The Stack and Heap

Rust, in Practice and in Theory

Lecture 3

`CAS CS 392 (M1)`

Outline

- Discuss a couple ways of managing memory
- Look at ownership rules, and how they are influenced by the layout of memory
- **Workshop:** Finish Assignment 1
- **If you finish:** slow_primes and RustViz

Memory Layout

The Punchline: Ownership

The notion of ownership is based on two simple rules

1. Every value has one owner at any given time
2. When the owner of a value goes out of scope, any memory associated with the value is freed

Areas of Memory

1. **Static Memory.** Where global variables are stored
2. **The Stack.** Where data local to a function call are stored
3. **The Heap.** Where persistent dynamically-size data are stored

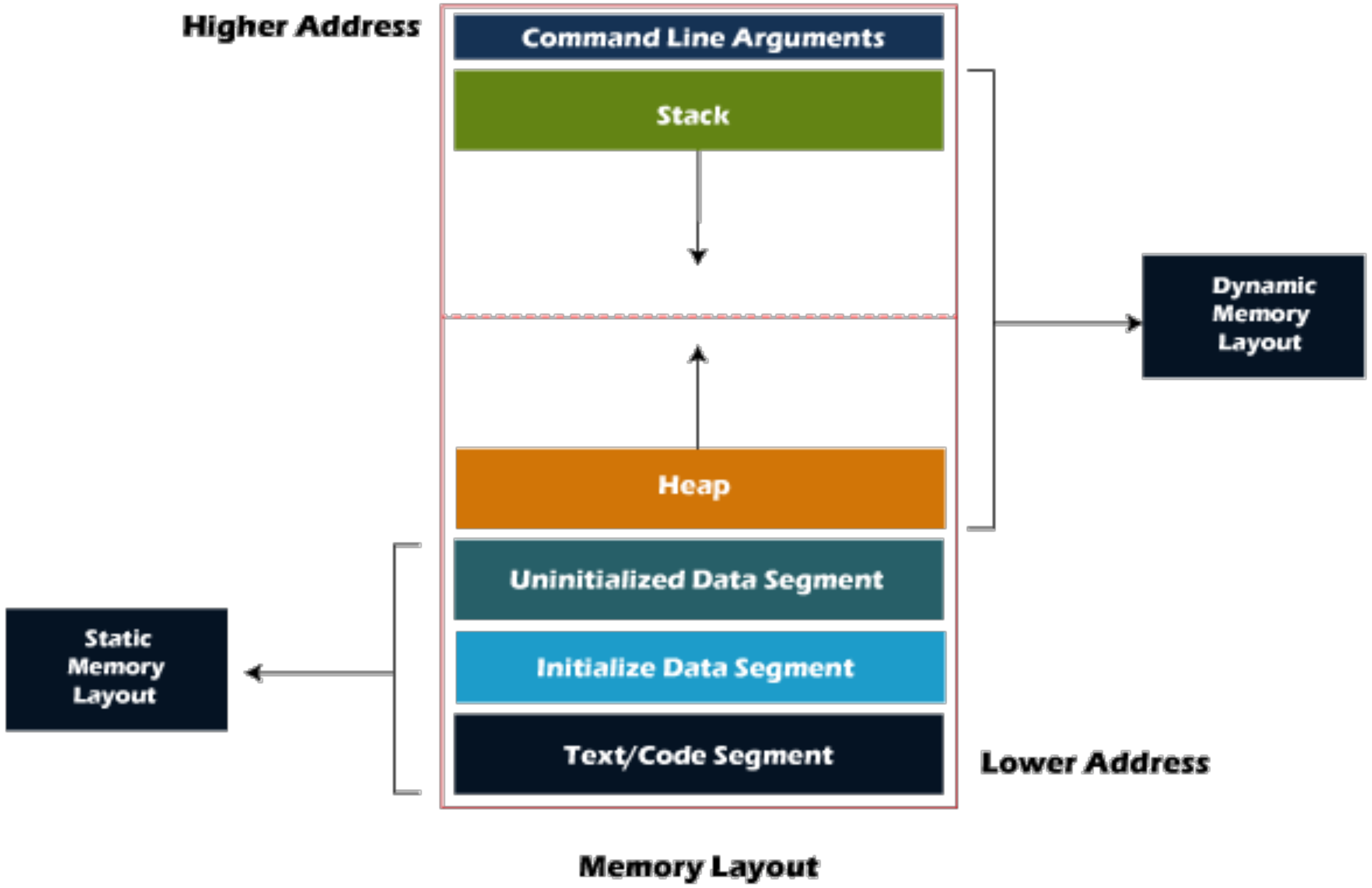
Areas of Memory

1. **Static Memory.** Where global variables are stored
2. **The Stack.** Where data local to a function call are stored
3. **The Heap.** Where persistent dynamically-size data are stored

Typical Memory Layout

The stack typically grows down and the heap grows up

The stack is very small (something like 8mb)



