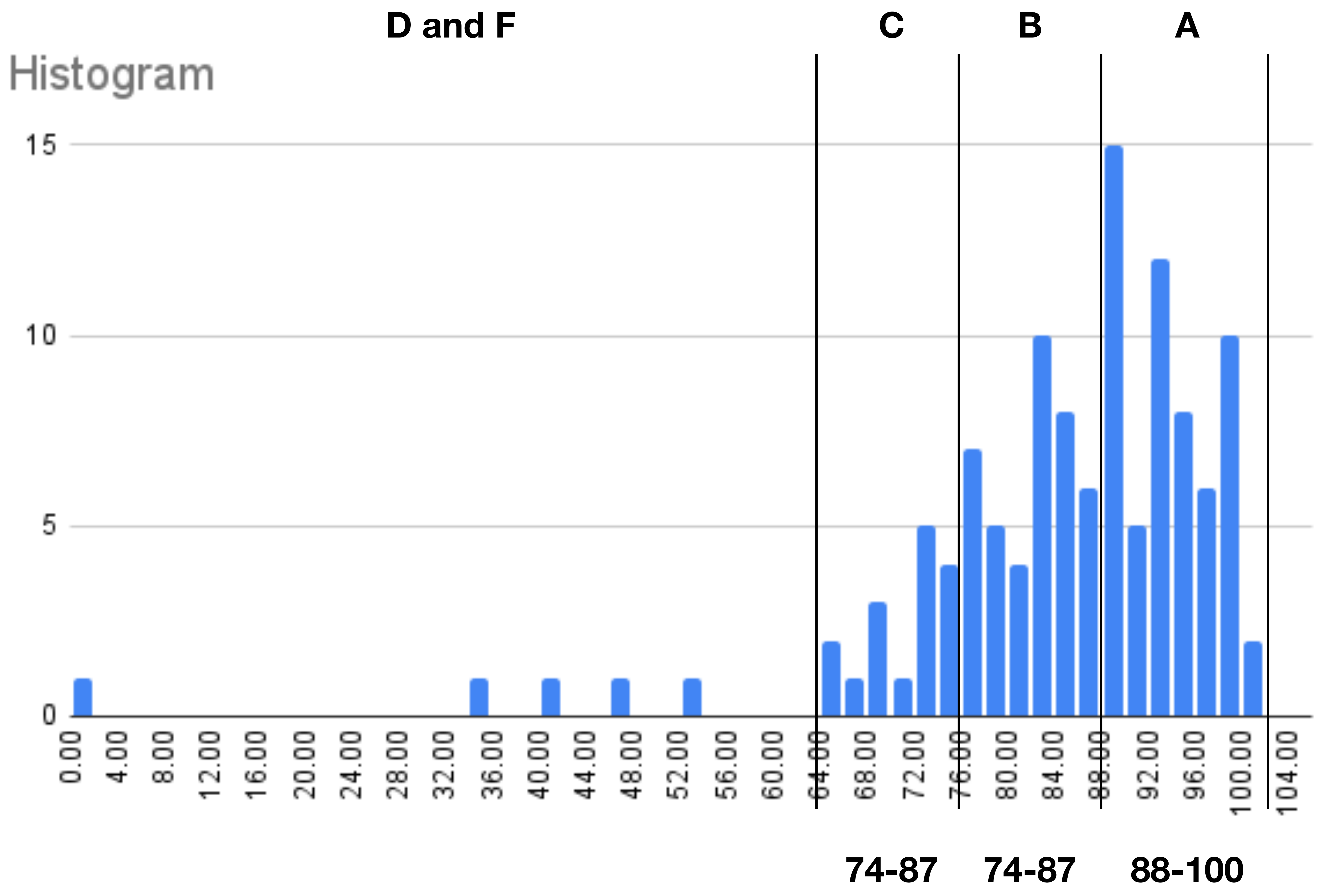
CMPT 383

Lecture 14: Midterm and Ownership Review

Anders Miltner

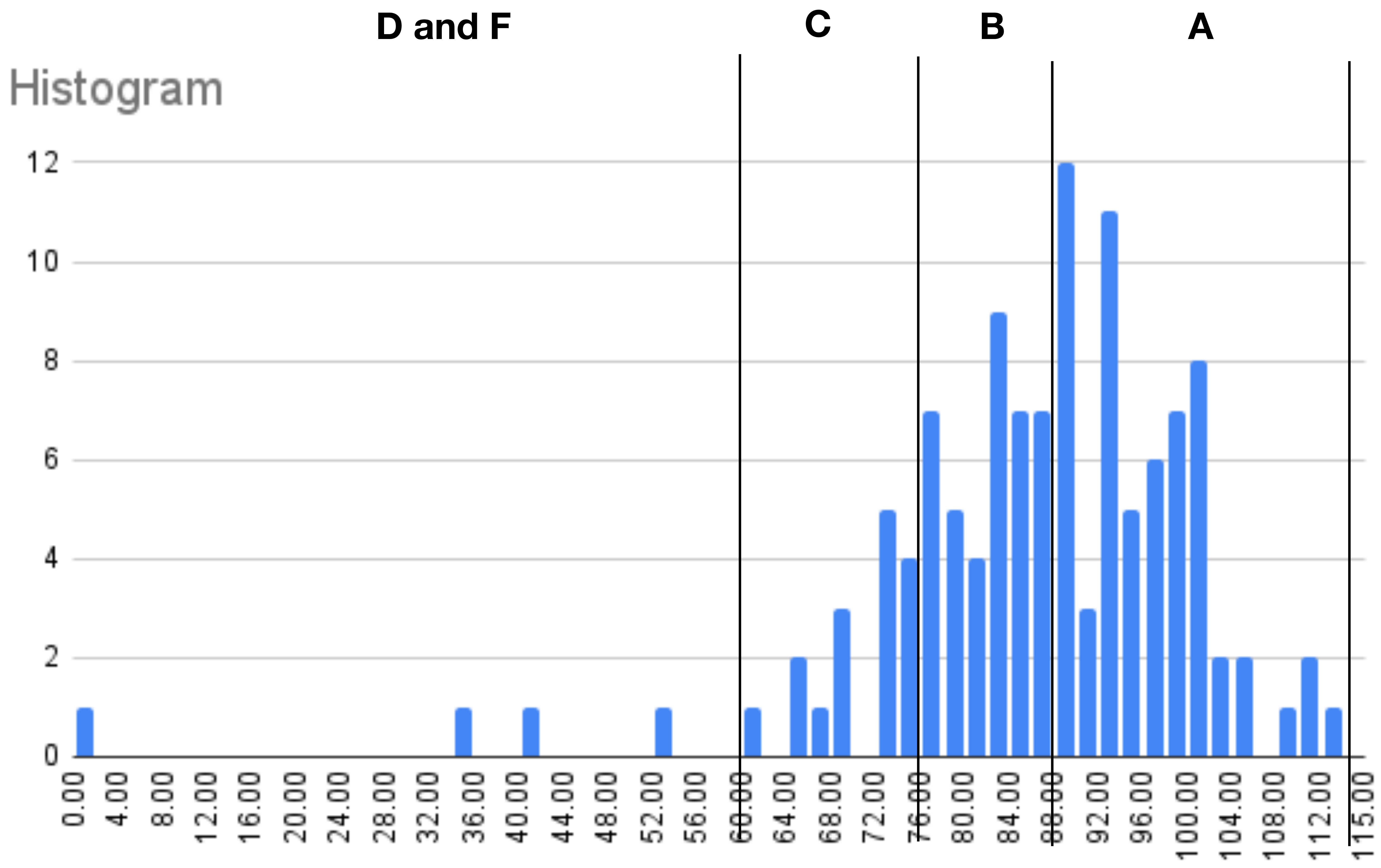
Midterm Scores

On canvas



Midterm Scores

On canvas



Fun Stats!

- Pre EC
  - Maximum 100% (two people received)
  - Quartiles: 100-93,93-87,87-79
- With EC
  - Maximum 112%
  - Quartiles: 100-94.5,94.5-88,88-79

Go Through Solutions!

