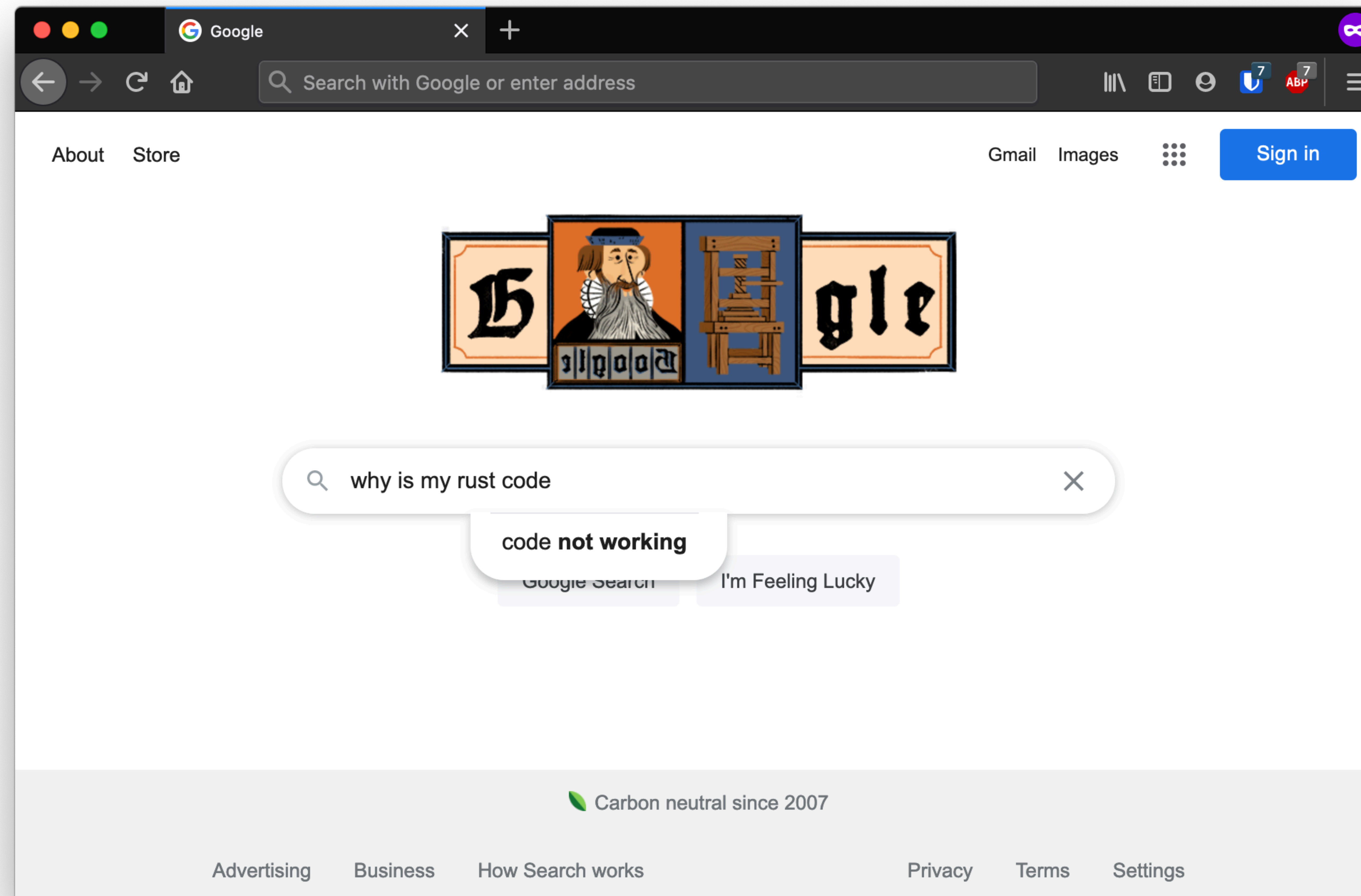
Custom Types

CS110L
January 24th, 2022

Today & [probably part of] Wednesday



Let's implement a linked list together!

Follow along

- Rust Playground: https://play.rust-lang.org
- Create your own Rust package locally:
  - Run in shell: **`cargo new --bin linked-list-example`**
    - *linked-list-example = name of directory to create (whatever you want!)*
    - *More on creating new cargo packages here: https://doc.rust-lang.org/cargo/commands/cargo-new.html*
  - You can now open the `linked-list-example` directory and write your code there! (in src/main.rs, as usual.)

Quick review: what’s a linked list?

Group discussion:

- How would you go about implementing a Linked List class in C or C++?
  - What structs would you need?
  - What kinds of methods would you provide?
  - What would your test code look like?
  - In terms of memory errors we've been talking about, what could go wrong?
- Based on what you know about Rust so far, what do you think will be challenging about implementing a linked list in Rust?

Quick review: what's a linked list?

```
C++ :

struct Node {
    int value;
    Node* next;
}
```

```
int main() {
   Node* first = (Node*)malloc(sizeof(Node));
   first->value = 1;
   Node* second = (Node*)malloc(sizeof(Node));
   second->value = 2;
   first->next = second;

   /* do stuff (e.g., print the list) */

   free(first);
   free(second);
}
```

Defining structs in Rust (general syntax)

```
struct Person {
   name: String,
   location: String,
}


fn main() {
   let thea = Person { name: “thea”.to_string(),
                       location: “Boulder Creek”.to_string() };
   println!(“{} lives in {}”, thea.name, thea.location);
}
```

Defining a Node in Rust…?

C++ :

```
struct Node {
   int value;
   Node* next;
}
```

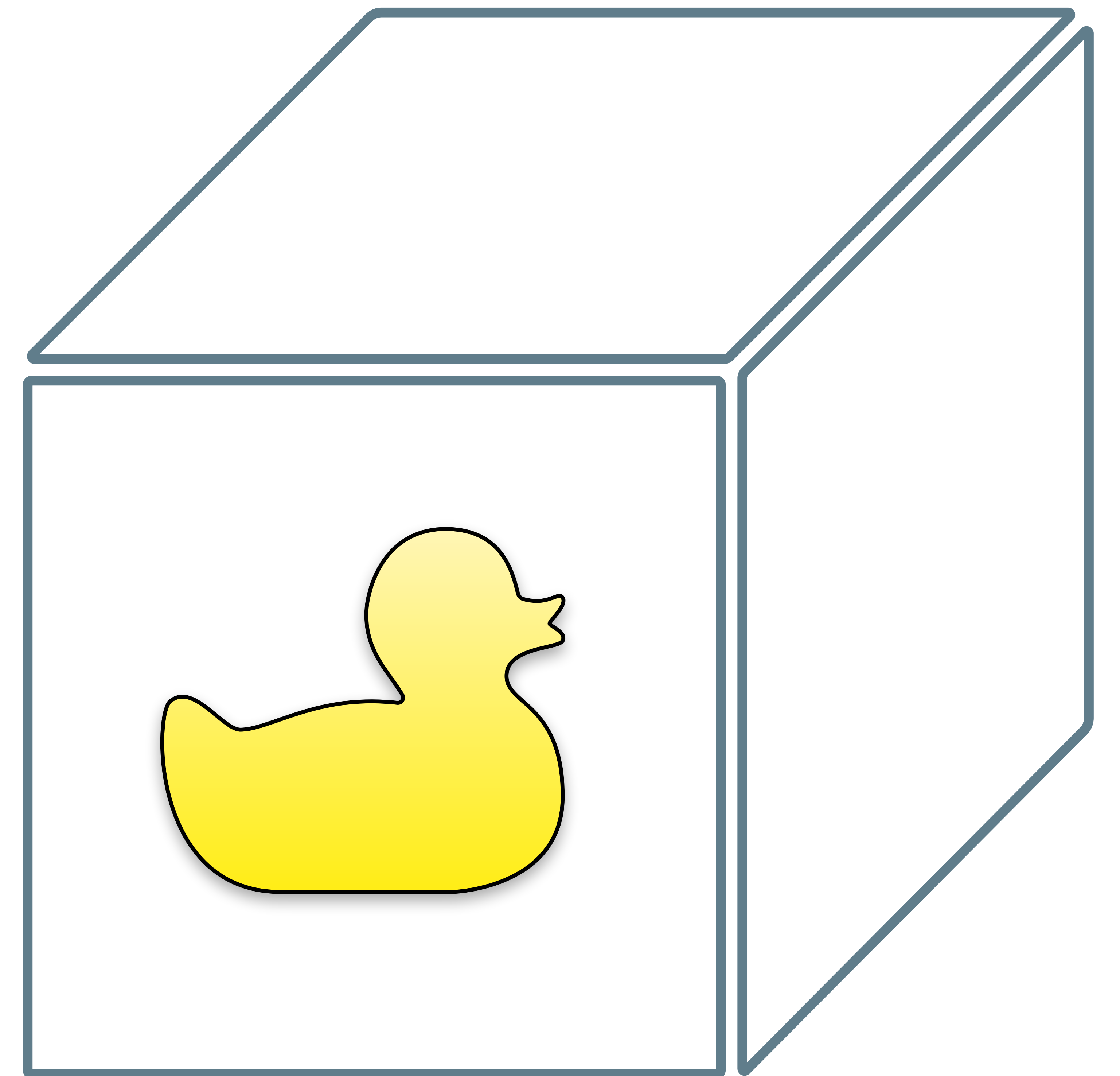
```
struct Node {
    value: i32,
    next: Node, /* won't work! recursive def. */
}
```

```
struct Node {
    value: i32,
    next: &Node, // not what we want! `&` does not create a pointer.
                 // — it implements "borrowing", which doesn't
                 // really apply here.
}
```

```
struct Node {
    value: i32,
    next: /* pointer to a node…? */
}
```

