



**little languages**

**lecture 04:**

# **Stack Values and References**

Draw the state of memory of the Java program below when it arrives at the "breakpoint" comment.

```
public class ABCs {  
    public static void main(String[] args) {  
        int a = 0;  
        int b = a;  
        b = 1;  
  
        int[] e = { 2 };  
        int[] f = e;  
        f[0] = 3;  
        // breakpoint  
    }  
}
```

# Environment Diagrams

- In this course you will draw environment diagrams to represent a program's state in memory
  - These diagrams are a powerful tool to help you intuitively understand scopes, values, references, and the stack versus the heap
- You can apply them to any language however each language will have its own rules regarding what's possible (more precisely called ***semantics***)
- A key distinction between *systems programming languages* and *applications languages* is how "close to memory" your code can operate
  - In *systems languages* like C, C++, and Rust you have direct access to memory
  - In *applications languages* there are abstractions sitting between your code's variables and memory that limit what you're able to express

# Aside: Stack versus Heap in Java and JavaScript

- The values of variables holding value-type data live on the *call stack*
  - The local variables of each function call or method call reside in their own *frame*
- The values of variables holding reference-type data lives on *the heap*
  - Local variables "storing" an object or array are actually pointers to locations on the heap
  - Every time you use the *new* keyword in Java you're allocating storage space on the heap


# Important Semantics of Common Application Languages

- Many popular applications languages (including Java, JavaScript, Python) share similar semantics around *value* and *reference* types
- There are ideas you *cannot express* in an applications programming language because it has semantic abstractions between your variables and memory
  - For example, if you declare a *value type* variable in Java the *only way* to change its value is to assign to the variable directly.
  - Why? It is *not possible* for one variable on the stack to reference another via a pointer.
    - All references/pointers are directed toward heap locations.
- Heap objects remain present for at least as long as there is a way to reach them from *the stack* by following pointers
  - Some amount of time after a heap object becomes impossible to reach from the stack, the language runtime's *garbage collector* will reclaim the memory.

# Environment Diagrams in Rust

Stack

Heap



```
fn main() {  
    let a: u8 = 1;  
    a_fun(a);  
}  
  
fn a_fun(a: u8) {  
    let b: u8 = 2;  
    let c: u8 = 3;  
    println!("{}", a + b + c);  
    // BREAKPOINT  
}
```

main

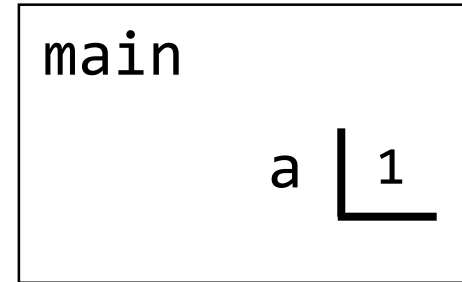
Each function call has its own stack frame for local variables.

Environment Diagrams:

Variables declarations added to the current frame.

```
fn main() {  
  → let a: u8 = 1;  
    a_fun(a);  
}  
  
fn a_fun(a: u8) {  
  let b: u8 = 2;  
  let c: u8 = 3;  
  println!("{}", a + b + c);  
  // BREAKPOINT  
}
```

Stack



Heap



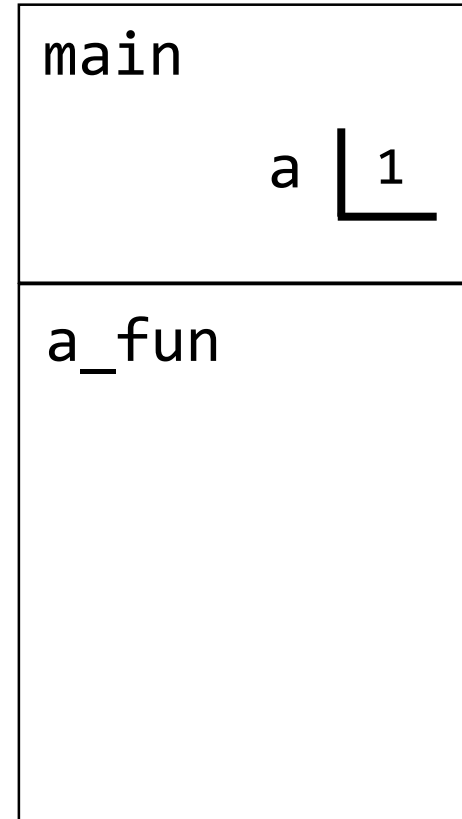
## Environment Diagrams:

Function calls allocate a new frame on the stack.

```
fn main() {  
  let a: u8 = 1;  
  → a_fun(a);  
}  
  
fn a_fun(a: u8) {  
  let b: u8 = 2;  
  let c: u8 = 3;  
  println!("{}", a + b + c);  
  // BREAKPOINT  
}
```

Stack

Heap



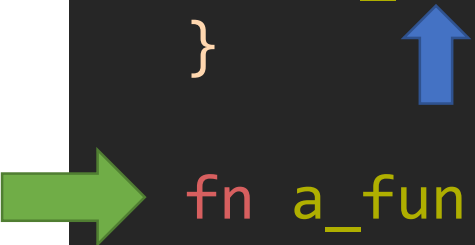
In this instance the frame size is larger because looking ahead toward `a_fun` you can see it has a parameter and 2 local variables so we'll need room for additional variables in the frame.



Environment Diagrams:

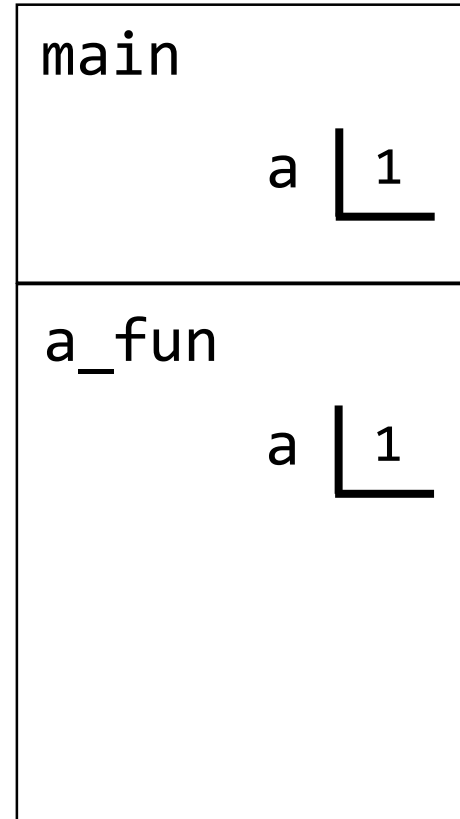
Function calls drop a bookmark to return to and jump into function.

```
fn main() {  
    let a: u8 = 1;  
    a_fun(a);  
}  
  
fn a_fun(a: u8) {  
    let b: u8 = 2;  
    let c: u8 = 3;  
    println!("{}", a + b + c);  
    // BREAKPOINT  
}
```



Stack

Heap




Notice it's just a coincidence a\_fun's parameter name is the same as the variable it was given in the function call. These are two separate, unrelated variables in memory. Either could have a different name.

Environment Diagrams:

More variables are declared and initialized.

```
fn main() {  
    let a: u8 = 1;  
    a_fun(a);  
}  
  
fn a_fun(a: u8) {  
    let b: u8 = 2;  
    let c: u8 = 3;  
    println!("{}", a + b + c);  
    // BREAKPOINT  
}
```



Stack

main	
a	1
a_fun	
a	1
b	2
c	3

Heap

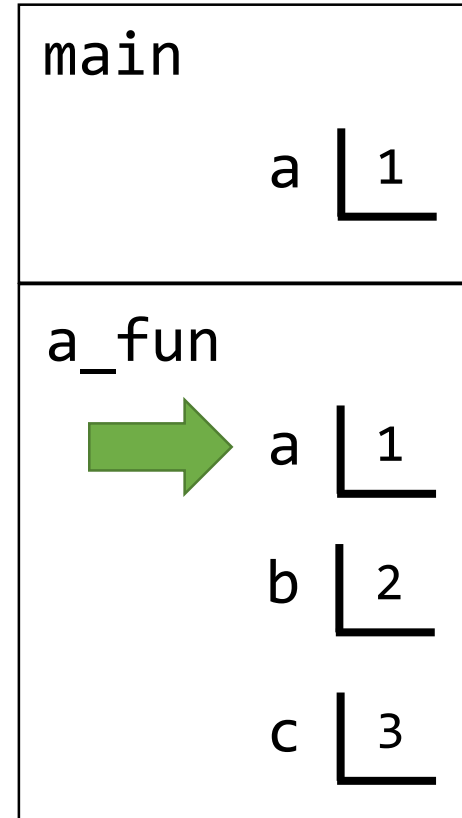
In reality these steps would happen one at a time, we've fast forwarded here for brevity.