Rest of Today

- Go through more and more cases of borrowing
- Start talking about lifetimes

Purely Functional Code?

```
fn cons(x:i32,mut vec: Vec<i32>) -> Vec<i32> {
    vec.push(x);
    vec
}
```

```
fn main() {
    let x = vec![];
    let y = cons(1,x);
    let z = cons(1,cons(2,cons(y)));
}
```

```
fn main() {
    let x = vec![];
    let y = cons(1,x);
    let z = cons(2,x);              ✗
}
```

Cannot transfer ownership twice

Purely Functional Code?

```
fn cons(x:i32,mut vec: Vec<i32>) -> Vec<i32> {
    vec.push(x);
    vec
}
```

```
fn main() {
    let x = vec![];
    let y = cons(1,x);
    y.push(5);
}
```

✗

Cannot edit immutable data

```
fn cons(x:i32,vec: Vec<i32>) -> Vec<i32> {
    vec.push(x);
    vec
}
```

✗

Cannot edit immutable data

Purely Functional Code?

```
fn cons(x:i32,mut vec: Vec<i32>) -> Vec<i32> {
    vec.push(x);
    vec
}
```

```
fn main() {
    let x = vec![];
    let y = cons(1,x.clone());
    let z = cons(2,x.clone());
    println!(“{}”,y);
    println!(“{}”,z);
}
```

Purely Functional Code?

```
fn cons(x:i32,mut vec: Vec<i32>) -> Vec<i32> {
    vec.push(x);
    vec
}
```

```
fn nil() -> Vec<i32> {
    vec![]
}
```

```
fn nil() -> Vec<i32> {
    vec![]
}
```

What happens if I don’t make “nil” a function, like we do in Haskell

Mutable Code?

```
fn push2(x:i32,vec: & mut Vec<i32>) {
    vec.push(x);
    vec.push(x);
}
```

```
fn main() -> Vec<i32> {
    let mut vec0 = vec![];
    let vec0ref0 = & mut vec0;
    let vec0ref1 = & mut vec0;
    push2(12,vec0ref0);
}
```

✗

Cannot mutable borrow while another mutable borrow is active

Mutable Code?

```
fn push2(x:i32,vec: & mut Vec<i32>) {
    vec.push(x);
    vec.push(x);
}
```

```
fn main() -> Vec<i32> {
    let mut vec0 = vec![];
    let vec0ref0 = & mut vec0;
    let vec0ref1 = & vec0;
    push2(5,vec0ref0);
}
```

✗

Cannot REGULAR borrow while another mutable borrow is active

Mutable Code?

```
fn push2(x:i32,vec: & mut Vec<i32>) {
    vec.push(x);
    vec.push(x);
}
```

```
fn main() -> Vec<i32> {
    let mut vec0 = vec![];
    let vec0ref0 = & vec0;
    let vec0ref1 = & vec0;
    println(“{:?}”,vec0ref0);
}
```

Go absolutely wild with regular borrows if there’s no mutable borrows going on

Mutable Code?

```
fn push2(x:i32,vec: & mut Vec<i32>) {
    vec.push(x);
    vec.push(x);
}
```

```
fn main() -> Vec<i32> {
    let mut vec0 = vec![];
    let vec0ref0 = & vec0;
    push2(8,& mut vec0);
    println!(“{:?}”,vec0ref0);
}
```

✗

Cannot mutable borrow while another regular borrow is active

Slices

- You don’t just have to borrow all the memory, you can also borrow bits of the memory
- Not available on all types

String Slices

- Only slice we need to worry about for now
  - We’ll also get into array slices later

```
fn printFirstTwoChars(x:&String) {
    println!(“{}”,&x[0..2]);
}
```

String Slice Type: &str

```
fn find_length(x:&str) {
    return x.len();
}
```

```
fn main(x:&str) {
    let str = “Hello World!”;
    asserteq!(find_len(str[0..6]),6);
}
```

We’ve provided borrowed data as inputs

What about outputting borrowed data?

Returning Borrowed Data

```
fn find_longest(x:&Vec<Vec<u32>>) -> &Vec<u32> {
    let mut longest = &x[0];
    for v in x {
        if v.len() > longest.len() {
            longest = v;
        }
    }
    return &longest;
}
```

What about this?

```rust
fn find_longer(x:&Vec<u32>,y:&Vec<u32>) -> &Vec<u32> {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```

✗

We don’t know how to correctly ensure the borrowed data exists the appropriate amount of time

Lifetimes!

- You can describe the lifetime of a certain borrowed variable
- You can ensure that the lifetimes are agreed upon

```
fn find_longer<'a>(x:&'a Vec<u32>,y:&'a Vec<u32>) -> &’a Vec<u32> {
    if x.len() > y.len() {
        return x;
    } else {
        return y;
    }
}
```