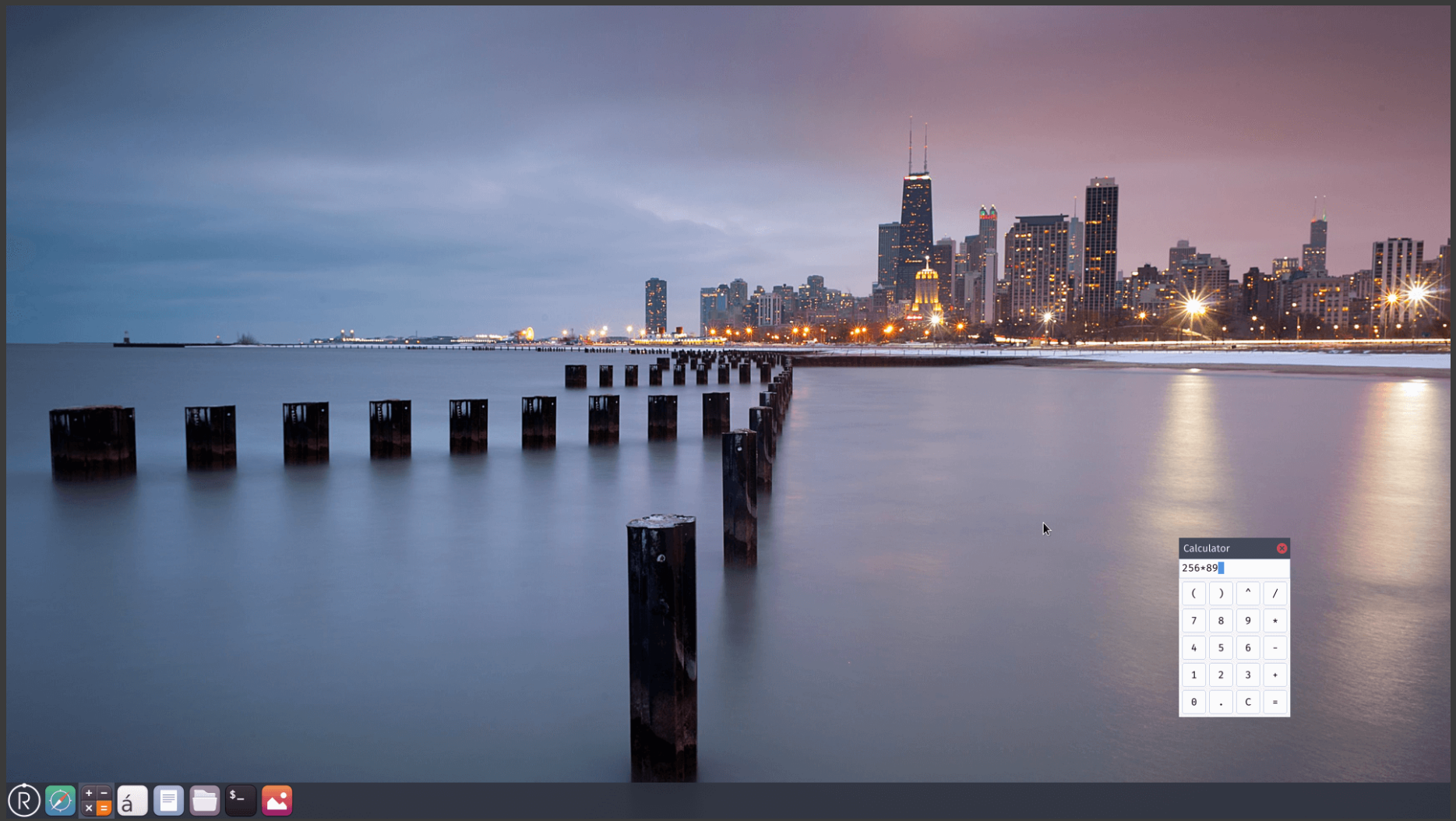
# Servo Browser Engine



# Some Cool Projects

- **Servo** – A modern, high-performance browser engine designed for both application and embedded use
- **Redox OS** – A modern microkernel operating system (with optional GUI)
- **Alacritty** – a GPU accelerated terminal emulator ("the fastest in existence")