## Environment Diagrams:

Variable accesses look up values in the current scope.

```
fn main() {  
  let a: u8 = 1;  
  a_fun(a);  
}  
  
fn a_fun(a: u8) {  
  let b: u8 = 2;  
  let c: u8 = 3;  
  println!("{}", a + b + c);  
  // BREAKPOINT  
}
```

Stack



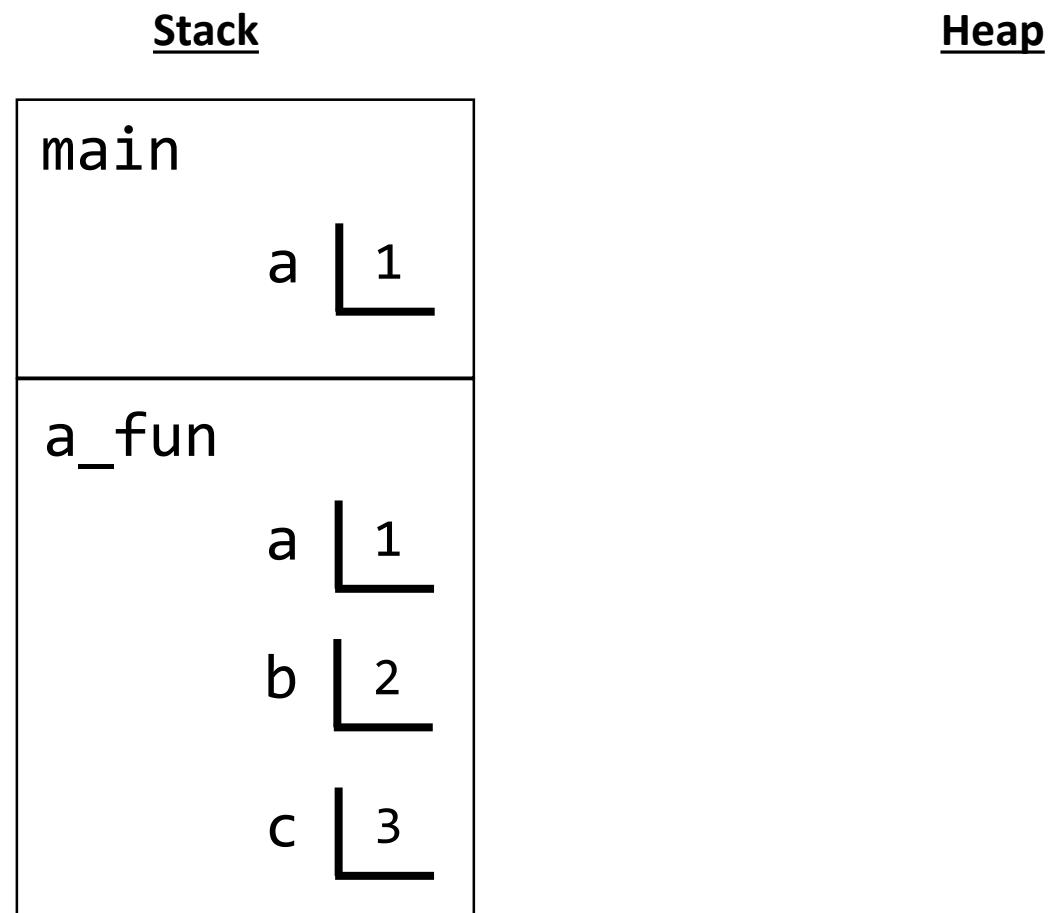
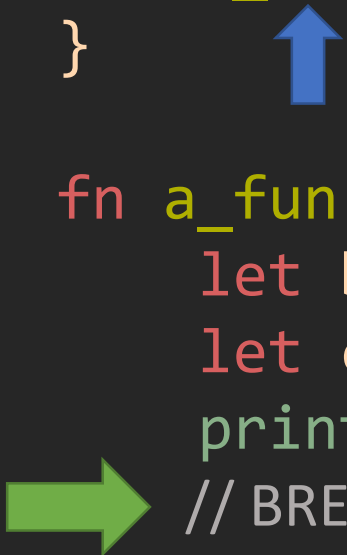
Heap

Note that the two `a` variables currently on the stack *could easily* have different values.

