RUSTikales Rust for advanced coders

Plan for today

Plan for today

1. Recap

Plan for today

1. Recap
2. Smart Pointers

1. Recap

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
  - [T] does not implement the Sized trait → Can't use directly, always need a reference

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
  - Pointer into original collection
  - Length of the slice

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str
  - String literals are Arrays behind the scenes

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str
  - String literals are Arrays behind the scenes
  - String structs are Vectors behind the scenes

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str
- Normal Ownership and Borrow Checker rules apply

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str
- Normal Ownership and Borrow Checker rules apply
  - Slices count as immutable borrows
    - Mutable Slices are also possible

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str
- Normal Ownership and Borrow Checker rules apply
  - Slices count as immutable borrows
  - Slices don’t own any elements

1. Recap

- Slices allow us to reference contiguous sequences of a collection, instead of the whole collection
- Type signature for Slices is [T]
- &[T] is a special reference made out of two fields
- You can get a Slice of a collection by using ranges as indices
- Most commonly seen form is the String Slice &str
- Normal Ownership and Borrow Checker rules apply
  - Slices count as immutable borrows
  - Slices don’t own any elements
  - Slices don’t move or copy any data

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

String literals are located in the data section of the executable
→ Array of bytes

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Reference into the data section, knows the size of the literal

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Create a slice of the original String

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Borrow the slice
→ Can‘t know at compile time how big the Slice is
→ Must create reference to the Slice

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Still a slice into the data section

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Data Section												
H	e	l	l	o	,		W	o	r	l	d	!

Stack		
original	ptr	???
	len	???
slice	ptr	???
	len	???

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Data Section												
H	e	l	l	o	,		W	o	r	l	d	!

Stack		
original	ptr	
	len	13
slice	ptr	???
	len	???

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

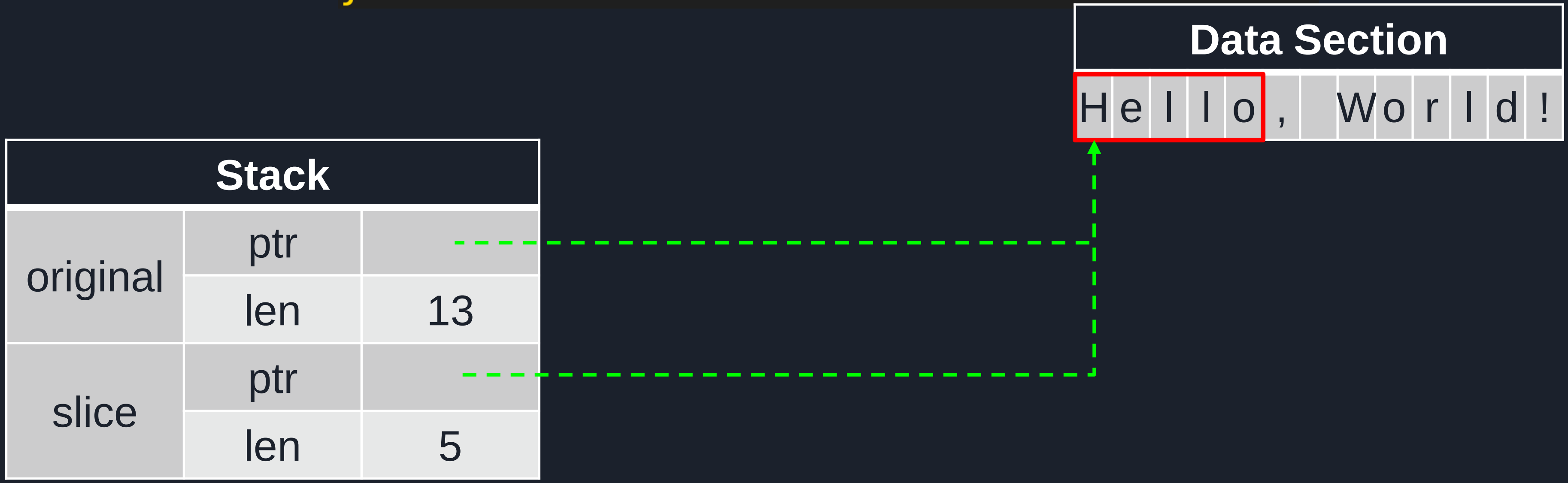
Data Section

H	e	l	l	o	,		W	o	r	l	d	!