https://www.javatpoint.com/memory-layout-in-c

The Stack

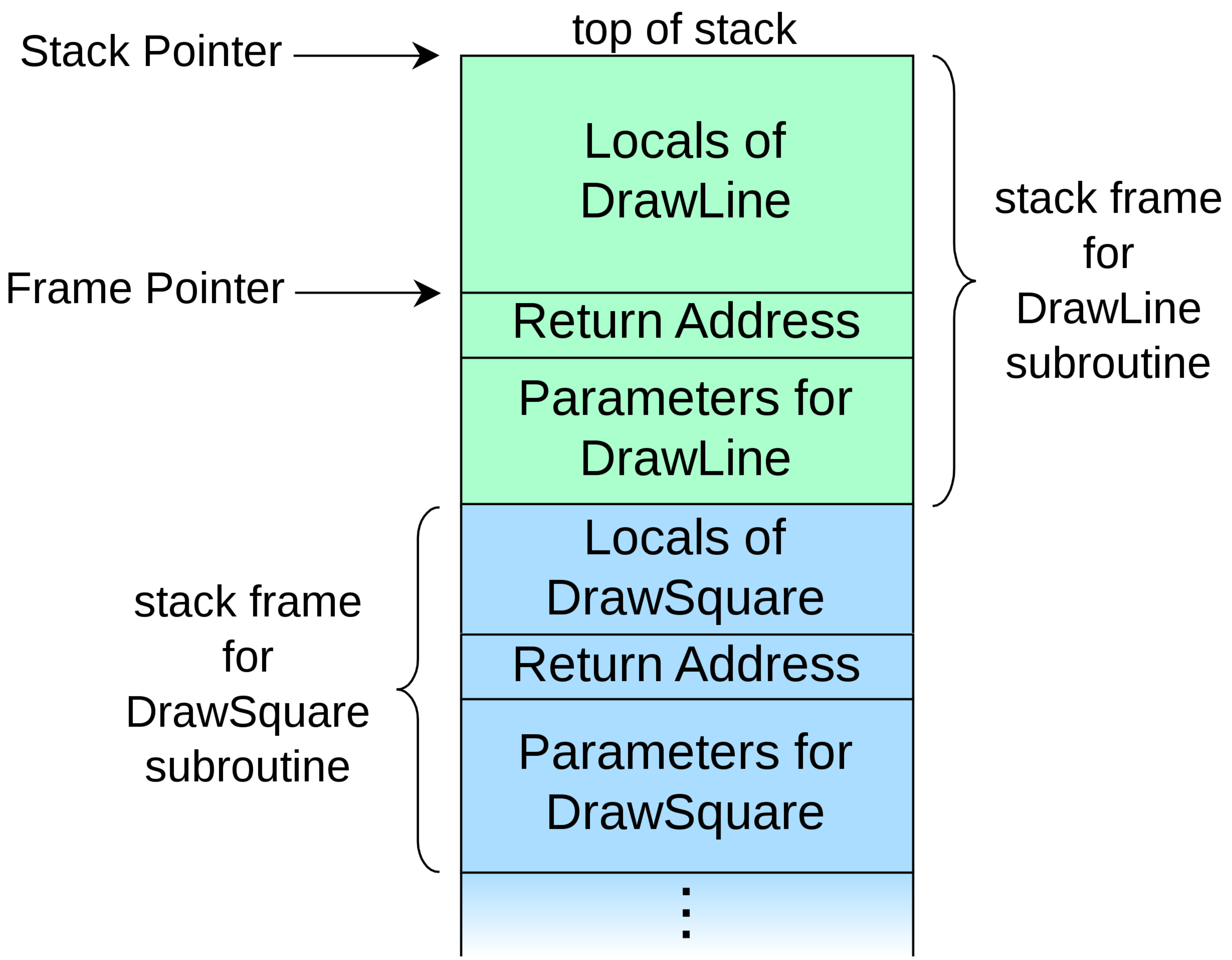
The Stack

The stack stores local variables for function calls

It can hold **activation records** or **call frames** which include extra data required by the function

It's fast to access, it's "right" there

It's well-organized, no wasted space



https://commons.wikimedia.org/wiki/File:Call_stack_layout.svg

What goes on the stack?

& str : string slice

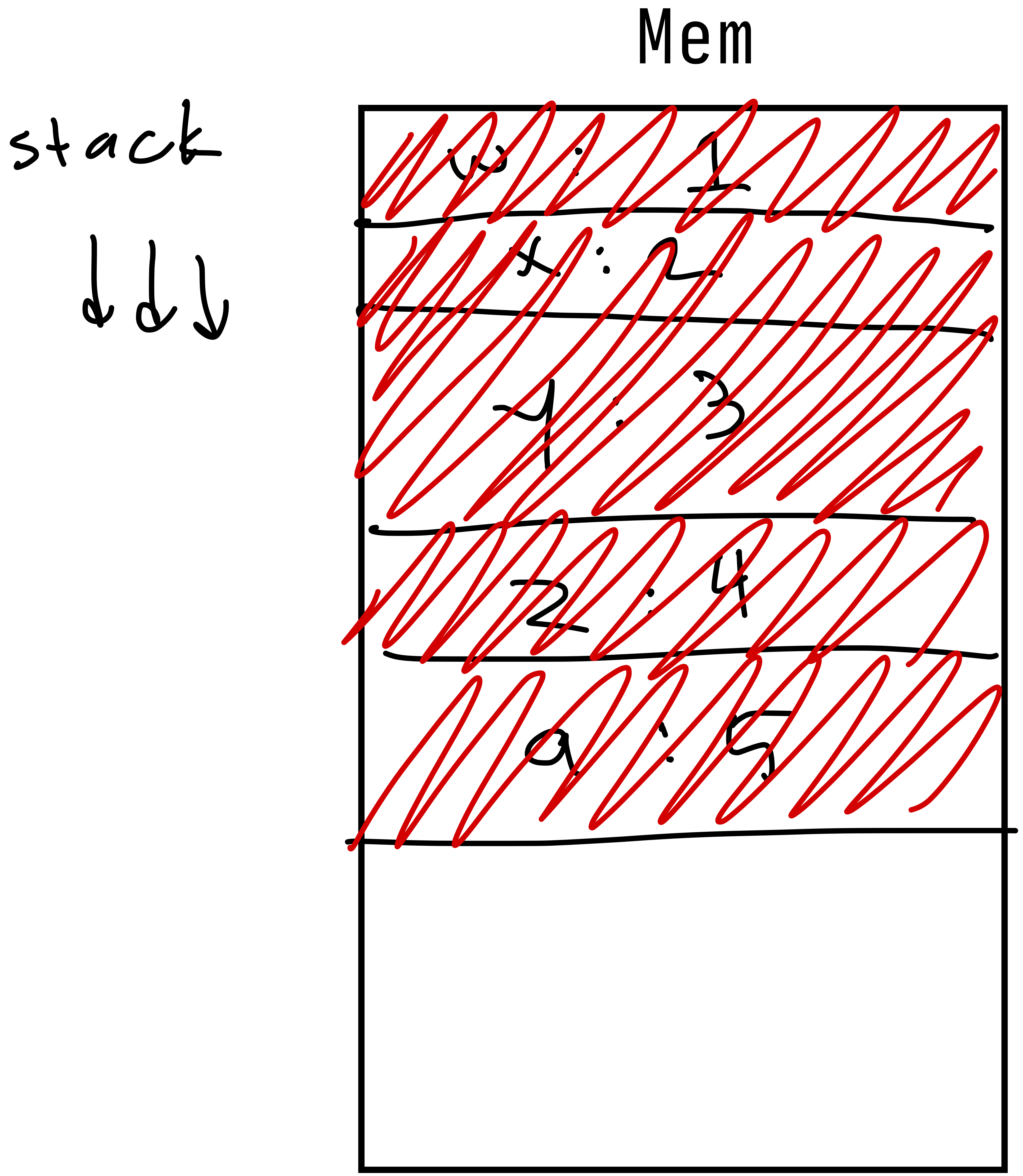
Anything whose size is fixed and known at compile time:

- primitives like numbers, string slices, arrays
- references

and which is not needed after the control is returned to the function caller

Basic Example

```
fn bar() {
    let _z = 4;
    let _a = 5;
}

fn foo() {
    let _x = 2;
    let _y = 3;
    bar();
}

fn main() {
    let _w = 1;
    foo()
}
```