Box in Rust

- Create a Box
- Box goes on the heap
- Anything can go in the box
- Box *owns* whatever is in the box. When box goes out of scope -> value in box destroyed.
- *Same thing as unique_ptr in C++:*
    - *"A smart pointer that owns and managers another object through a pointer and disposes of that object when the unique_ptr goes out of scope."*



https://doc.rust-lang.org/rust-by-example/std/box.html

Box in Rust

```
struct Node {
    value: i32,
}


fn main() {
    let node = Box::new(Node {value: 1});
    println!(“{}”, node.value);
}
```

- Variable `node` owns Box<Node>
- When `node` is no longer in use, Box is (automatically) destroyed
  - *Compiler inserts call to Box’s `drop` function*
- When Box is destroyed, Node object is destroyed

Defining a Node in Rust: what do we need?

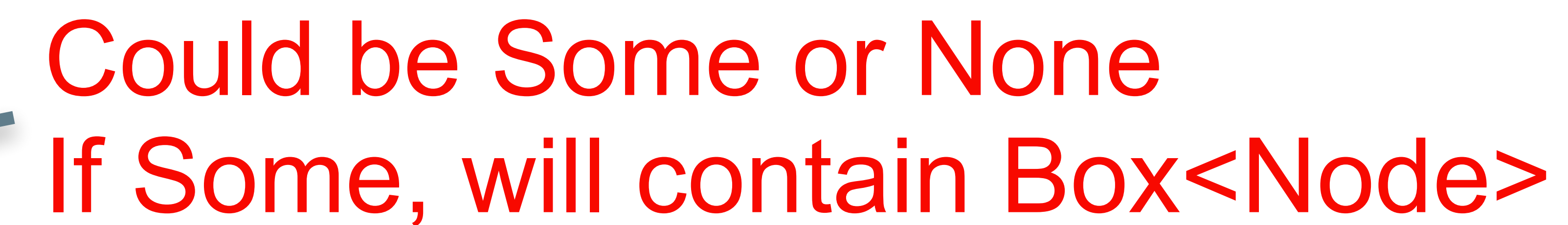
```
struct Node {
    value: i32,
    next: Box<Node>,
}
```

Using a Node: one-element linked list

```
struct Node {
   value: i32,
   next: Box<Node>,
}


fn main() {
   let node = Box::new( Node {
                           value: 1,
                           next: /* equiv. of nullptr…?*/,
                        });
}
```

Throwback to Options

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}
```




```
fn main() {
    let node = Box::new( Node {
                          value: 1,
                          next: None
                       });
}
```

This has made its way back into C++ — see [std::optional]

Let's make a longer list ~~~ take 1

does not compile

```
struct Node {
   value: i32,
   next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let second = Box::new( Node { value: 2, next: None });
    first.next = second;
}
```

Reminder: we want to change `first`, so explicitly make it mutable

Let’s make a longer list ~~~ take 1

This SHOULD be an Option…

but you’re giving me a Box?????

```
|      first.next = second;
|                   ^^^^^^
|                   |
|                   expected enum `std::option::Option`, found struct `std::boxed::Box`
|                   help: try using a variant of the expected enum: `Some(second)`
|
= note: expected enum `std::option::Option<std::boxed::Box<_>>`
           found struct `std::boxed::Box<_>`

or[E0308]: mismatched types
```

Let’s make a longer list ~~~ take 2

Compiles!

```
struct Node {
   value: i32,
   next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let second = Box::new( Node { value: 2, next: None });
    first.next = Some(second);    // This is now Option<Box<Node>>
}
```

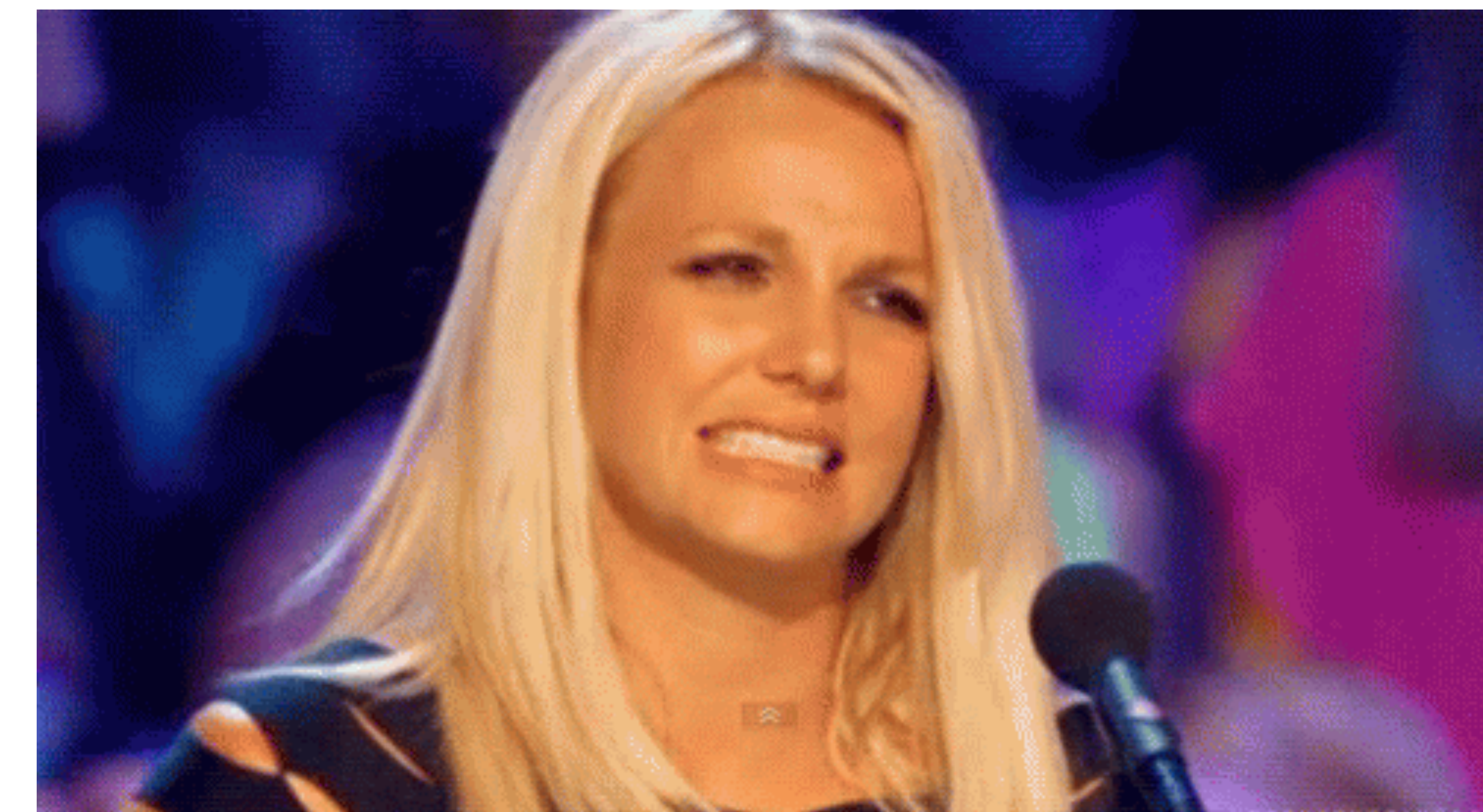
Let’s make an even longer list ~~ take 1

does not compile

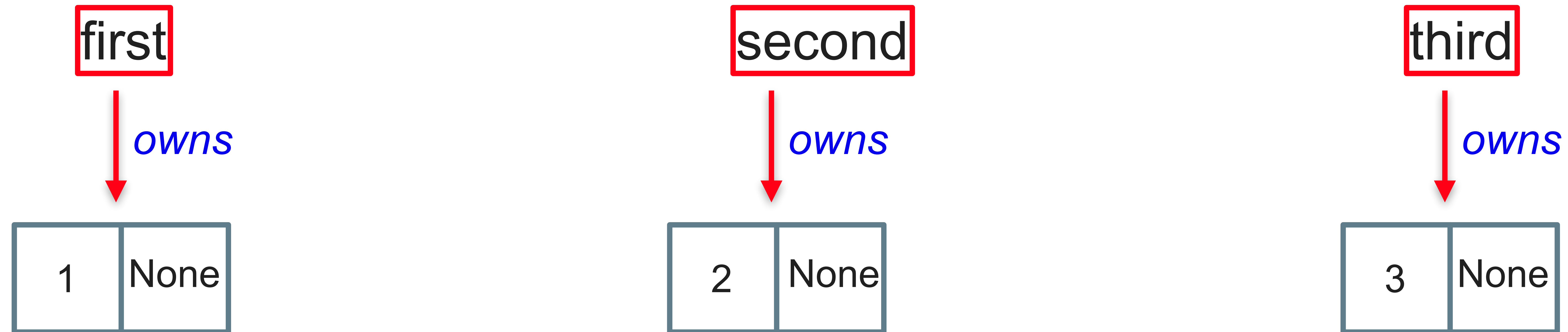
```
struct Node {
   value: i32,
   next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