## Environment Diagrams:

When the breakpoint is reached the stack and heap are as so...

```
fn main() {  
    let a: u8 = 1;  
    a_fun(a);  
}  
  
fn a_fun(a: u8) {  
    let b: u8 = 2;  
    let c: u8 = 3;  
    println!("{}", a + b + c);  
    // BREAKPOINT  
}
```




All of this behaves just the same as it would in an applications language like Java. No surprises here. We're not going to digress into the `println!` macro for now. Lots of wonderful magic is happening there.

# Environment Diagrams with References

Environment Diagrams with References:

Main starts empty...



```
fn main() {  
    let a: u8 = 0;  
    let b: u8 = 1;  
    let p: &u8 = &b;  
    println!("{:p}", p);  
    println!("{}", *p);  
    // BREAK  
}
```

Stack



Heap

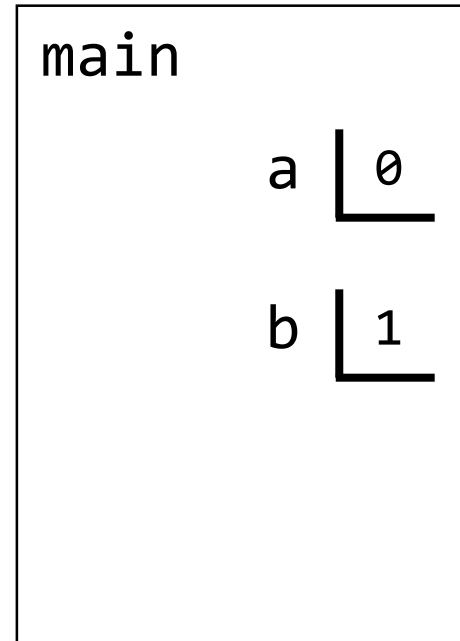
Environment Diagrams with References:

Variable declarations and initializations add entries to current frame.

```
fn main() {  
  let a: u8 = 0;  
  → let b: u8 = 1;  
  let p: &u8 = &b;  
  println!("{:p}", p);  
  println!("{}", *p);  
  // BREAK  
}
```

Stack

Heap



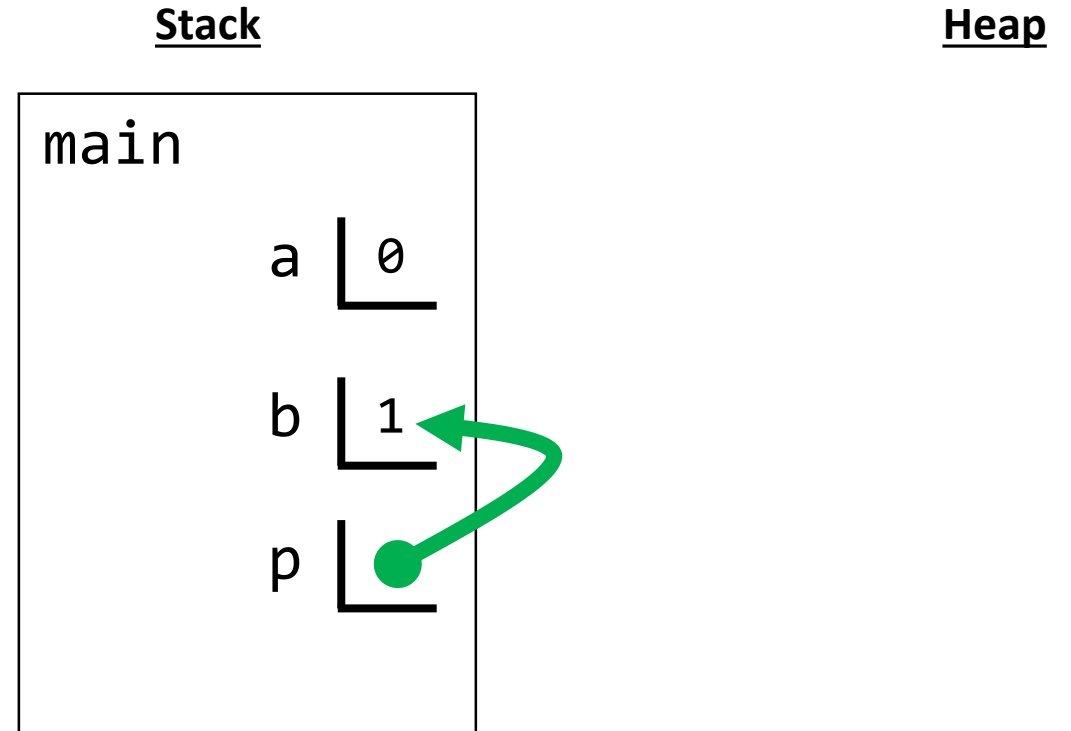
Nothing new and surprising, yet...