Stack		
original	ptr	
	len	13
slice	ptr	???
	len	???

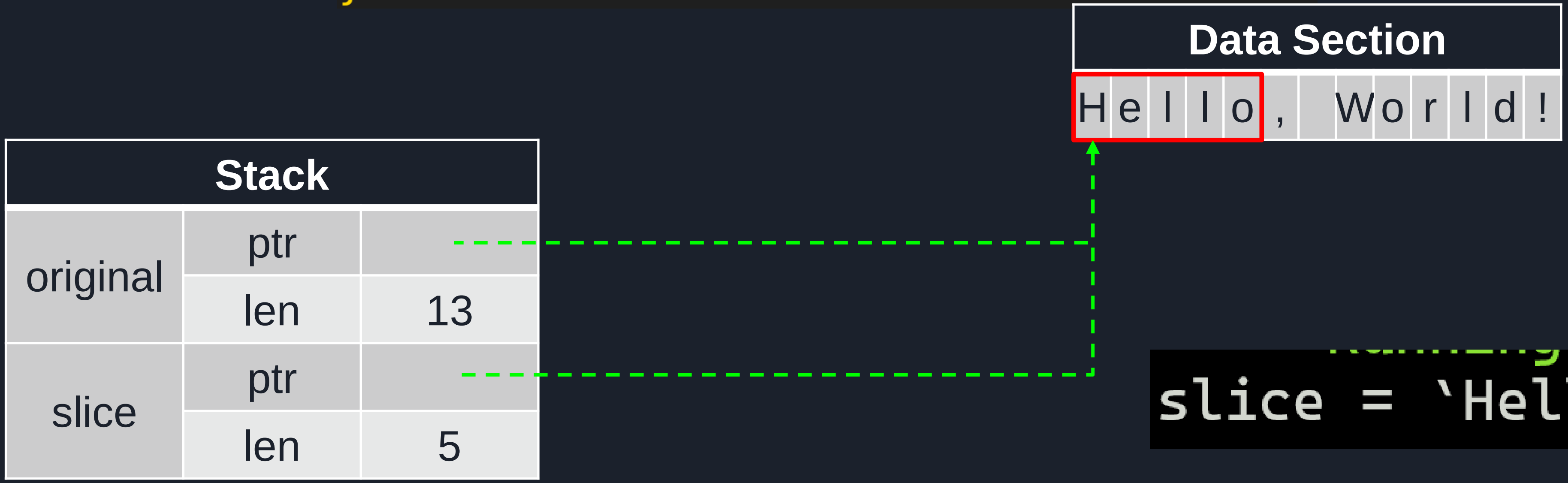
1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```



1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```




```
slice = `Hello`
```

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Data Section

H	e	l	l	o	,		W	o	r	l	d	!

Stack

1. Recap

```rust
fn string_slice() {
    let original: &str = "Hello, World!";
    let slice: &str = &original[0..5];
    println!("slice = `{}`", slice);
}
```

Data Section												
H	e	l	l	o	,		W	o	r	l	d	!

Stack

String literals have the lifetime ‘static
→ They live for the entire duration of the program

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    assert!(slice_arr == slice_vec);
    assert!(&array[..] != &vector[..]);
}
```

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    assert!(slice_arr == slice_vec);
    assert!(&array[..] != &vector[..]);
}
```

You can slice into arrays and vectors

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    assert!(slice_arr == slice_vec);
    assert!(&array[..] != &vector[..]);
}
```

The collection type info is lost,
we only have the elements now

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    assert!(slice_arr == slice_vec);
    assert!(&array[..] != &vector[..]);
}
```

But that allows us to easily compare different data structures!
Here: Check that some elements in an Array are the same as in a Vector

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    takes_slice(slice_arr);
    takes_slice(slice_vec);
    takes_slice(&[5, 20, 35]);
}

fn takes_slice(slice: &[i32]) {
    println!("length: {}", slice.len());
    println!("elems: {:?}", slice);
}
```

1. # Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    takes_slice(slice_arr);
    takes_slice(slice_vec);
    takes_slice(&[5, 20, 35]);
}

fn takes_slice(slice: &[i32]) {
    println!("length: {}", slice.len());
    println!("elems: {:?}", slice);
}
```