Let’s make an even longer list ~~ take 1

```
error[E0382]: assign to part of moved value: `*second`
  --> src/main.rs:12:5
   |
8  |     let mut second = Box::new(Node {value: 2, next: None});
   |         ---------- move occurs because `second` has type `std::boxed::Box<Node>`, whic
h does not implement the `Copy` trait
...
11 |     first.next = Some(second);
   |                       ------ value moved here
12 |     second.next = Some(third);
   |     ^^^^^^^^^^^ value partially assigned here after move
```

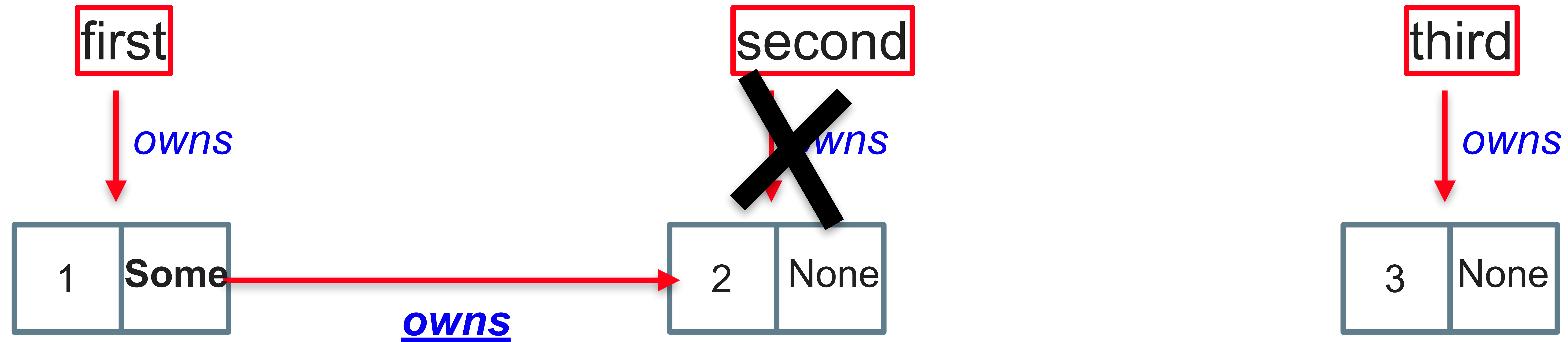


What's going on here???



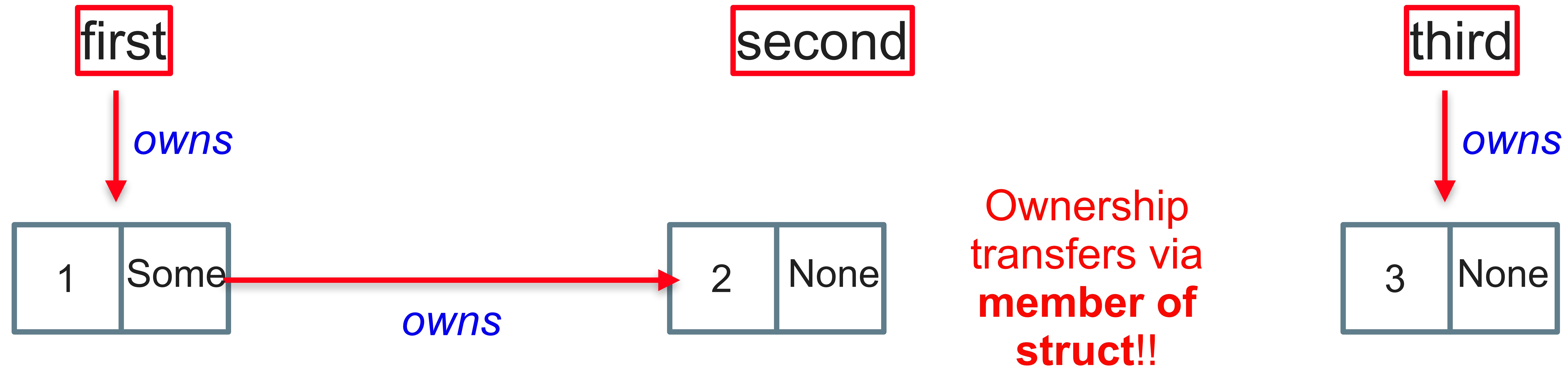

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

What’s going on here???