The Problem

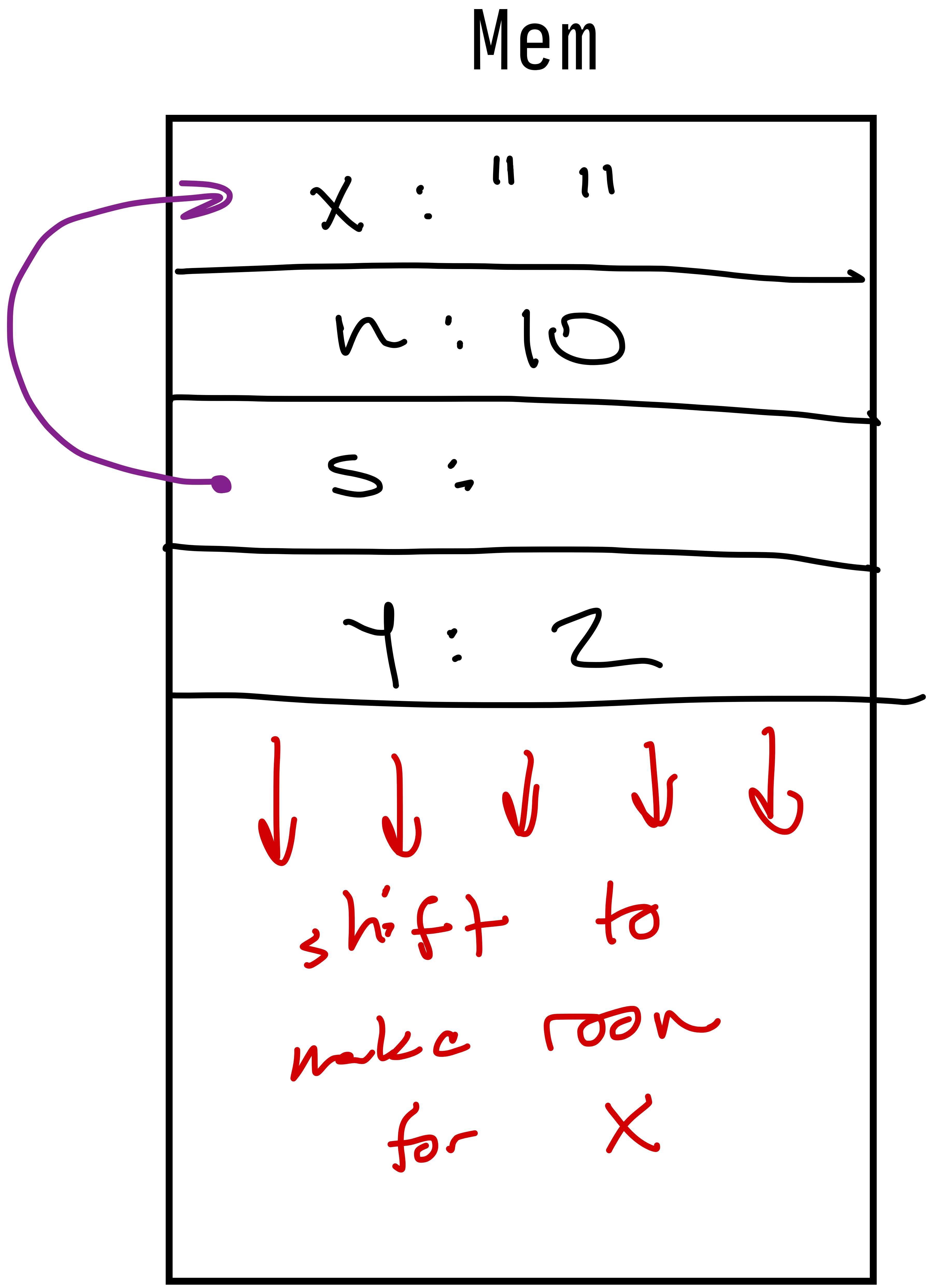
Not everything has fixed size known at compile time

We often want data we can refer to after a function call has returned control

Growing Data Example

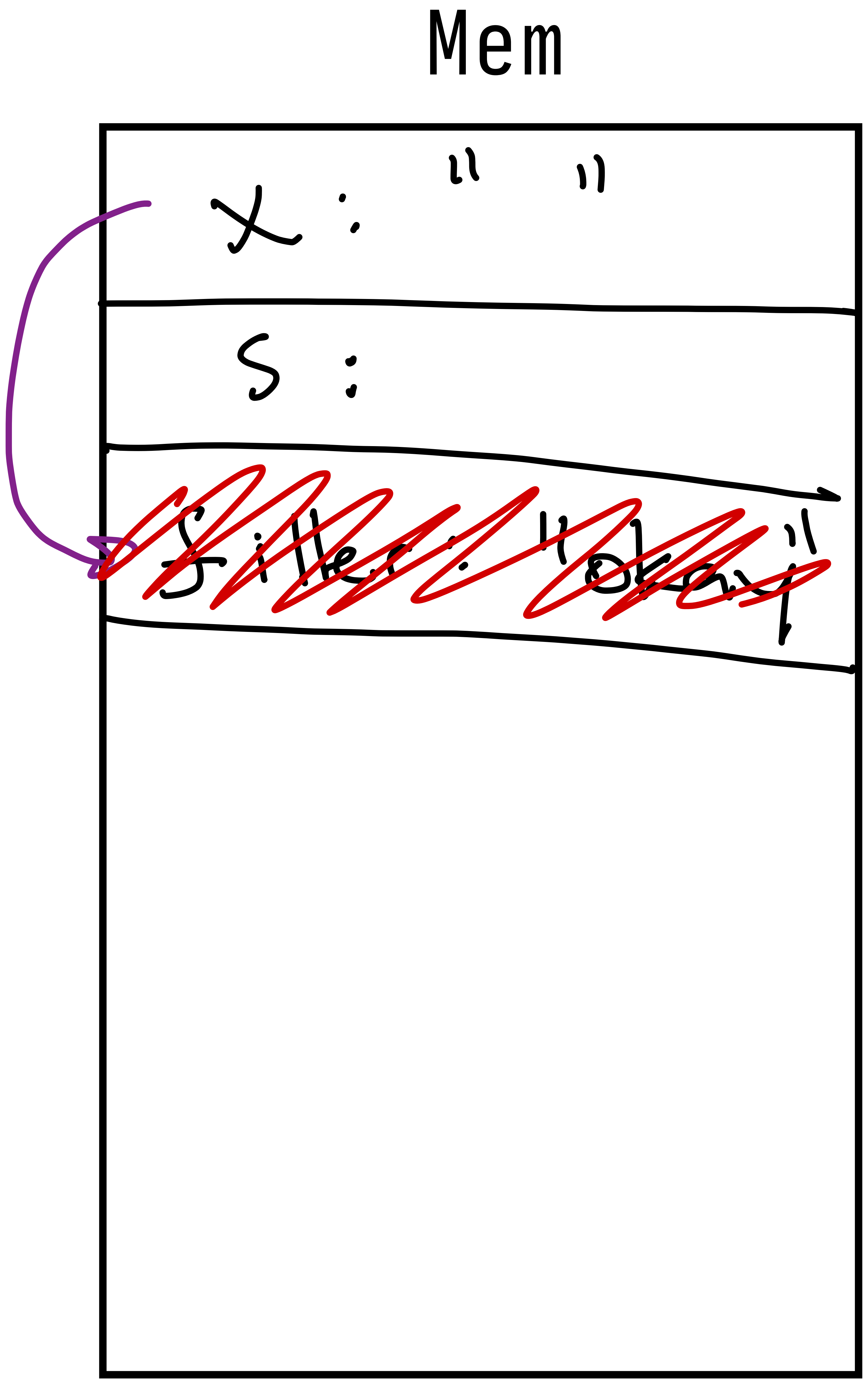
```
fn indirection(n: i32, s: &mut String) {
    let _y = 2;
    for _ in 0..n {
        *s += "okay";
    }
}

fn main() {
    let mut x : String = String::default();
    indirection(10, &mut x);
    println!("{x}");
}
```