Slices allow us to write more efficient functions

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    takes_slice(slice_arr);
    takes_slice(slice_vec);
    takes_slice(&[5, 20, 35]);
}

fn takes_slice(slice: &[i32]) {
    println!("length: {}", slice.len());
    println!("elems: {:?}", slice);
}
```

Slices allow us to write more efficient functions
→ Accepting a Vec<i32> is overkill (needs heap)
→ Big overhead because of .to_vec()

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    takes_slice(slice_arr);
    takes_slice(slice_vec);
    takes_slice(&[5, 20, 35]);
}

fn takes_slice(slice: &[i32]) {
    println!("length: {}", slice.len());
    println!("elems: {:?}", slice);
}
```

Slices allow us to write more efficient functions
→ Accepting a Vec<i32> is overkill
→ Accepting an Array of i32 is difficult
  → Must specify a size at compile time

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    takes_slice(slice_arr);
    takes_slice(slice_vec);
    takes_slice(&[5, 20, 35]);
}

fn takes_slice(slice: &[i32]) {
    println!("length: {}", slice.len());
    println!("elems: {:?}", slice);
}
```

Slices allow us to write more efficient functions
→ Accepting a Vec<i32> is overkill
→ Accepting an Array of i32 is difficult
→ Slices accept both collections, with little overhead

1. Recap

```rust
fn other_slices() {
    let array: [i32; 3] = [15, 20, 25];
    let vector: Vec<i32> = vec![10, 15, 20];
    let slice_arr: &[i32] = &array[0..2];
    let slice_vec: &[i32] = &vector[1..3];
    takes_slice(slice_arr);
    takes_slice(slice_vec);
    takes_slice(&[5, 20, 35]);
}

fn takes_slice(slice: &[i32]) {
    println!("length: {}", slice.len());
    println!("elems: {:?}", slice);
}
```

```
length: 2
elems: [15, 20]
length: 2
elems: [15, 20]
length: 3
elems: [5, 20, 35]
```

2. Smart Pointers

2. Smart Pointers

– Recap on Rust

2. Smart Pointers

- Recap on Rust
  - Ownership

2. Smart Pointers

- Recap on Rust
  - Ownership
    - Every variable, every value, everything in Rust has exactly one owner

2. Smart Pointers

- Recap on Rust
  - Ownership
    - Every variable, every value, everything in Rust has exactly one owner
    - Ownership conflicts are resolved by moving values

2. Smart Pointers

- Recap on Rust
  - Ownership
    - Every variable, every value, everything in Rust has exactly one owner
    - Ownership conflicts are resolved by moving values
    - When the owner is dropped, the value is dropped (memory is freed)

      → Statically known

      → The compiler inserts some code at specific locations to drop values

2. Smart Pointers

- Recap on Rust
  - Ownership
  - Borrow Checker

2. Smart Pointers

- Recap on Rust
  - Ownership
  - Borrow Checker
    - Can either get infinite immutable borrows, or a single mutable borrow, but not both at the same time

2. Smart Pointers

- Recap on Rust
  - Ownership
  - Borrow Checker
    - Can either get infinite immutable borrows, or a single mutable borrow, but not both at the same time
    - Every reference has a Lifetime

2. Smart Pointers

- Recap on Rust
  - Ownership
  - Borrow Checker
  - Memory Safety guarantees

2. Smart Pointers

- Recap on Rust
  - Ownership
  - Borrow Checker
  - Memory Safety guarantees
  - The compiler is very conservative

2. Smart Pointers

- Recap on Rust
  - Ownership
  - Borrow Checker
  - Memory Safety guarantees
  - The compiler is very conservative

    → If a program obeys those rules, it is valid Rust code

    → If the compiler can‘t statically prove that, it has to reject the code

2. Smart Pointers

– If the compiler can't statically prove that the code obeys the rules, it has to reject the code

2. Smart Pointers

- If the compiler can‘t statically prove that the code obeys the rules, it has to reject the code
- It is really easy to write code that brings the compiler to its limit

2. Smart Pointers

- If the compiler can‘t statically prove that the code obeys the rules, it has to reject the code
- It is really easy to write code that brings the compiler to its limit
  - Cyclic data structures

2. Smart Pointers

– If the compiler can't statically prove that the code obeys the rules, it has to reject the code