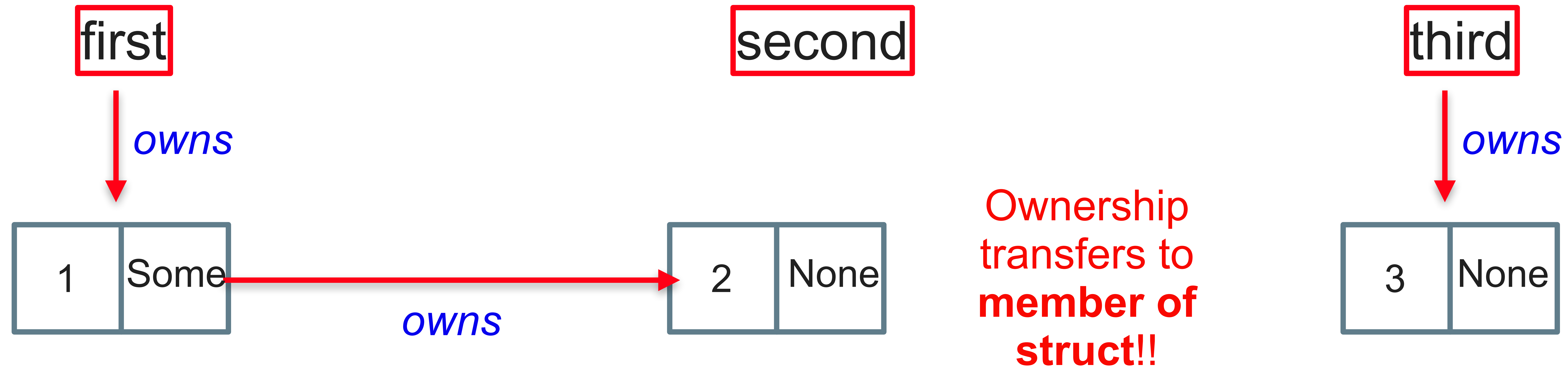

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

What’s going on here???




```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

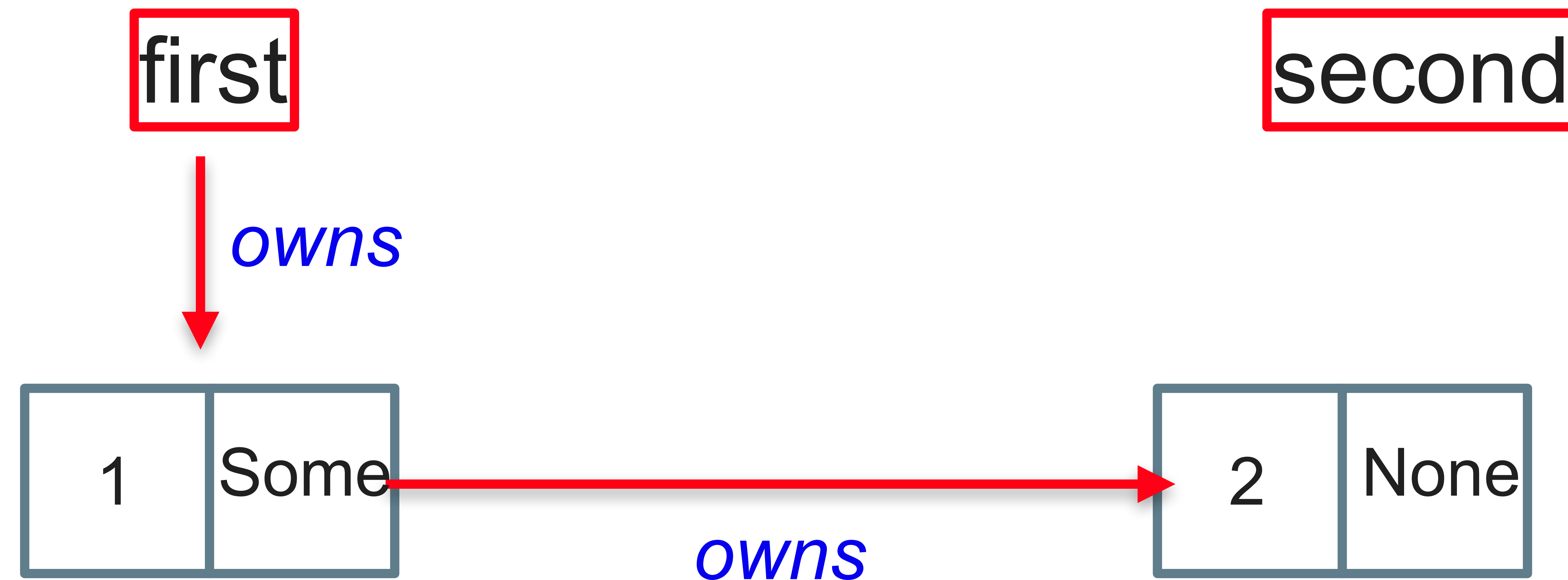
What’s going on here???




```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    first.next = Some(second);
    second.next = Some(third);
}
```

Error — can no longer access Box<Node> via variable `second`

“Chain of ownership”



- Implication: when `first` is dropped (destroyed):
  - First node of list is dropped,
  - …so Option (in Node struct) is dropped,
  - …so Box (in Option) is dropped,
  - …so second Node (in Box) is dropped.
- Everything is cleaned up :)

- …But we can’t use `second` anymore to access this node.
- *These are the type of issues that can get really annoying in Rust :(*

Let’s make an even longer list ~~ take 2

compiles

```
struct Node {
   value: i32,
   next: Option<Box<Node>>,
}


fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third); // swap order of these lines
    first.next = Some(second); // use `second` to access node
                               // before it loses ownership.
}
```

Let’s print the list!

C++ :

```
struct Node {
    int value;
    Node* next;
}
```

C++:

```
int main() {
    Node* first = (Node*)malloc(sizeof(Node));
    first->value = 1;
    Node* second = (Node*)malloc(sizeof(Node));
    second->value = 2;
    first->next = second;

    Node *curr = first;
    while (curr != NULL) {
        printf("%d\n", curr->value);
        curr = curr->next;
    }

    free(first);
    free(second);
}
```

How do we turn this into Rust?

- There are no “pointers” in Rust; what type should `curr` be?
- We probably want `curr` to refer to the `first` node to start with, but we don’t want `first` to lose ownership of the node. (We don’t want the list to get freed once `curr` isn’t used anymore!)
- What’s the condition of our loop? (How do we know when we’ve reached the end?)

C++:

```
Node *curr = first;
while (curr != NULL) {
    printf("%d\n", curr->value);
    curr = curr->next;
}
```

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = /* something */;

}
```

make `curr` mutable, because we’re going to reassign it

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);

}
```

`curr` is an **Option<&Box<Node>>**

- **Option**: can be `Some` or None
  - Use `None` to indicate end of List
- **&Box<Node>:**
  - If Some: <&Box<Node>>
  - Want to take the Box by reference (why might this be important?)
  - Box “contains” heap-allocated Node

Let's print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr: Option<&Box<Node>> = Some(&first);

}
```

Reminder: you can explicitly write in the types of variables if you want to. Otherwise, Rust compiler infers for us.

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
      // print value
      // update curr
    }
}
```

`Option` has `is_some()` and `is_none()` methods.
We want to keep looping while `curr` has some value.
(Same logic as `while curr != NULL` in C++ example.)

Let’s print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!(“{}”, curr.value);
        // update curr
    }
}
```

does not compile

`curr` is an Option — `.value` isn’t valid.

Let's print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!("{}", curr.unwrap().value);
        // update curr
    }
}
```

compiles!

- if `curr` is Some, extract the value
- Otherwise, panic (crash the program)
- Here: pretty safe to assume `curr` is `Some`, since we just checked on the previous line.

Reminder/review: Option, enum, unwrap

```
println!("{}", curr.unwrap().value);
```

- `curr` is an Option
- An Option is an `enum` defined in the Rust standard lib
    - An `enum` is a type that can take on a specific, finite number of defined variants
    - In Rust, `enums` can store values.
- An Option can be `Some` or `None`
- If `Some`, it stores an object (here: &Box<Node>)
- `curr.unwrap()` means:
    - If `curr` is Some, return the thing inside of the Some
    - If `curr` is None, panic

Std Rust lib:

```
enum Option {
    Some(<T>),
    None,
}
```

Let's print the list!