* this is not how this works



Disappearing Data Example

```
fn fill(s : &mut String){
    let filler = ~~"Okay";~~ String::from("Okay")
    ~~*s = String::from(filler);~~
}   s= filler;

fn main() {
    let mut x : String = String::default();
    fill(&mut x);
}
```



The Heap

The Heap

The heap stores data that cannot be put on the stack (or in static memory)

It's slow to access, we have to follow *references*

It's less efficiently organized, it may become *fragmented* over time

But there's a lot of it, and it's very flexible

What goes on the heap?

Dynamically-sized persistent data:

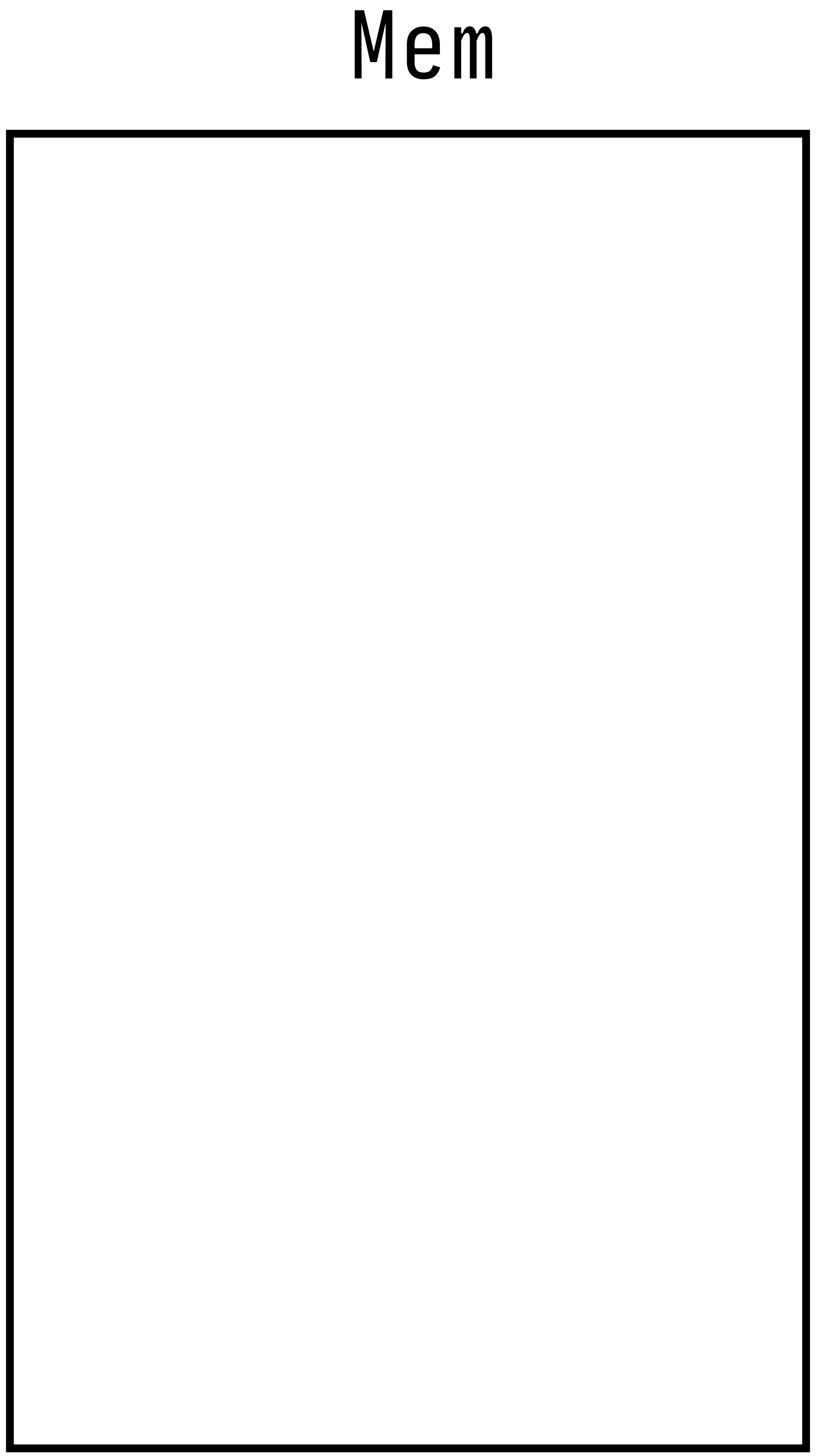
- String, Vec, Map
- pretty much everything else

