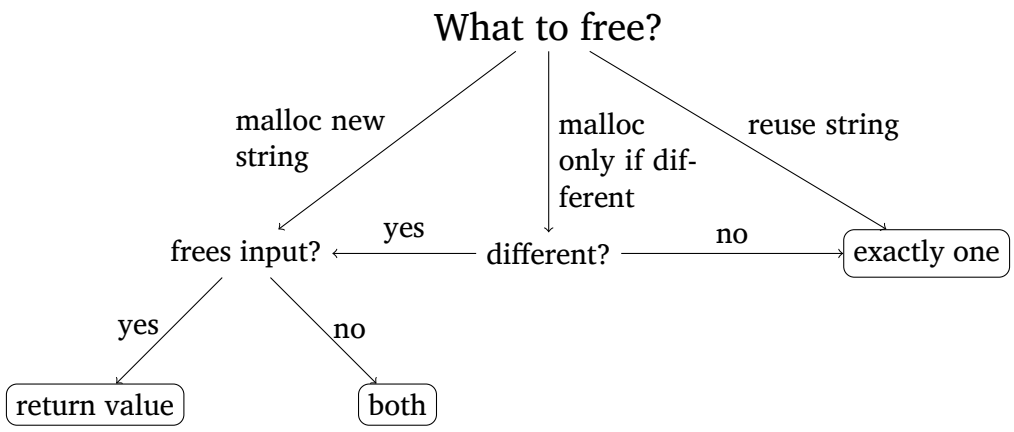
Lecture 2

Ownership model

Double free and use after free

Real issues in systems programming we will be solving.
C example:

```c
// make a string lowercase
char *to_lowercase(char *string);
```



Ownership rules

1. Every value has an owner variable/struct owns its data
2. That owner is unique
3. When the owner goes out of scope, the value is dropped

```rust
{
    // string owns the string allocated on the heap
    let string: String = "Hello".to_string()
} // string freed automatically
```

Moving

transfer ownership, old value invalid

```
fn say_string(string: String) { // takes ownership of `string`
    println!("{}", string);
} // dropped here

let string: String = "Hello".to_string();
say_string(string); // ownership transferred from `string`
say_string(string); // can't use `string` anymore
```

One solution:

```
fn say_string(string: String) {
    println!("{}", string);
}

let string = "Hello".to_string();
say_string(string.clone());
say_string(string);
```

Ugh, this makes an extra allocation, but sometimes necessary
Rust equivalent to C code from before:

```
fn to_lowercase(string: String) -> String;
```

What happens here?

- Double free No double free since owner unique
- Use after free No use after free, since once ownership is transferred you can't access the value

Borrowing

- References ≈ pointers
- Example We don't want `say_string` to take ownership of `string`

```
fn say_string(string: &String) {
    println!("{}", string);
}

let string = "Hello".to_string();
say_string(&string);
say_string(&string);
```

- Mutable borrowing

```
struct Counter {
    count: u32,
}

impl Counter {
    fn get_count(&self) -> u32 {
        self.count
    }

    fn increment(&mut self) {
        self.count += 1;
    }
}
```

- Slices + XOR aliasing

```
let mut vector: Vec<i32> = vec![1, 2, 3, 4];
let two: &i32 = &vector[1];
vector.clear(); // uh oh!
```