# Redox OS with Oribital GUI



# Some Cool Projects

- **Servo** – A modern, high-performance browser engine designed for both application and embedded use
- **Redox OS** – A modern microkernel operating system (with optional GUI)
- **Alacritty** – a GPU accelerated terminal emulator ("the fastest in existence")

# Alacritty Terminal Emulator

Alacritty

```rust
/// Run Alacritty
///
/// Creates a window, the terminal state, pty, I/O event loop, input processor,
/// config change monitor, and runs the main display loop.
fn run(mut config: Config, options: cli::Options) -> Result<(), Box<Error>> {
    // Create a display.
    //
    // The display manages a window and can draw the terminal
    let mut display = Display::new(&config, &options)?;

    println!(
        "PTY Dimensions: {:?} x {:?}",
        display.size().lines(),
        display.size().cols()
    );

    // Create the terminal
    //
    // This object contains all of the state about what's being displayed. It's
    // wrapped in a clonable mutex since both the I/O loop and display need to
    // access it.
    let size = display.size().to_owned();
    let terminal = Arc::new(FairMutex::new(Term::new(size)));

    // Create the pty
    //
    // The pty forks a process to run the shell on the slave side of the
    // pseudoterminal. A file descriptor for the master side is retained for
    // reading/writing to the shell.
    let mut pty = tty::new(display.size());

    // Create the pseudoterminal I/O loop
    //
    // pty I/O is ran on another thread as to not occupy cycles used by the
    // renderer and input processing. Note that access to the terminal state is
    // synchronized since the I/O loop updates the state, and the display
    // consumes it periodically.
    let event_loop = EventLoop::new(
        terminal.clone(),
        display.notifier(),
        pty.reader(),
```

85,5 47%

1:vim* 21:49 :: Sunday, January 01, 2017

# Error Handling

Run- and compile-time errors are classified in one of two groups:

- recoverable: `Result<T,E>`
- unrecoverable: `panic!`

# Error Handling: `panic!`

- `panic!` is a rust macro
- prints an optional failure message
- unwinds and cleans up stack, then quits
- occurs when a bug of some kind has been detected and it's not clear to the programmer how to handle the error
  - *note:* `cargo` *calls* `panic!` *at compile-time unless explicit instructions are provided for all values*

# Memory Safety Gaurentees

## How Does Rust Guarantee Memory Safety?

- Ownership
- Borrowing
- Lifetimes
- Unsafe

# Ownership

- **Variable bindings** have **ownership** of the value bound to them
  - *This is by design*
  - Every value has a *single*, *statically-known*, *owning path* in the source
- A data value can only have **one owner at a time**
- When a binding goes out of scope, rust frees the bound resource

  → note: rust is **block-scoped**
- Data values are categorized into two group based on the presence of the `Copy` trait
  - **Copy Types**
  - **Move Types**

# Ownership

```rust
// Copy Type example
fn main() {
    let a: [i32; 3] = [1, 2, 3];
    let b = a; // Static Arrays implement the ‘Copy‘ trait

    println!("{:?} {:?}", a, b);
    // [1, 2, 3] [1, 2, 3]
}
```

```rust
// Move Type example
fn main() {
    let a: Vec<i32> = vec![1, 2, 3]; //std::box::Box::new(...)
    let b = a; // Ref types don’t implement the ‘Copy‘ trait

    println!("{:?} {:?}", a, b);
    // panic!       ...          ^ value borrowed here after move
}
```