We need the heap to do "real" programming

Memory Allocator

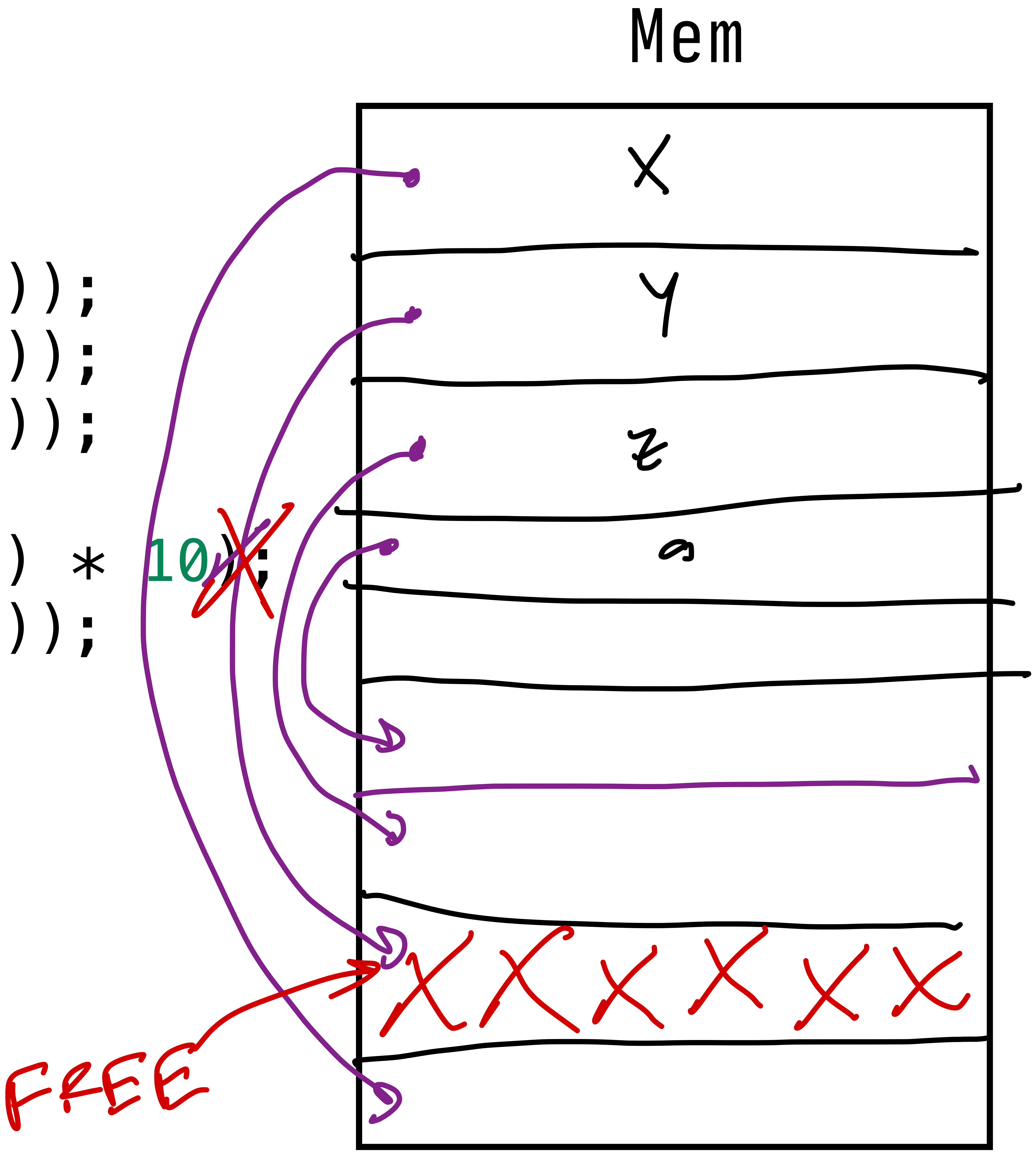
In rough terms, a memory allocator figures out how to layout data in the heap. This means:

- finding an open spot of the right size
- returning the *address* of the beginning of the spot chosen



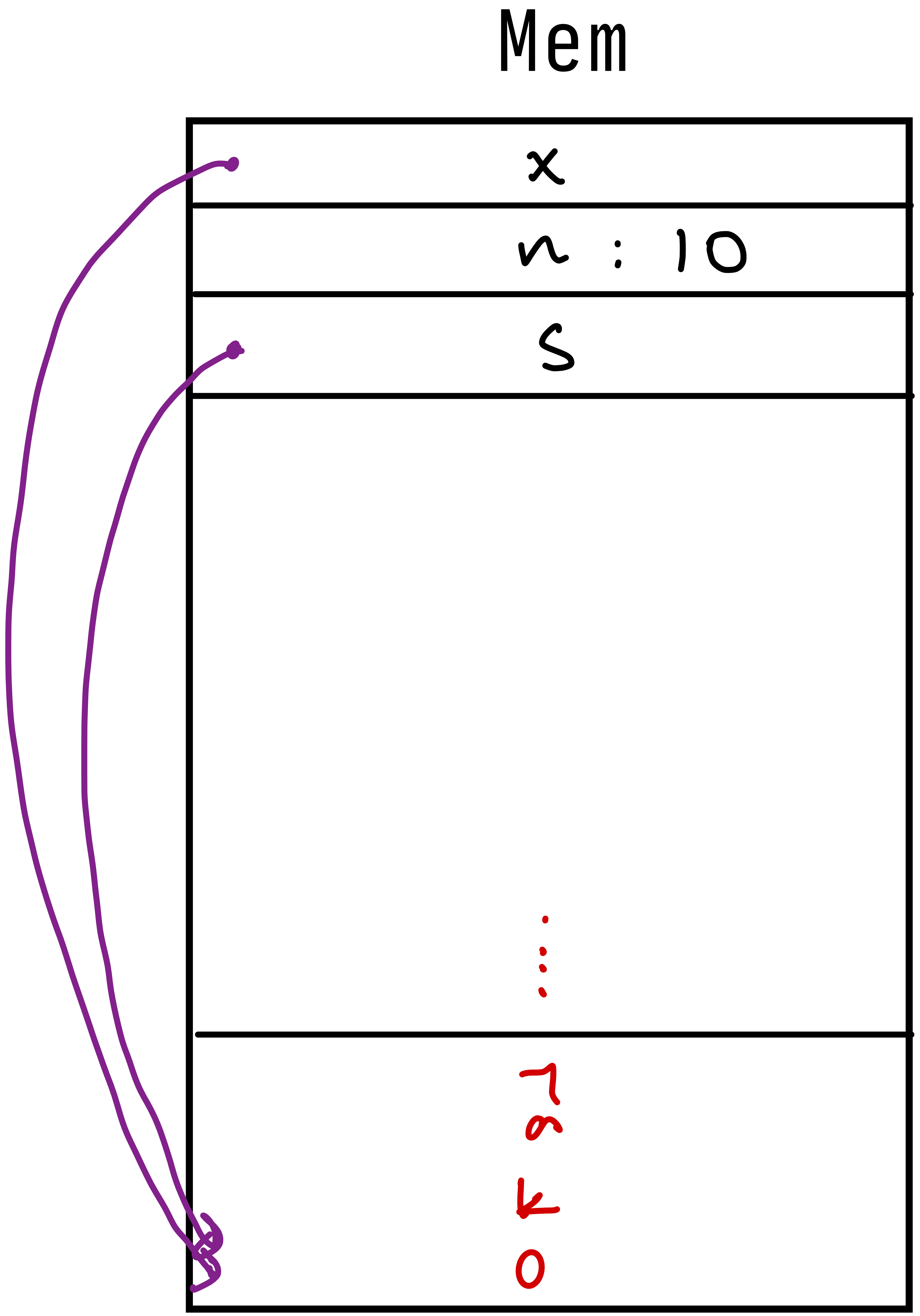
Memory Allocator

```
int main(void) {
  int *x = (int*)malloc(sizeof(int));
  int *y = (int*)malloc(sizeof(int));
  int *z = (int*)malloc(sizeof(int));
  free(y);
  int *a = (int*)malloc(sizeof(int) * 10);
  int *b = (int*)malloc(sizeof(int));
  free(x);
  free(z);
  free(a);
  free(b);
  return 0;
}
```