## Environment Diagrams with References:

References types are assigned pointers to locations in memory.

```
fn main() {  
  let a: u8 = 0;  
  let b: u8 = 1;  
  let p: &u8 = &b;  
  println!("{:p}", p);  
  println!("{}", *p);  
  // BREAK  
}
```



Notice we have a pointer from one location on the stack to another! This *isn't possible* in Java.

# Breaking down a reference assignment...

```
let p: &u8 = &b;
```



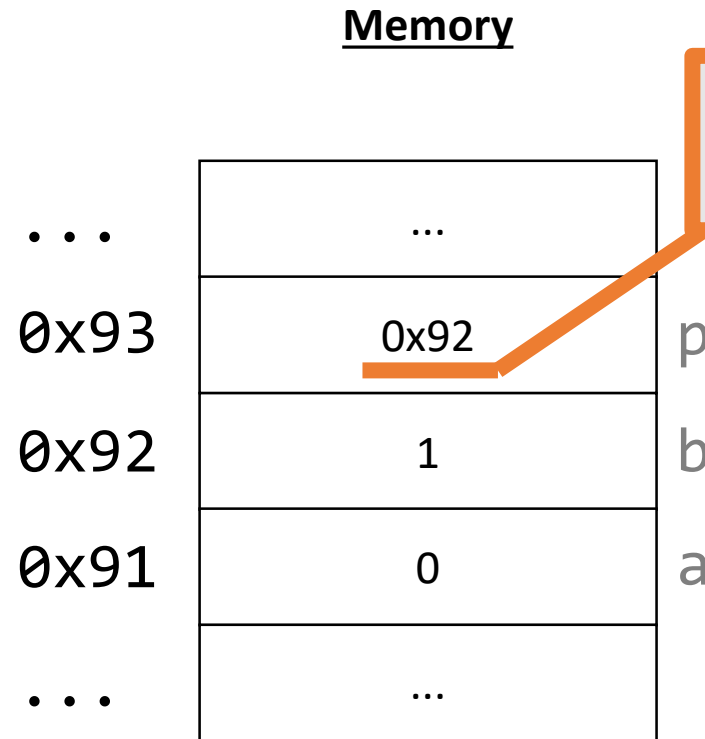
The *type* is a **reference** to an unsigned 8-bit integer.

The value assigned is the **address of** the variable **b**.

- What does the address of operator actually resolve to?
  - The *memory address* of a variable... aka **a pointer**.
  - A pointer is a memory address. We draw an arrow *only as a visualization*.

Reality looks a little bit more like this....

```
fn main() {  
    let a: u8 = 0;  
    let b: u8 = 1;  
    let p: &u8 = &b;  
    println!("{:p}", p);  
    println!("{}", *p);  
    // BREAK  
}
```



Notice p stores the address of b in memory.

There are still many oversimplifications present in this representation, however, it illustrates the main concept you need to know: a pointer is just an address of some place in memory.


We'll soon discuss slices and "smart pointers" in more detail. They expand on the idea of a simple pointer and store additional information about the memory being pointed to.

Environment Diagrams with References:

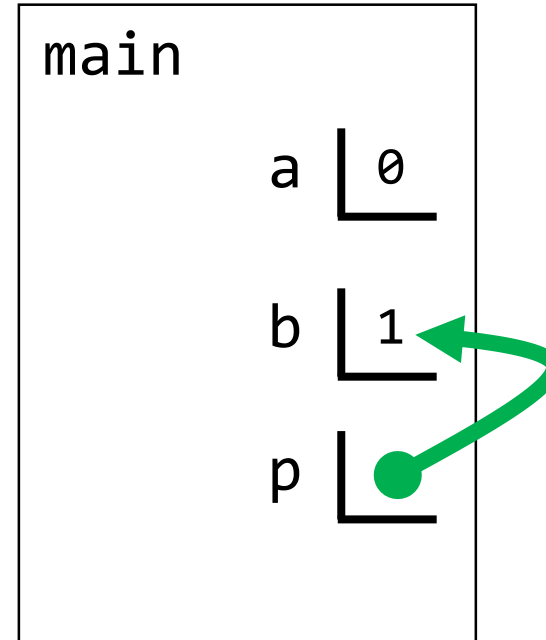
You can print the address of a reference type using a `{:p}` placeholder.