# Memory Safety Gaurentees

## How Does Rust Guarantee Memory Safety?

- Ownership
- Borrowing
- Lifetimes
- Unsafe

# Ownership and Borrowing

Alice

Bob

```
1  fn main() {
2      let alice = vec![1, 2, 3]; // Alice owns the bound Vec
3      {
4          let bob = alice;
5          println!("bob: {}", bob[0]);
6      }
7      println!("alice: {}", alice[0]);
8  }
```

# Ownership and Borrowing

Alice

Bob

```
1  fn main() {
2      let alice = vec![1, 2, 3];
3      {
4          let bob = alice;
5          println!("bob: {}", bob[0]);
6      }
7      println!("alice: {}", alice[0]);
8  }
```

# Ownership and Borrowing

Alice

Bob

```
1 fn main() {
2     let alice = vec![1, 2, 3];
3     {
4         let bob = alice; // alice is moved to bob
5         println!("bob: {}", bob[0]);
6     }
7     println!("alice: {}", alice[0]);
8 }
```

# Ownership and Borrowing

Alice

Bob

```
1 fn main() {
2     let alice = vec![1, 2, 3];
3     {
4         let bob = alice;
→ 5         println!("bob: {}", bob[0]); // bob: 1
6     }
7     println!("alice: {}", alice[0]);
8 }
```

# Ownership and Borrowing

Alice

Bob

```
1  fn main() {
2      let alice = vec![1, 2, 3];
3      {
4          let bob = alice;
5          println!("bob: {}", bob[0]);
→ 6      }
7      println!("alice: {}", alice[0]);
8  }
```

# Ownership and Borrowing

```
error[E0382]: borrow of moved value: `alice`
 --> src/main.rs:7:27
  |
2 |     let alice = vec![1, 2, 3];
  |         ----- move occurs because `alice` has type `std::vec::Vec<i32>`,
which does not implement the `Copy` trait
3 |     {
4 |         let bob = alice;
  |                   ----- value moved here
...
7 |     println!("alice: {}", alice[0]);
  |                           ^^^^^ value borrowed here after move
```

```
1  fn main() {
2      let alice = vec![1, 2, 3];
3      {
4          let bob = alice;
5          println!("bob: {}", bob[0]);
6      }
→ 7    println!("alice: {}", alice[0]);
8  }
```

# Ownership and Mutable Borrowing

```rust
// Valid example; bob mutates alice's owned object
fn main() {
    let mut alice = 1;
    {
        let bob = &mut alice;
        *bob = 2;
        println!("bob: {}", bob);
    }
    println!("alice: {}", alice);
}
```

The lifetime of a borrowed reference must end before the lifetime it's owner

*what happens if it doesn't...*

# Memory Safety Gaurentees

## How Does Rust Guarantee Memory Safety?

- Ownership
- Borrowing
- Lifetimes
- Unsafe

# Lifetimes

- Pointers to values have a limited [valid] duration, known as a **lifetime**
  - Lifetimes are *statically tracked*
- All pointers to all values are known **statically**

# Lifetimes tl;dr

| | | | |
|---|---|---|---|
| Ownership | `T` | "owned" | ~~*Aliasing*~~ + *Mutation* |
| Exclusive Access | `&mut T` | "mutable" | ~~*Aliasing*~~ + *Mutation* |
| Shared Access | `&T` | "read-only" | *Aliasing* + ~~*Mutation*~~ |
| Corrupted | ? | $ | *Aliasing* + *Mutation* |

# Memory Safety Gaurentees

## How Does Rust Guarantee Memory Safety?

- Ownership
- Borrowing
- Lifetimes
- Unsafe

# Unsafe

All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time.