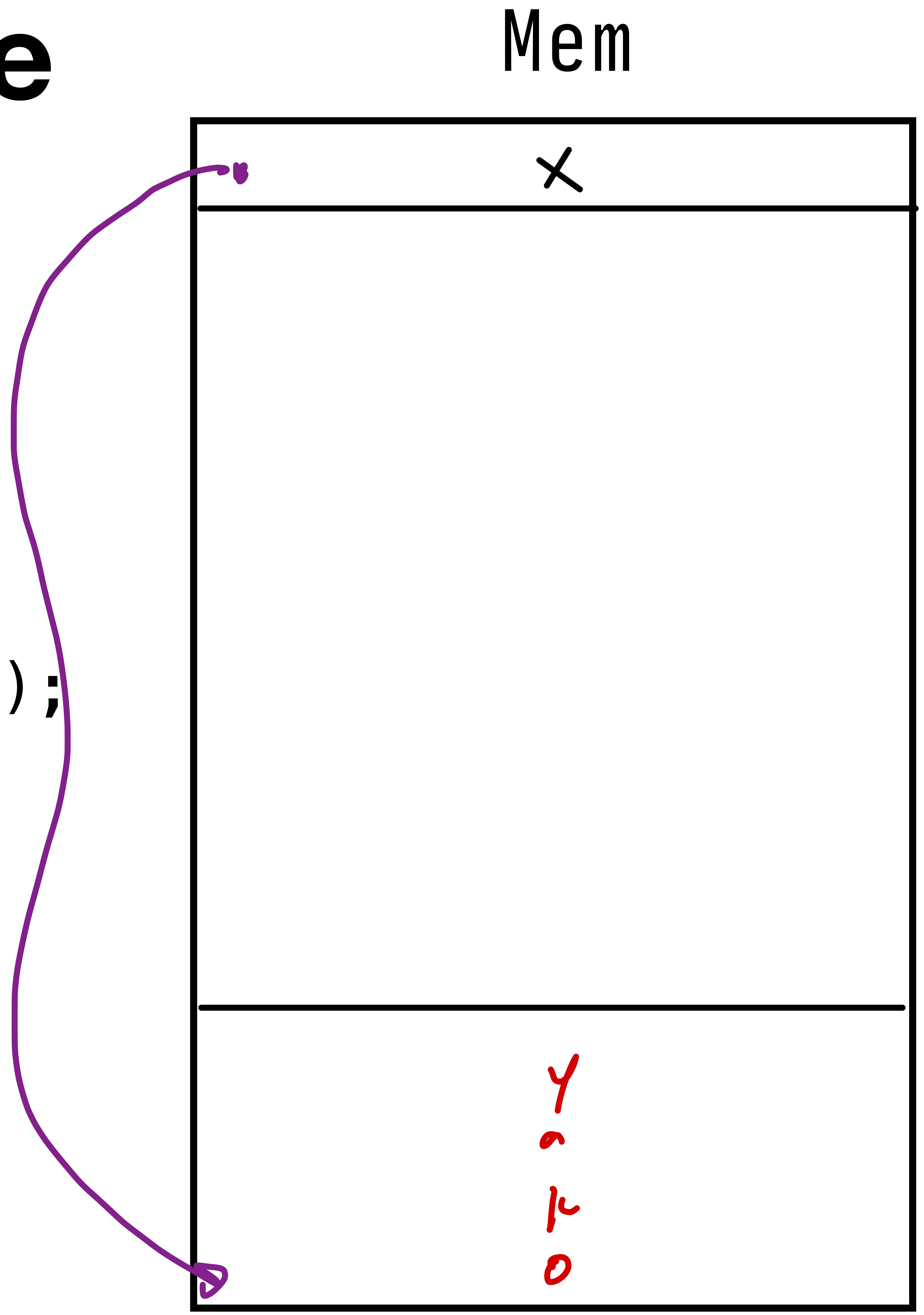
Growing Data Example

```
fn indirection(n: i32, s: &mut String) {
    let _y = 2;
    for _ in 0..n {
        *s += "okay";
    }
}

fn main() {
    let mut x : String = String::default();
    indirection(10, &mut x);
    println!("{x}");
}
```



Disappearing Data Example

```
fn fill(s : &mut String){
    let filler = "okay";
    *s = String::from(filler);
}

fn main() {
    let mut x : String = String::default();
    fill(&mut x);
}
```



Memory Bugs

Once we are *referring* to data on the heap, we're also able to create more errors:

- **Dangling pointers**, references to invalid data
- **Memory Leaks**, losing references to valid data
- **Data races**, changing the same data with multiple processes

Memory Management

Four Kinds of Memory Managment

1. Explicit allocation/deallocation (C)
2. Ownership (Rust)
3. Automatic Reference Counting (Swift)
4. Garbage Collection (Python, Java, OCaml, ...)

Explicit Allocation

The approach of "traditional" systems languages like C: *the programmer is in charge of managing allocation/deallocation*

malloc allocates data on the heap and **free** deallocates it so it can be used again.