Stack

Heap



```
fn main() {  
    let a: u8 = 0;  
    let b: u8 = 1;  
    let p: &u8 = &b;  
    println!("{:p}", p);  
    println!("{}", *p);  
    // BREAK  
}
```



Notice that **p** itself is of type **&u8** which is a reference type holding the address of another location.


Environment Diagrams with References:

You can *dereference* a reference type with a `*`.

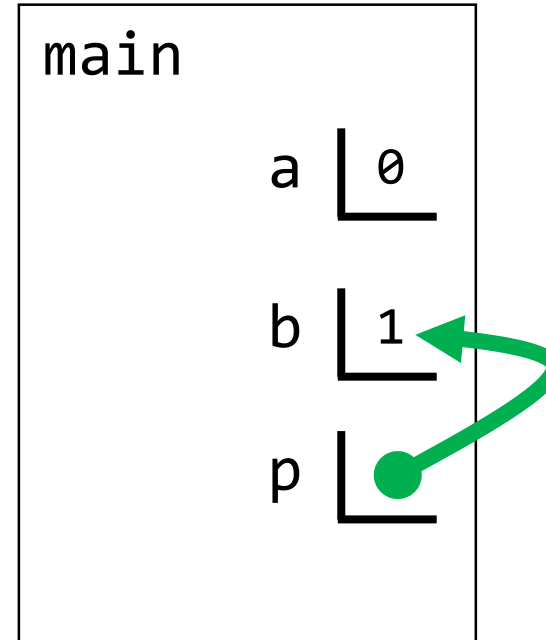
This means: "follow the pointer"

Stack

Heap



```
fn main() {  
    let a: u8 = 0;  
    let b: u8 = 1;  
    let p: &u8 = &b;  
    println!("{:p}", p);  
    println!("{}", *p);  
    // BREAK  
}
```



When you dereference a pointer you're accessing the value stored in the address the pointer holds.

There are common situations where Rust will automatically dereference on your behalf. You can always do it explicitly as shown here, though. While you're getting comfortable with the concept we encourage being explicit.

# Environment Diagrams with Mutable Variables and Mutable References


```
fn main() {  
    let mut a: u8 = 0;  
    let mut b: u8 = 1;  
  
    let mut p: &mut u8 = &mut a;  
    *p = 10;  
  
    p = &mut b;  
    *p = 11;  
  
    println!("a:{} - b:{}", a, b);  
    // BREAK  
}
```

[PollEv.com/compunc](https://PollEv.com/compunc) – what is the output of this code listing?



## Environment Diagrams with Mutable References:

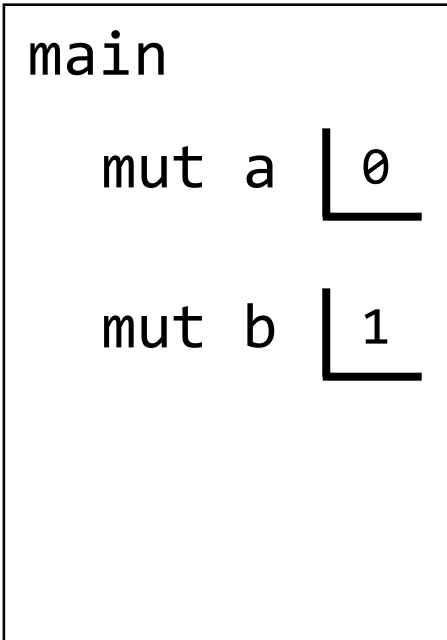
A variable's contents are mutable only if declared **mut**.



```
fn main() {  
  let mut a: u8 = 0;  
  let mut b: u8 = 1;  
  
  let mut p: &mut u8 = &mut a;  
  *p = 10;  
  
  p = &mut b;  
  *p = 11;  
  
  println!("a:{} - b:{}", a, b);  
  // BREAK  
}
```

Stack

Heap




Why all this concern over mutability?

In single-threaded programming, actively avoiding mutability makes it easier to reason about code.