```
struct BinaryTree<T> {
    value: T,
    left: Option<BinaryTree<T>>,
    right: Option<BinaryTree<T>>,
}
```

2. Smart Pointers

– If the compiler can't statically prove that the code obeys the rules, it has to reject the code

```
struct BinaryTree<T> {
    value: T,
    left: Option<BinaryTree<T>>,
    right: Option<BinaryTree<T>>,
}
```

Structs need to be Sized, the size must be known at compile time.
How big are recursive structs?

2. Smart Pointers

- If the compiler can‘t statically prove that the code obeys the rules, it has to reject the code
- It is really easy to write code that brings the compiler to its limit
  - Cyclic data structures
  - Shared Ownership vs No Ownership at all

2. Smart Pointers

```
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

2. Smart Pointers

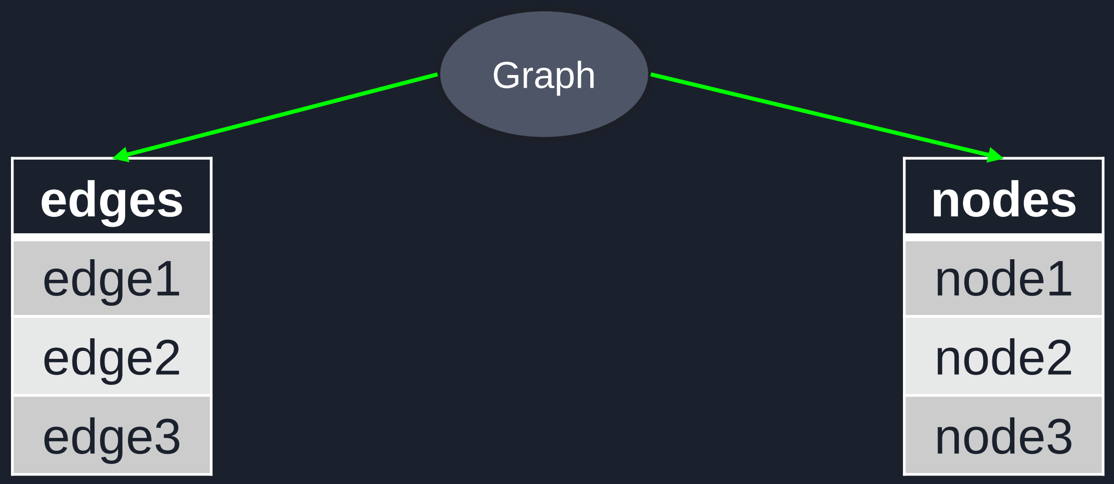
```
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

Who owns what?

2. Smart Pointers