However, Rust has a **second language** hiding inside of it that does not enforce these memory safety guarantees:

unsafe Rust

$\rightarrow$ works just like regular Rust, but gives you extra superpowers

# The Hidden [Unsafe] Language

# Unsafe Superpowers

In rust, `unsafe` code simply means code whose security cannot be verified by the compiler

- Dereference a raw pointer
- Access or modify a mutable static variable
- Call an unsafe function or method
- Implement an unsafe trait

# Dereference a raw pointer

```
unsafe {
    let address = 0x012345usize;
    let r = address as *const i32;
}
```

Can produce Unbounded Lifetimes
(leading to Arbitrary RW)

# Unsafe Superpowers

- Dereference a raw pointer
- Access or modify a mutable static variable
- Call an unsafe function or method
- Implement an unsafe trait

# Access or modify a mutable static variable

```
1  static mut COUNTER: u32 = 0;
2  fn add_to_count(inc: u32) {
3      unsafe {
4          COUNTER += inc;
5      }
6  }
7
8  fn main() {
9      add_to_count(3);
10     unsafe {
11         println!("COUNTER: {}", COUNTER);
12     }
13 }
```

Can lead to data races

# Unsafe Superpowers

- Dereference a raw pointer
- Access or modify a mutable static variable
- Call an unsafe function or method
- Implement an unsafe trait

# Call an unsafe function or method

```
1 unsafe fn dangerous() {
2     let address = 0x012345usize;
3     let r = address as *const i32;
4 }
5
6 fn main() {
7     unsafe { dangerous(); }
8 }
```

Called function may cause undefined behavior

# Unsafe Superpowers

- Dereference a raw pointer
- Access or modify a mutable static variable
- Call an unsafe function or method
- Implement an unsafe trait

# Implement an unsafe trait

```
1  unsafe trait Foo {
2      fn new(name: &'static i32) -> Self;
3  }
4
5  unsafe impl Foo for i32 {
6      extern C {
7          fn abs(input: &self) -> i32
8      }
9  }
10
11 fn main() {
12     let val = Foo::new(5);
13     println!("Abs: {}", val.abs())
14 }
```

Called traits may cause undefined behavior

# Unsafe, Unsound, Unbounded, Undefined

→ Unsafe leads to unsound or unbounded

→ Unsound/Unbounded leads to undefined

→ Undefined leads to the dark side of the force ($$$)

# Un[.*]

- Unsafe
- Unsound
- Undefined
- Unbounded
- Implied Bounds

# Un[.*]

- Unsafe
- Unsound
- Undefined
- Unbounded
- Implied Bounds $\leftarrow$ ?

# Unsound Implied Bounds

One of the few (long standing) soundness holes in the rust type system

*bonus: it's unrelated to* `unsafe`

$\rightarrow$ A combination of contravariance and implied bounds for *nested function references* allows one to convert a **reference lifetime** to the **static lifetime** (bypassing the borrow checker)

```
1 static S: &'static &'static () = &&();
2
3 fn foo<'a, 'b, T>(_: &'a &'b (), v: &'b T) -> &'a T { v }
4
5 fn bad<'a, T>(x: &'a T) -> &'static T {
6     let f: fn(_, &'a T) -> &'static T = foo;
7     f(S, x)
8 }
```

# Unsound Implied Bounds

One of the few (long standing) soundness holes in the rust type system

*bonus: it's unrelated to* `unsafe`

→ A combination of contravariance and implied bounds for *nested function references* allows one to convert a **reference lifetime** to the **static lifetime** (bypassing the borrow checker)

```
1 static S: &'static &'static () = &&();
2
3 fn foo<'a, 'b, T>(_: &'a &'b (), v: &'b T) -> &'a T { v }
4
5 fn bad<'a, T>(x: &'a T) -> &'static T {
6     let f: fn(_, &'a T) -> &'static T = foo;
7     f(S, x)
8 }
```

Let's break this down

# An Aside: Bounds

*Lifetime* and *trait bounds* provide a way for generic items to restrict their parameters based on **types** and **liftimes**