```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!("{}", curr.unwrap().value);
        curr = curr.unwrap().next;
    }
}
```

does not compile

```
struct Node {
    value: i32,
    next: Option<Box<Node>>,
}
```

- curr.unwrap() gives us a Node
- Node.next gives us Option<Box<Node>>

Why is this not what we want?

Introducing `as_ref()`

- Converts **&**Option<T> into Option<**&**T>
- If provided Option is None, returns None
- E.g.:

```
let mut curr = Some(&first);
while curr.is_some() {
    println!("{}", curr.unwrap().value);
    curr = (&curr.unwrap().next).as_ref();
}
```

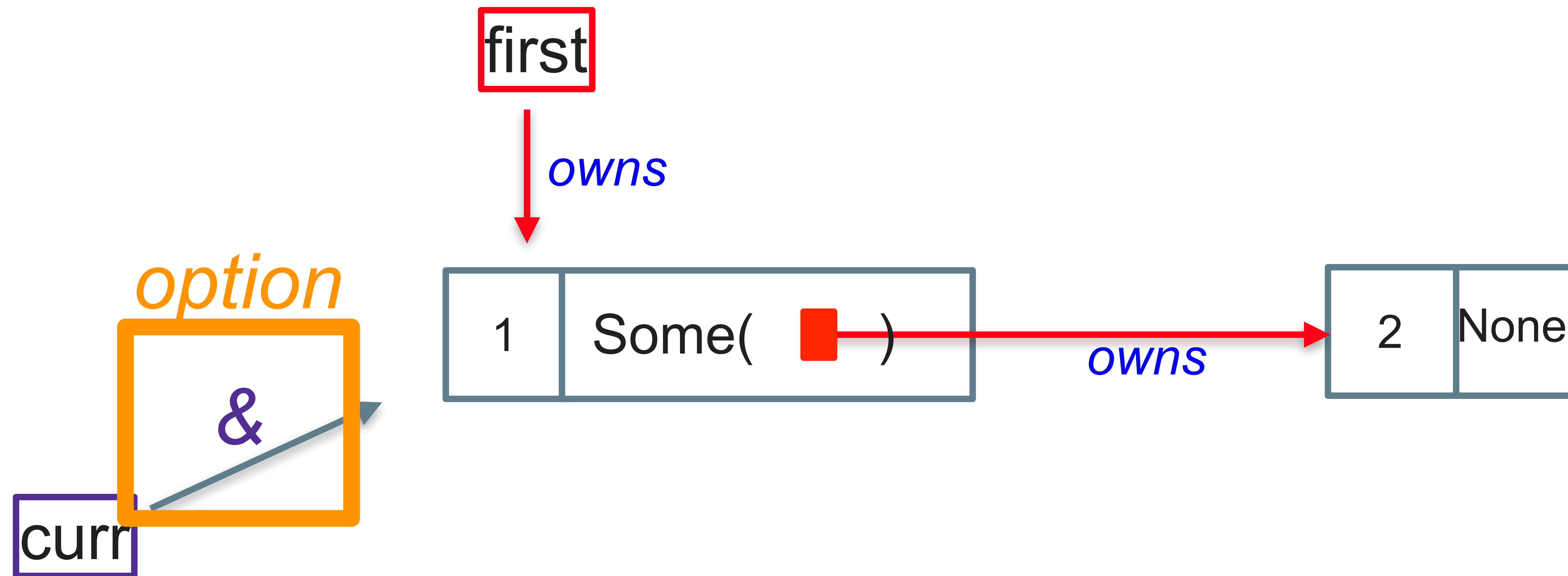
- curr.unwrap().next gives us Option<Box<Node>>
- Applying & gives us (**&**Option<Box<Node>>)
- Applying as_ref() gives us Option<**&**Box<Node>>
- If curr.unwrap().next is None, as_ref() returns None

Let’s print the list!

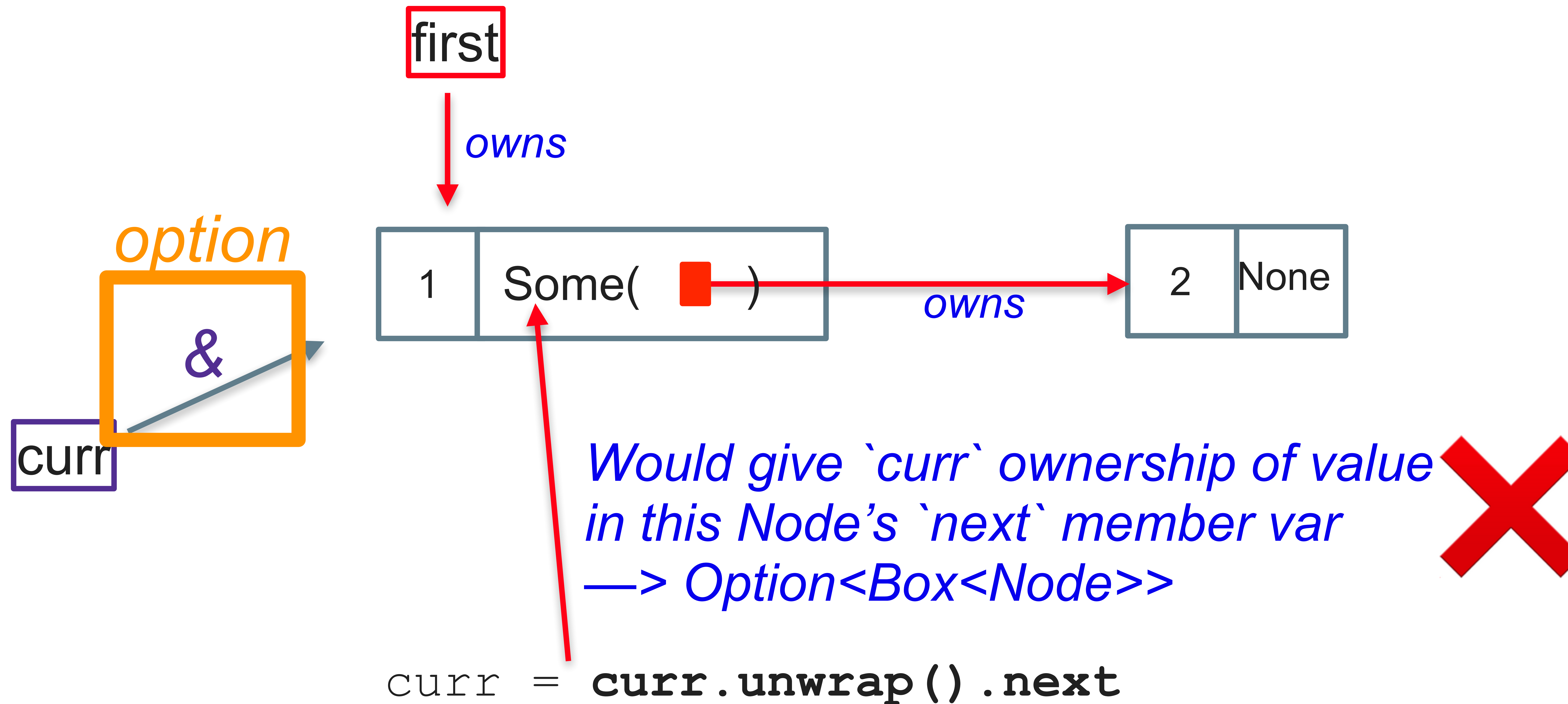
```
fn main() {
    let mut first = Box::new( Node { value: 1, next: None });
    let mut second = Box::new( Node { value: 2, next: None });
    let third = Box::new( Node { value: 3, next: None });
    second.next = Some(third);
    first.next = Some(second);