```rust
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

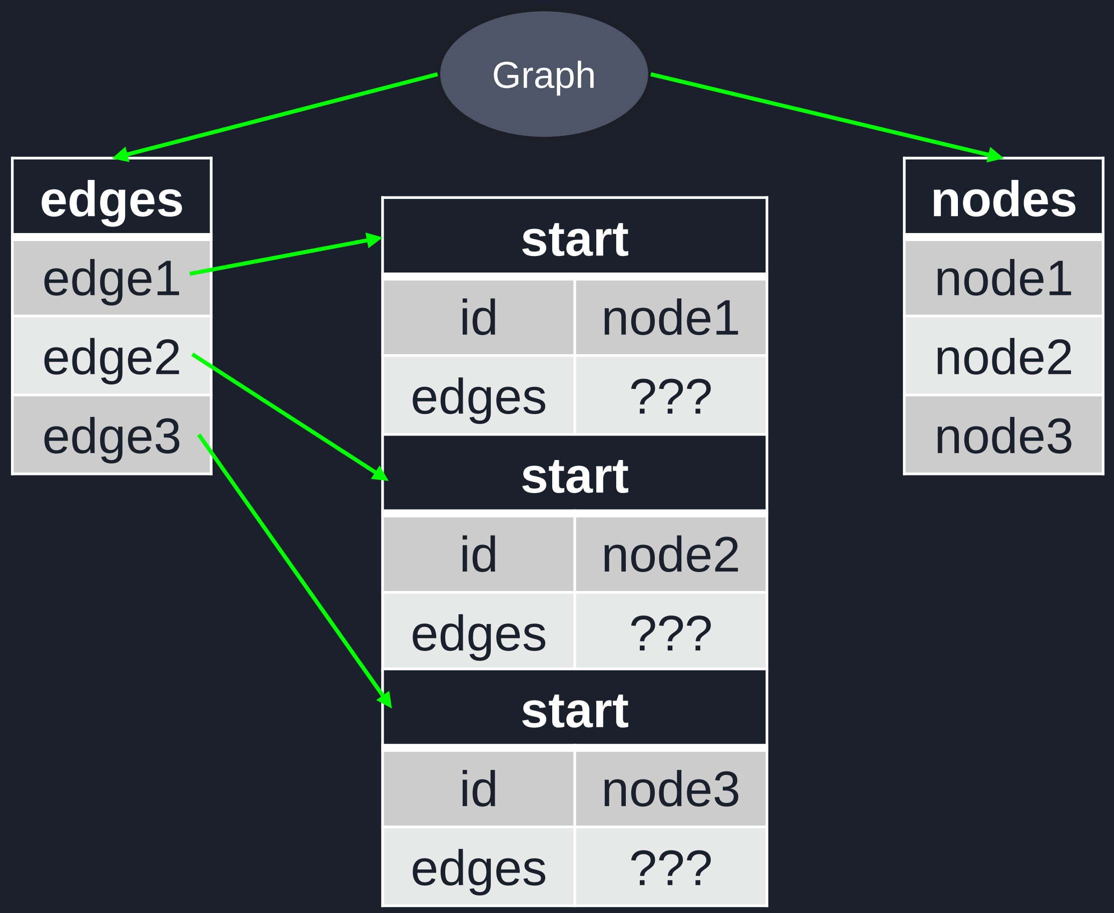
Who owns what?



2. Smart Pointers

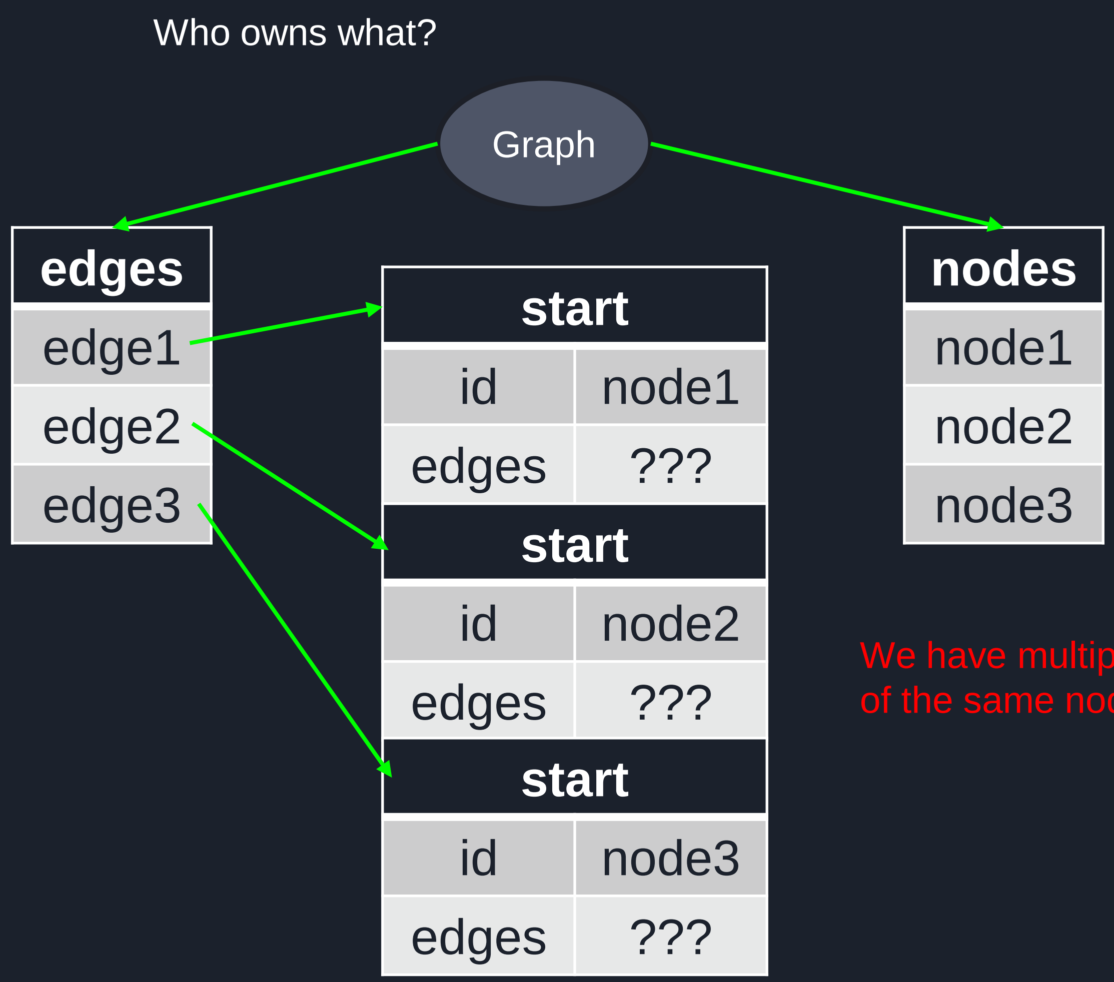
```rust
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

Who owns what?



2. Smart Pointers