Benefits: It's simple and general

Downsides: It's highly prone to error

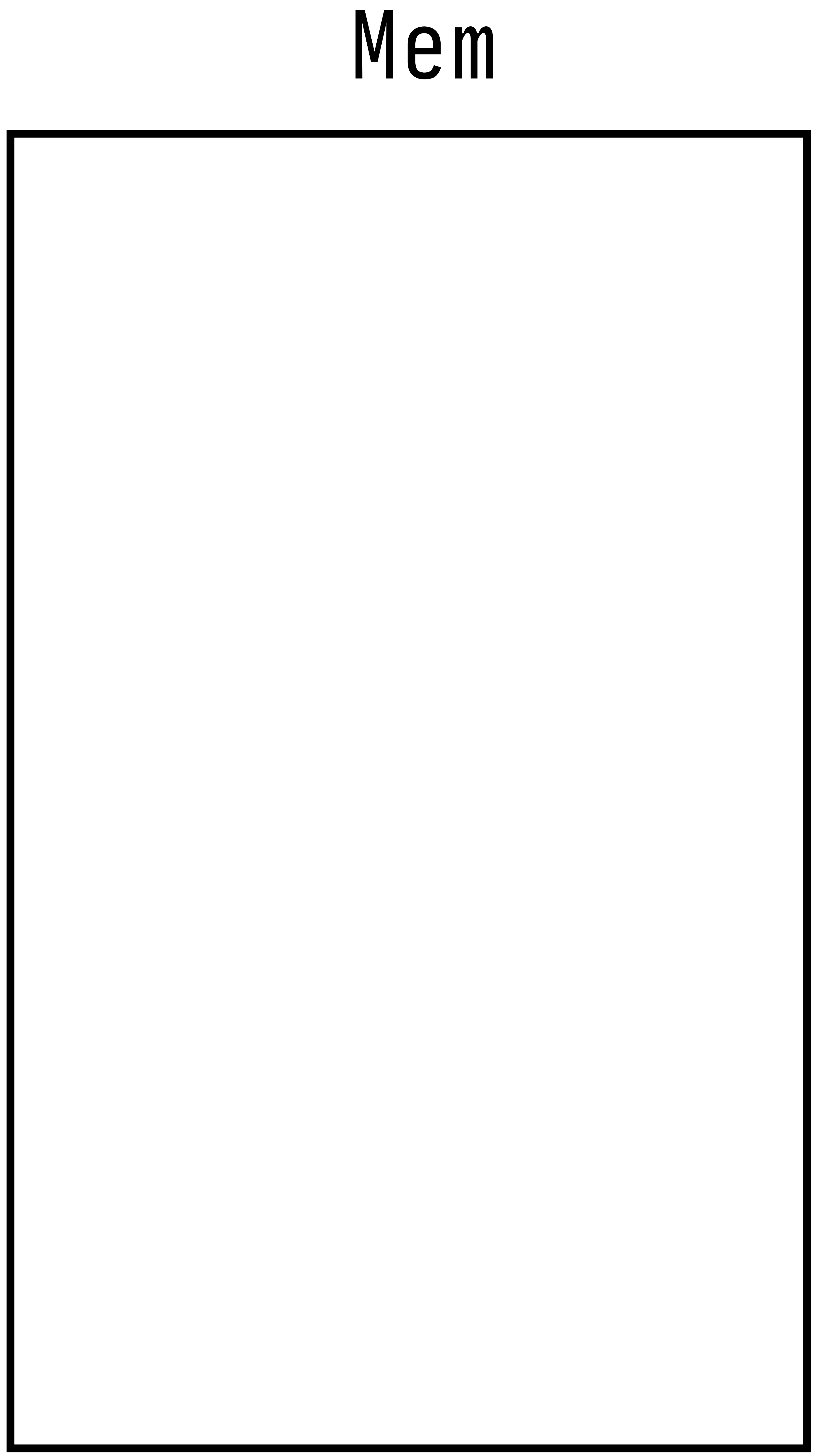
Dangling Pointer (C)

Mem

```
int main(void) {
   int *x = (int*)malloc(sizeof(int));
   *x = 2;
   free(x);
   printf("%d\n", *x);
   return 0;
}
```

Memory Leak

```
void leak(void) {
  int *x = (int*)malloc(sizeof(int));
  *x = 2;
  printf("%d\n", *x);
}

int main(void) {
  leak();
  return 0;
}
```



Garbage Collection

The approach of modern high-level languages: *periodically check the stack for what heap data is still valid and then clean up the heap*

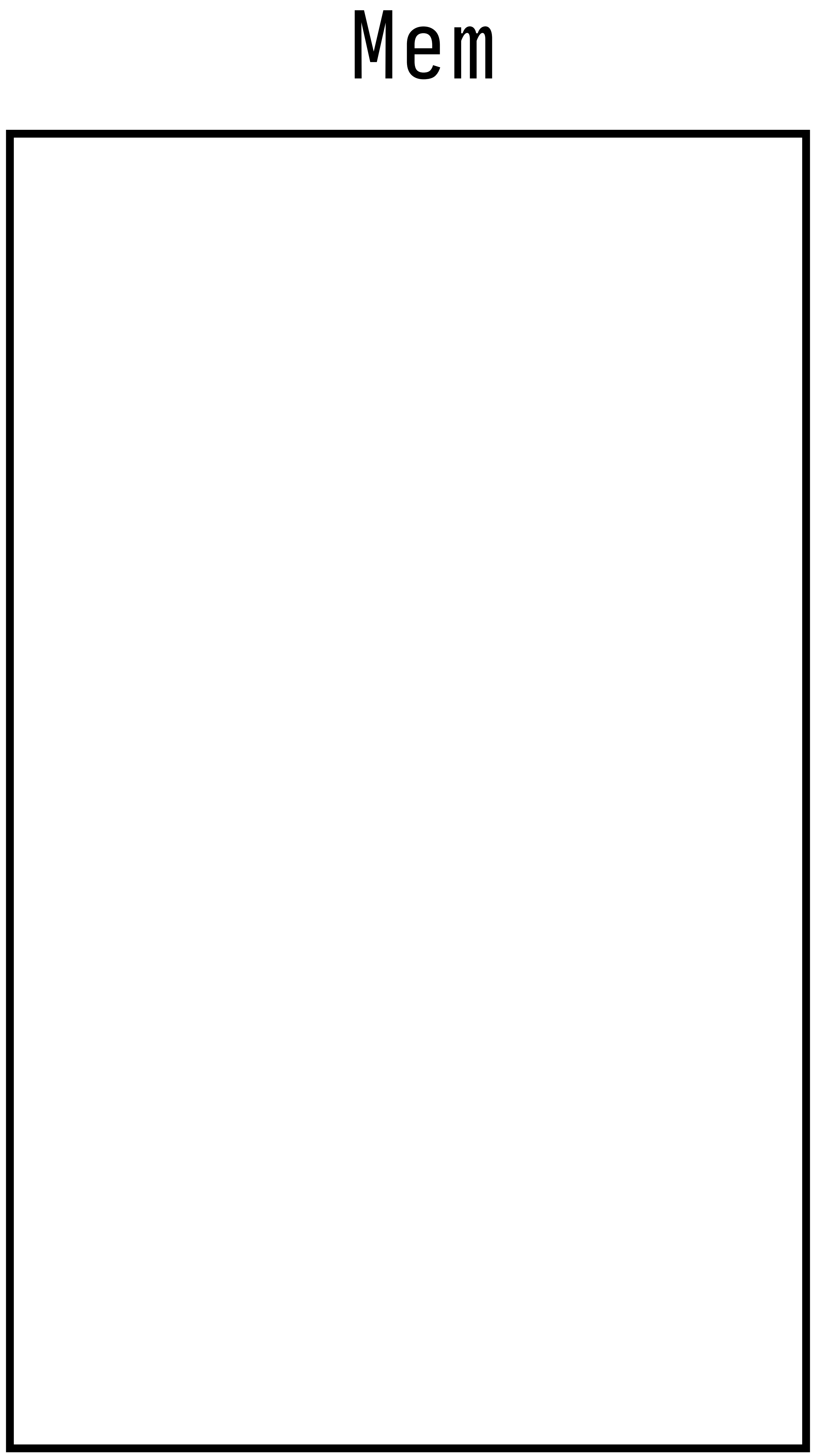
Benefits: Easy on the programmer, works fine in most cases