    let mut curr = Some(&first);
    while curr.is_some() {
        println!(“{}”, curr.unwrap().value);
        curr = (&curr.unwrap().next).as_ref();
    }
}
```

compiles

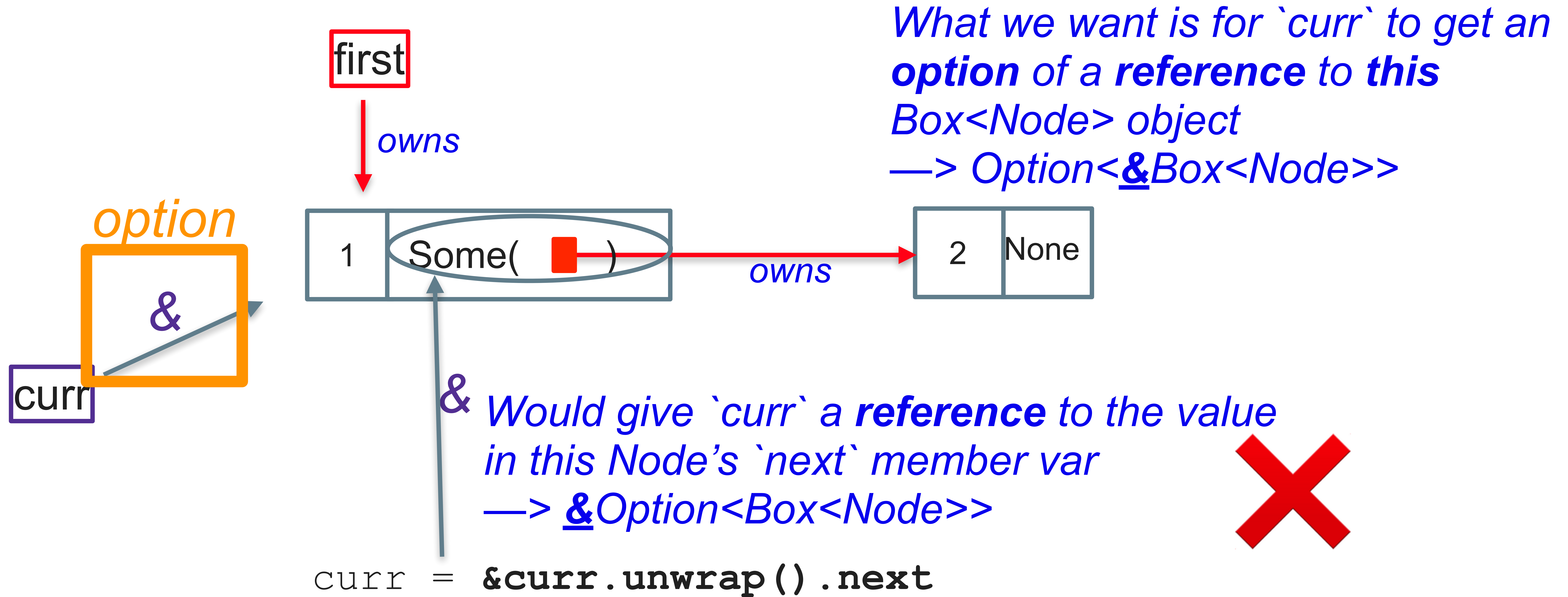
Changing `curr`, illustrated



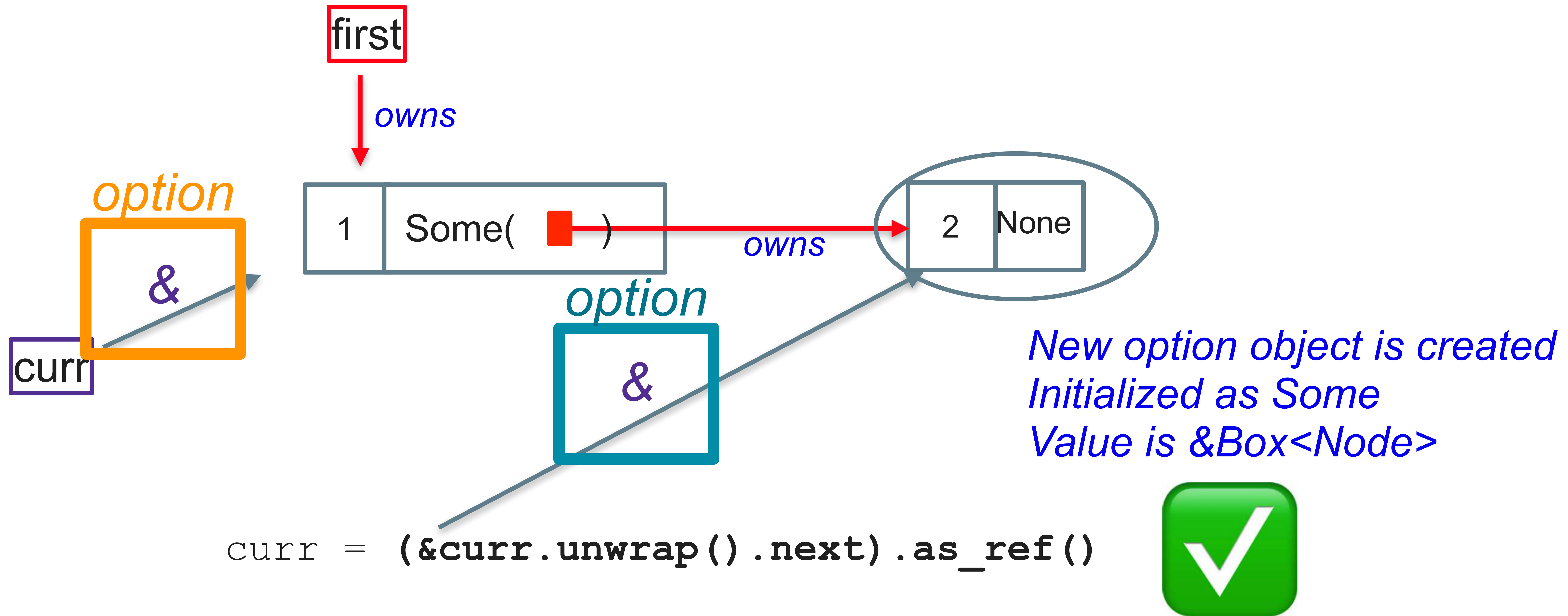
Changing `curr`, illustrated



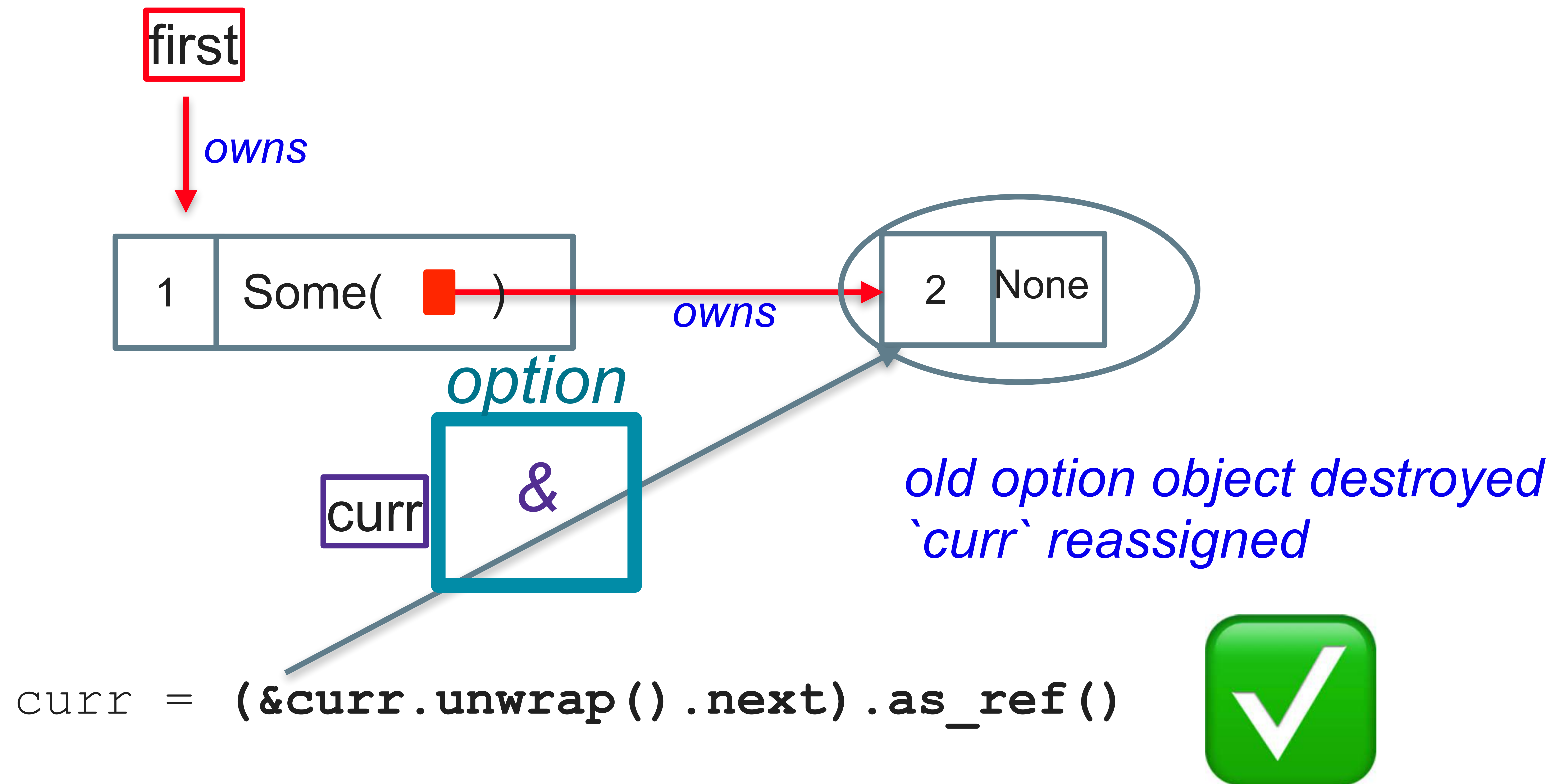
as_ref(): a [kinda bad] illustration



as_ref(): a [kinda bad] illustration



as_ref(): a [kinda bad] illustration