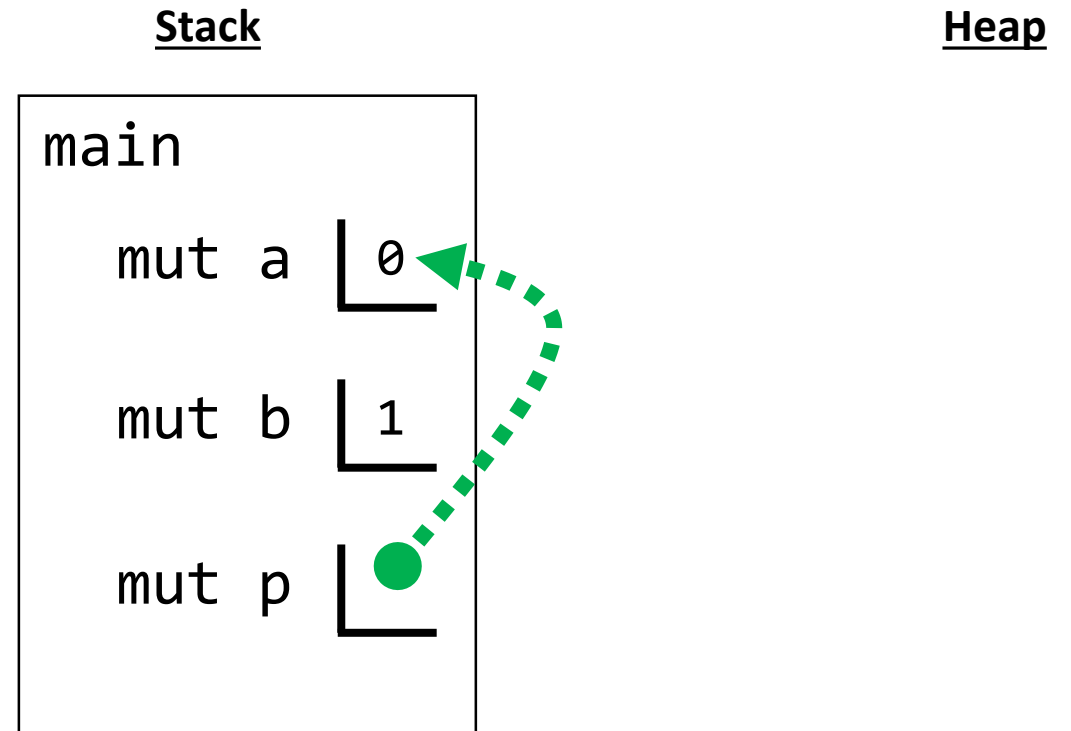
In multi-threaded programming, mutability is the root cause of almost every problem you'll encounter.

## Environment Diagrams with Mutable References:

If you'd like for a reference to be able to mutate its referent, it must be assigned an `&mut` pointer.



```
fn main() {  
  let mut a: u8 = 0;  
  let mut b: u8 = 1;  
  
  let mut p: &mut u8 = &mut a;  
  *p = 10;  
  
  p = &mut b;  
  *p = 11;  
  
  println!("a:{} - b:{}", a, b);  
  // BREAK  
}
```



We will use dashed lines to indicate mutable references.

There are three different meanings of `mut` in this line. Let's break them down...

# Different meanings of **mut**

```
let mut p: &mut u8 = &mut a;
```



When you declare a variable as **mutable** you can reassign that variable's value. Here we're declaring **p** as a mutable variable whose type is a mutable reference. Thus, it can be reassigned in the future.

The *type* is a **reference** to a u8 value that can be mutated by dereferencing.


A special variant of the address of operator returns the address of a while letting the compiler know it's a mutable reference.

The compiler will ensure **a** was declared mutable.

- The **mut** keyword is info the compiler uses to ensure safety guarantees.
- Once the program compiles, **mut** has no cost in space or time.

Environment Diagrams with Mutable References:

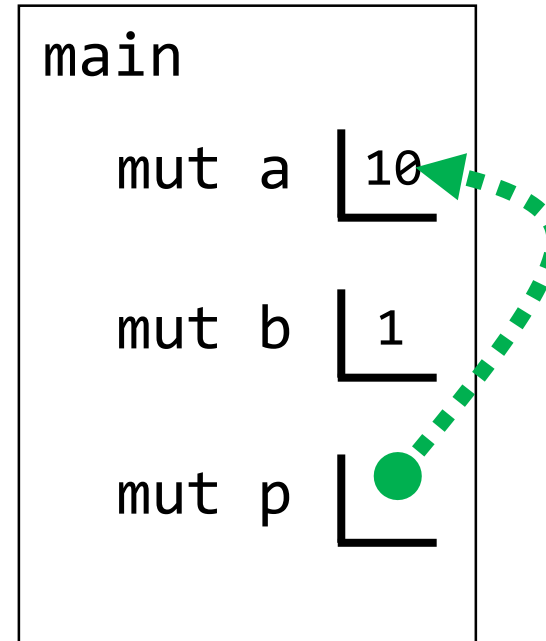
Assigning to a mutable reference changes the referent value.



```
fn main() {  
  let mut a: u8 = 0;  
  let mut b: u8 = 1;  
  
  let mut p: &mut u8 = &mut a;  
  *p = 10;  
  
  p = &mut b;  
  *p = 11;  
  
  println!("a:{} - b:{}", a, b);  
  // BREAK  
}
```