Downsides: Very little programmer control, difficult to performance optimize

Rough Sketch

Step 1: DFS from stack and mark

Step 2: Sweep the heap and clear unmarked data



Automatic Reference Counting

The approach taken by Swift (and C++ via smart pointers): *Count the number of references to a piece of heap data, free when it's down to zero*

Benefits: Easy on the programmer like GC

Downsides: Reference cycles, overhead (?), still not that much control

Rough Sketch

```
class Stuff {
    init() {
        print("allocating")
    }
    deinit {
        print("deallocating")
    }
}

var r1 : Stuff? = Stuff()
var r2 : Stuff? = r1
var r3 : Stuff? = r2

r1 = nil
r2 = nil
r3 = nil
```

Mem

Ownership

The approach taken by Rust: *follow these two rules*

1. *Every value has one owner at any given time*
2. *When the owner of a value goes out of scope, any memory associated with the value is freed*

Benefits: User-control without requiring explicit allocation

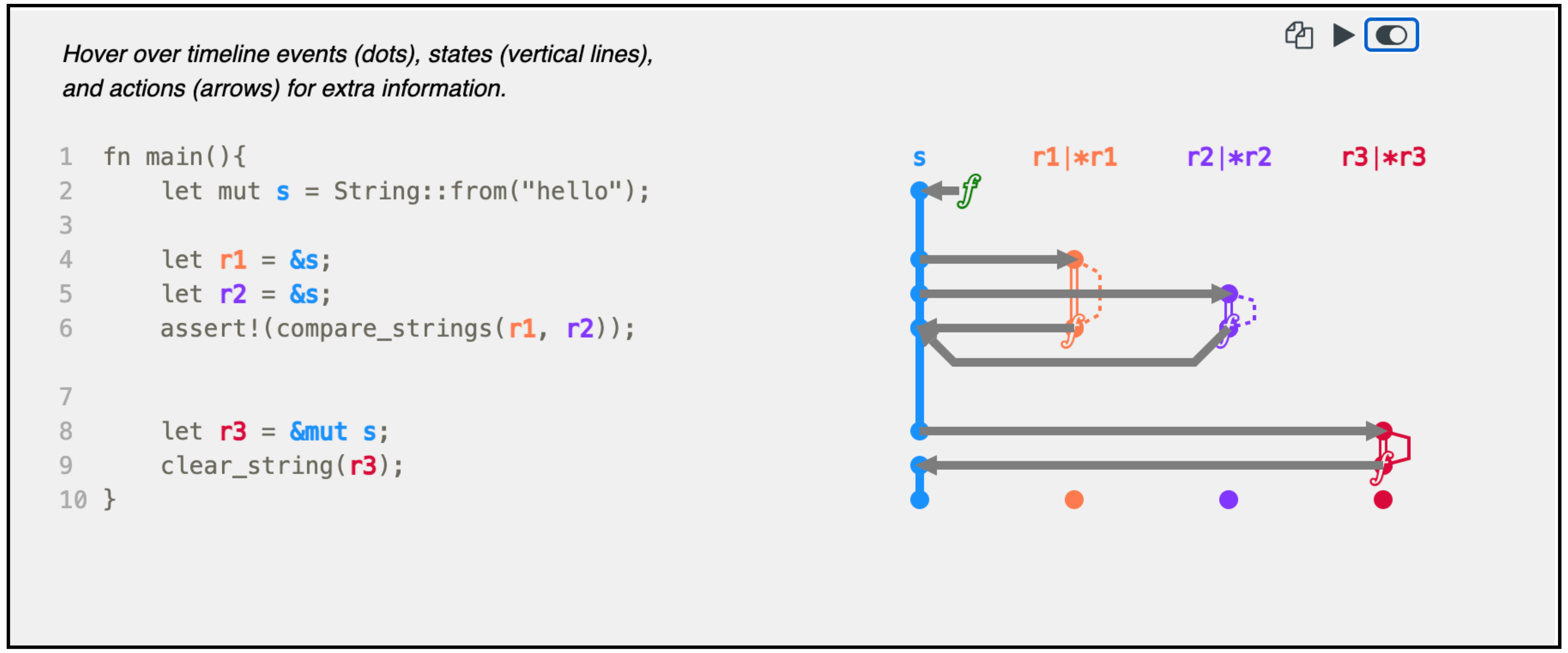
Downsides: Unintuitive at first

The Big Question

If we're not explicitly allocating/deallocating memory, when should it happen?

Rust's answer: as soon as a variable/parameter referring to it goes out of scope.

The Point



Hover over timeline events (dots), states (vertical lines), and actions (arrows) for extra information.

```
fn main(){
    let mut s = String::from("hello");

    let r1 = &s;
    let r2 = &s;
    assert!(compare_strings(r1, r2));


    let r3 = &mut s;
    clear_string(r3);
}
```

Ownership allows this stupid-simple deallocation pattern

If only one variable owns the data, then if they go out of scope, **no one owns the data**

https://github.com/rustviz/rustviz/blob/master/src/svg_generator/example.png

But this stupid-simple, cheap approach means that we can't do many "intuitive" things

No References to the Same Data

```rust
fn main() {
    let x = String::from("hello world");
    let y = x;
    println!("{x}");
    println!("{y}");
}
```

It's not possible to have two references to the same piece of data

(this doesn't seem like a problem here)

A Note on the Philosophy of Rust

```c
int main(void) {
  char* x = "hello world";
  char* y = x;
  printf("%s\n", x);
  printf("%s\n", y);
  return 0;
}
```

The type/borrow checker disallows a lot of "natural" programs

Working with your hand tied behind your back makes you better with that one hand

Workshop: Finish Assignment 1

Workshop

A couple options today:

- Finish assignment 1
  - Look at crate slow_primes and see if you can speed up your nth_prime function
- Continue reading about borrowing
- Install rustviz