```
curr = (&curr.unwrap().next).as_ref()
```

Notes on writing Rust Code

- Read the compiler error messages. They're often helpful!
- `rustc --explain` for more info (sometimes helpful; sometimes overwhelming).
- Web search!! If you're having an issue, someone else has had it too.
- I highly recommend the [rust-analyzer]() plugin for your editor
  - *See week 2 exercises handout. If working on myth, see week 3 exercises for manual download instructions.*
  - In addition to basic warnings and errors, rust-analyzer will show you what compiler infers for variable types

```
let mut curr: Option<&Box<Node>> = Some(&first);
while curr.is_some() {
    println!("{}", curr.unwrap().value);
    /* temp variable for illustrating rust-analyzer */
    let tmp: Option<&Box<Node>> = (&curr.unwrap().next).as_ref();
    curr = tmp;
}
```

Recap: new material [also in notes]

- You can define your own types/structs using this syntax:

```
struct MyType {
    field1: i32,
    field2: String,
}
```

- You can initialize a struct using this syntax:

```
let my_object = MyType { field1: 1, field2: “Hello”.to_string() };
```

Recap: new material [also in notes]

- The `Box` type stores a pointer to heap-allocated memory.
- You can put anything inside of a `Box`
  - For example, a `Box<u32>` is a heap-allocated unsigned integer (probably not something that makes sense to do, but you can)
  - A `Box<Node>` is a heap-allocated `Node`
- `Box::new(...)` allocates memory and initializes it to `...`
- The `Box` **`drop`** ("destruction") function frees the heap memory
  - Remember: call to `drop` will automatically be inserted by compiler when variable that owns the `Box` is no longer in use

Recap: new material [also in notes]

- `Option::as_ref`
- Use if:
  - You have a reference to an Option with something inside (`&Option<T>`)
  - You want an Option containing a reference to that thing (`Option<&T>`)
- Given an `&Option<T>, as_ref` will:
  - Will "look" inside the Option that you have a reference to
  - If that Option is None, returns None
  - If that Option is Some, returns a new Option that is `Some(reference to contained object)`
- You can implement equivalent functionality using a `match` expression on Option types, but it's a handy one-line trick :)

So we have some nodes… can we make a [better] linked list?

- Goal: a `list` “class” with functions (push_front, pop_front, insert, etc.)

```
Example interface, in C++:

std::list<int> myList;
myList.push_front (200); // Create new Node with value 200; insert at head of list
myList.push_front (300); // Create new Node with value 300; insert at head of list
myList.pop_back ();      // Remove & destroy last element of list
// etc.
```

Adapted from https://www.cplusplus.com/reference/list/list/

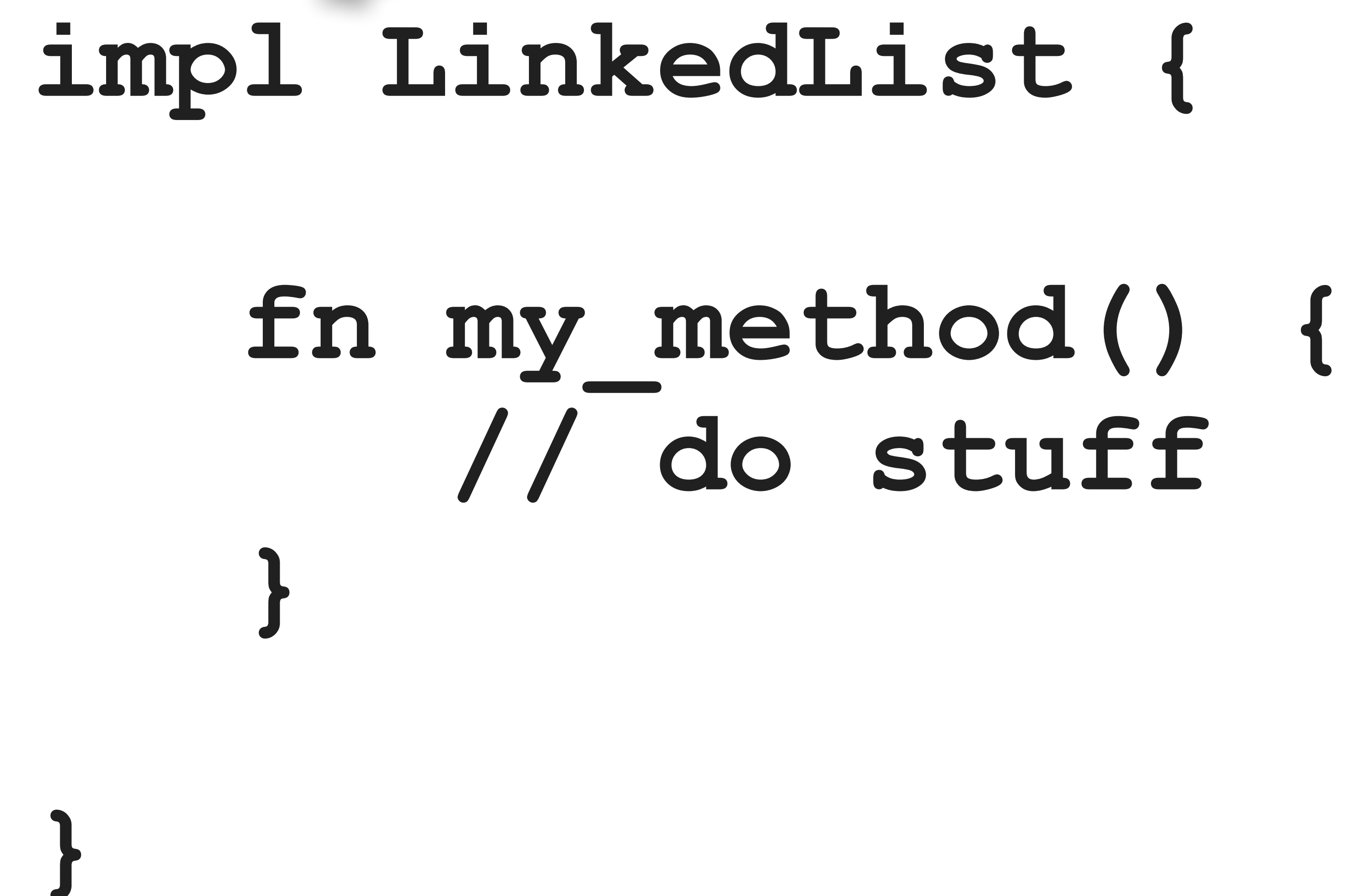
Create a new struct (we know how to do this!)