- Lifetime Bounds

```
1  // 'a: 'b === lifetime 'a outlives 'b
2  fn f<'a, 'b>(x: &'a i32, mut y: &'b i32) where 'a: 'b {
3      // &'a i32 is a subtype of &'b i32 because 'a: 'b
4      y = x;
5
6      // &'b &'a i32 is well formed because 'a: 'b
7      //
8      // (human-readable) type:
9      //    a reference of liftime 'b pointing
10     //    to a 32-bit integer reference bound
11     //    for lifetime 'a
12     let r: &'b &'a i32 = &&0;
13 }
```

# An Aside: Bounds

*Lifetime* and *trait bounds* provide a way for generic items to restrict their parameters based on **types** and **liftimes**

- Trait Bounds

```
// Restricts our generic funcuntion to
// only accept values of type Fn(&'a i32)
fn call_ref_zero<F>(f: F) where for<'a> F: Fn(&'a i32) {
    let zero = 0;
    f(&zero);
}
```

# Implied Bounds

Removes the need to repeat `where` clauses for type or trait declarations

```
1  struct HashSet<K: Hash> { ... }
2
3  // Here we explicitly set our trait bound; when
4  // used, explicit bounds take logical precedence
5  //
6  // The impl fn's inherit [im,ex]plicit K: Hash bound
7  impl<K> HashSet<K> { ... }                  // Implicit bound, or
8  impl<K> HashSet<K> where K: Hash { ... } // Explicit Bound
9
10 // Here we omit our `where` clause as K: Hash is
11 // implicity bound whenever HashSet<K> is an input type
12 fn loud_insert<K>(set: &mut HashSet<K>, item: K) {
13     println!("inserting!");
14     set.insert(item);
15 }
```

# Unsound Implied Bounds (Cont.)

```
1  static S: &'static &'static () = &&(); // reference lifetime
2
3  // Returns reference for type T of lifetime 'a
4  fn foo<'a, 'b, T>(_: &'a &'b (), v: &'b T) -> &'a T { v }
5
6  // Unsound --> implied bound computed for return type
7  //
8  // Returns reference for the implied type bound
9  fn bad<'a, T>(x: &'a T) -> &'static T {
10     // This is where our soundness hole is created
11     //
12     // Below we use contravariance to assign foo to f,
13     // allowing us to side-step our obligation to the rust
14     // borrow checker, i.e., to prove the implicitly set
15     // assertion that 'a: 'static ('a outlives 'static)
16     let f: fn(_, &'a T) -> &'static T = foo;
17     f(S, x) // Undefined behavior
18 }
```

# Can you spot the bug?

```
1  fn gn_fn<'a, 'b, T: ?Sized> (_: &'a &'b (), v: &'b mut T)
2  -> &'a mut T { v }
3
4  fn unsound_ib<'a, T: ?Sized> (input: &'a mut T)
5  -> &'static mut T {
6      let f: fn(_, &'a mut T) -> &'static mut T = gn_fn;
7      f(&&(), input)
8  }
9
10 // Returns Vec<u8> of length `size`
11 fn monster (size: usize) -> &'static mut [u8] {
12     // (heap-)allocation for `size` repetitions of 0x41
13     let mut object = vec![b'A'; size];
14     // Call our unsound function --> Returned as 'static
15     let r = unsound_ib(object.as_mut());
16     r
17 }
```

# Can you spot the bug? (Cont.)

```
1
2                                   [ ... ]
3  fn mash (
4      lines: &mut impl Iterator<Item = std::io::Result<String>>
5  ) -> &'static mut [u8] {
6      println!("How many monsters must mash?");
7
8      let line = lines.next().unwrap().unwrap();
9      let m_size = line.parse::<usize>().unwrap();
10
11     mash(m_size)
12 }
13
14 fn main() { // See gist for full source
15     let stdin = stdin();
16     let mut lines = stdin.lock().lines();
17     let monsters = Some(monster(&mut lines));
18 }
```

# It was a graveyard smash!

`nc apollo.bynx.io 31337`

`main.rs` source code

CLICK ME

*fin*