Stack

Heap



You must dereference the pointer in this scenario.

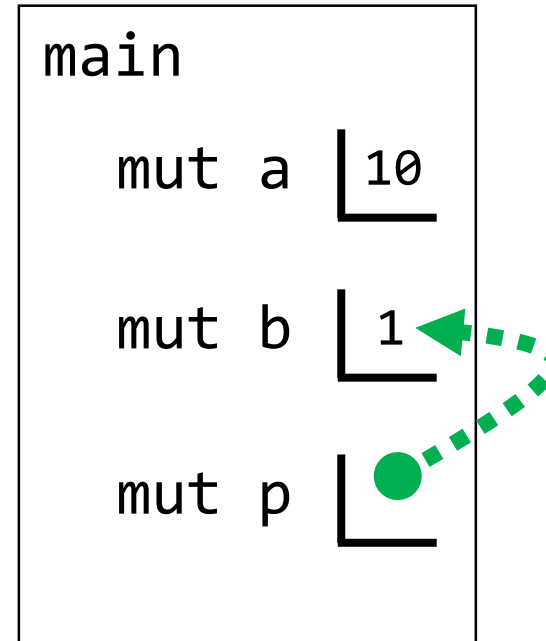
Environment Diagrams with Mutable References:

Reassigning a reference variable is possible if it was declared **mut**.

```
fn main() {  
    let mut a: u8 = 0;  
    let mut b: u8 = 1;  
  
    let mut p: &mut u8 = &mut a;  
    *p = 10;  
  
    p = &mut b;  
    *p = 11;  
  
    println!("a:{} - b:{}", a, b);  
    // BREAK  
}
```

Stack

Heap



If we had not declared **p** as a **mut** variable, this line would have been invalid.

Note: idiomatic Rust would prefer redeclaring **p** and not declaring it mutable. We're doing it this way here for illustrative purposes.

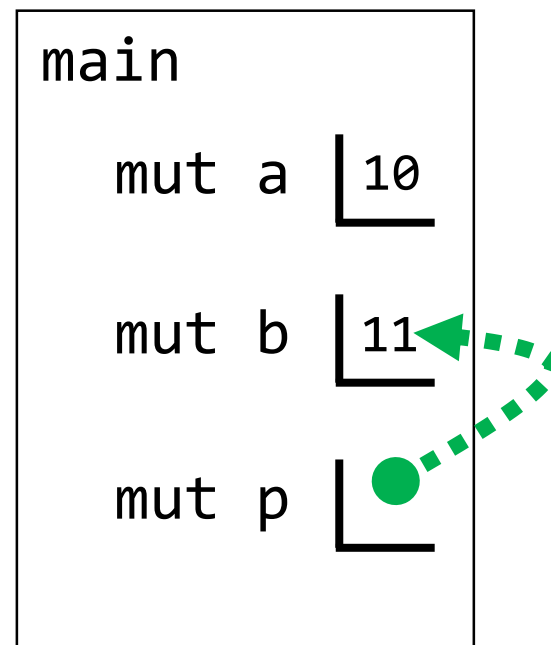
Environment Diagrams with Mutable References:

Assigning to a mutable reference changes the referent value.

```
fn main() {  
    let mut a: u8 = 0;  
    let mut b: u8 = 1;  
  
    let mut p: &mut u8 = &mut a;  
    *p = 10;  
  
    p = &mut b;  
    *p = 11;  
  
    println!("a:{} - b:{}", a, b);  
    // BREAK  
}
```

Stack

Heap



If we had not declared **p** as a **mut** variable, this line would have been invalid.

Note: idiomatic Rust would prefer redeclaring **p** and not declaring it mutable. We're doing it this way here for illustrative purposes.