```
struct LinkedList {

   head: Option<Box<Node>>,
   length: usize, // optional, but may be helpful

}
```

Creating methods for a struct

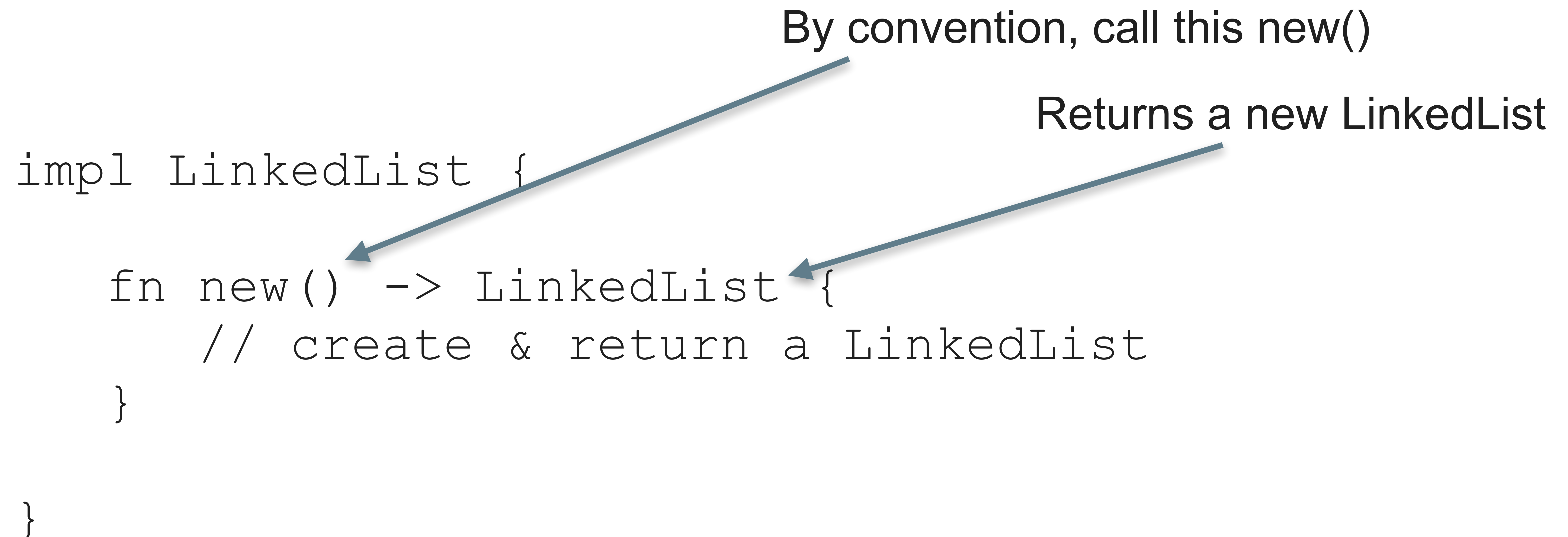
```
struct LinkedList {
   head: Option<Box<Node>>,
   length: usize, // optional, but may be helpful
}
```

“Implementation block”.
All methods associated with “LinkedList” go in here



Let’s make a constructor

- Unlike in C++, constructors are not a specific thing in Rust.
- Rust just has functions.
- By convention, we name “constructors”: new()

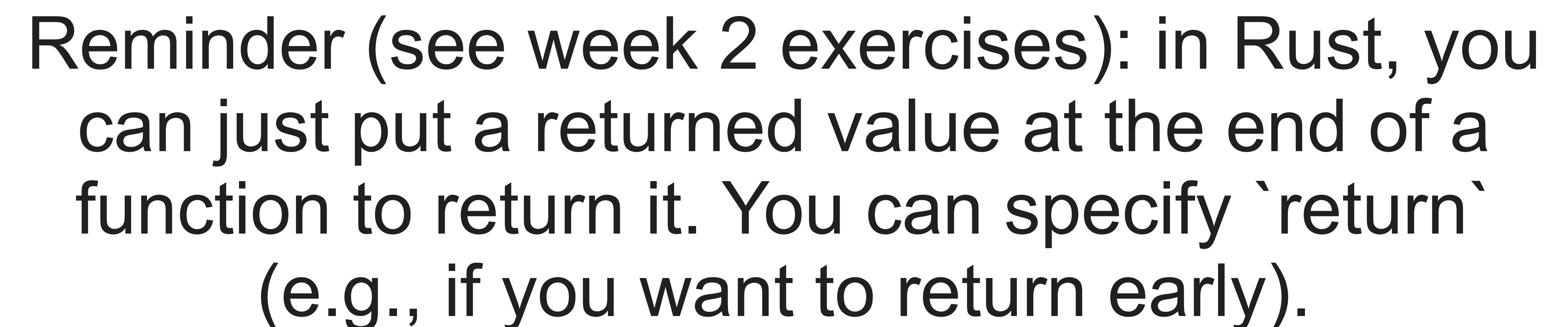


Let's make a constructor

```
struct LinkedList {
   head: Option<Box<Node>>,
   length: usize,
}
```

```
impl LinkedList {

    fn new() -> LinkedList {
        LinkedList { head: None, length: 0 }
    }

}
```

Reminder: syntax to create the struct


Reminder (see week 2 exercises): in Rust, you can just put a returned value at the end of a function to return it. You can specify `return` (e.g., if you want to return early).

Let’s make a constructor

```
struct LinkedList {
   head: Option<Box<Node>>,
   length: usize,
}
```

```
impl LinkedList {

   fn new() -> LinkedList {
      LinkedList { head: None, length: 0 }
   }

}
```

```
fn main() {
    let list1 = LinkedList::new();
}
```

Let’s make a function: take 1

```
fn main() {
    let list1 = LinkedList::new();
    let len = list1.len();
}
```

Let’s make a function: take 1

```
impl LinkedList {

    fn new() -> LinkedList {
        LinkedList { head: None, length: 0 }
    }

    fn len() -> usize {
        length
    }
}
```

does not compile

Let’s make a function: take 2

```
impl LinkedList {

    fn new() -> LinkedList {
        LinkedList { head: None, length: 0 }
    }

    fn len(&self) -> usize {
        self.length
    }
}
```

compiles!

Takes a parameter `self` — “the specific object you are operating on.”

Here, taking an immutable reference to `self`

Let’s make a function: take 2

```
impl LinkedList {

    fn new() -> LinkedList {
        LinkedList { head: None, length: 0 }
    }

    fn len(&self) -> usize {
        self.length
    }
}
```

Aside: why might we want to make sure to pass a reference here, rather than transferring ownership? What’s the difference? Why might the latter be impractical for the typical use case of a linked list?

```
fn main() {
    let list1 = LinkedList::new();
    let len = list1.len();
}
```

immutable reference to list1 implicitly passed as a parameter

Let’s make another function: `front`

- Goal: `lst.front()` to return an immutable reference to the head of the list (frontmost node), or None if list is empty

```
impl LinkedList {

    /* other methods */

    fn front(&self) -> Option<&Box<Node>> {
        (&self.head).as_ref()
    }
}
```

Let’s make another function: `front`

```
fn front(&self) -> Option<&Box<Node>> {
```

- We want this to be an Option, because it could be None (if the list is empty).
- We want &Box<Node>, because returning a reference is probably more practical than transferring ownership (for example, if you’re iterating through a list).

```
(&self.head).as_ref()
```

- Throwback to `as_ref`: converts &Option<T> to Option<&T>
    - self.head is Option<Box<Node>>
    - &self.head is &Option<Box<Node>>
    - (&self.head).as_ref() gives us a new option, containing Box<Node>, or None if self.head is None.

Recap: new material [also in notes]

- After defining a struct, you can define functions associated with it in an implementation block:

```
struct MyType {
    field1: i32,
    field2: String,
}
```

```
impl MyType {

    fn my_method() {

    }
}
```

Recap: new material [also in notes]

- If one of these methods operates on an existing object (e.g., an existing instantiation of MyType), you need a `self` parameter.
- The usual ownership, reference, mutability, etc. rules apply.

```
struct MyType {
    field1: i32,
    field2: String,
}
```

```
impl MyType {
    fn get_field1(&self) -> i32 {
        self.field1
    }
    fn set_field1(&mut self) {
        self.field1 = 1;
    }
}
```

Recap: new material [also in notes]

- By convention, we name the constructor method `new`
  - *This is not a rule enforced by the language. It's just a style convention.*
- It doesn't need a `self` parameter, because it's not operating on an existing object.

```
struct MyType {
    field1: i32,
    field2: String,
}
```

```
impl MyType {

    fn new() -> MyType {
        MyType { field1: 1, field2: "hello".to_string() }
    }

}
```

Notes:

- Note: CS110L is not a Rust class, and it’s *definitely* not an “implementing data structures in Rust” class. (That would be its own huge topic.) The goal of today was to offer a basic intro to object-oriented programming in Rust.
- The Rust book has a great chapter on Associated Functions & Methods
- Personal practice:
  - Can you expand our linked list implementation to more functions? Can you implement `back`?
  - Can you modify `front` to return a mutable reference?