```rust
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

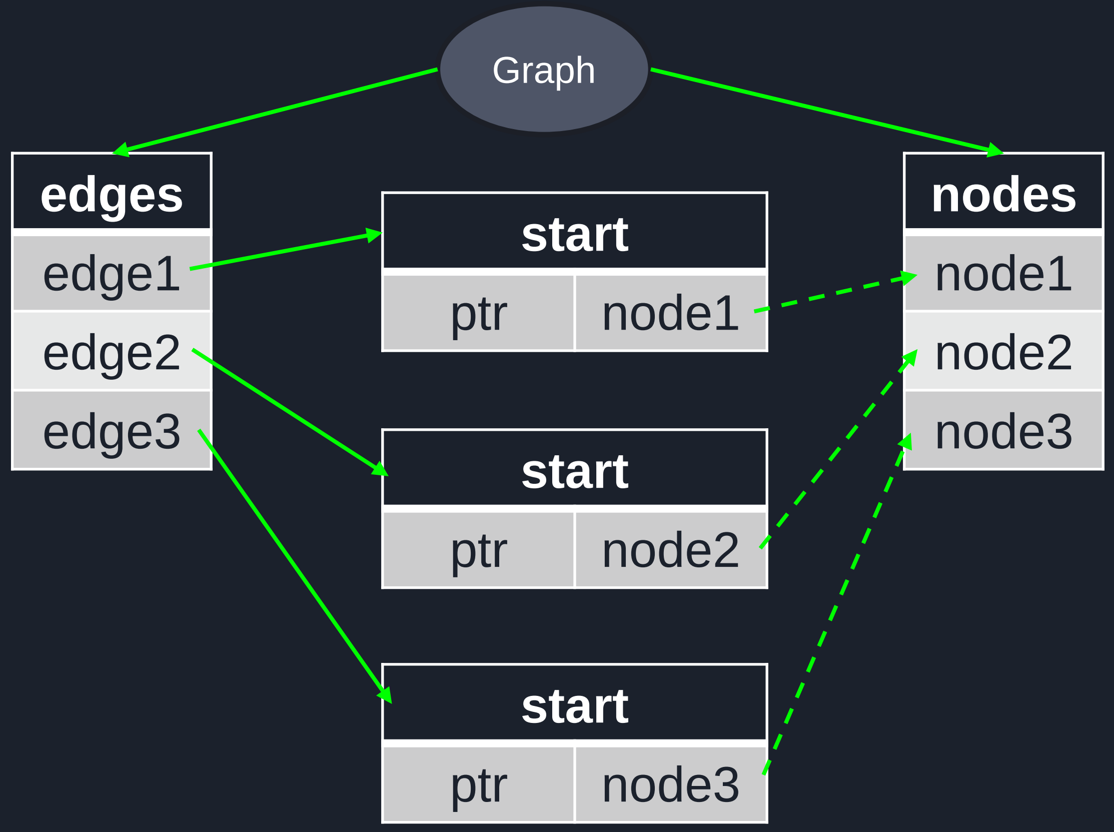


We have multiple copies of the same nodes!

2. Smart Pointers

```
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

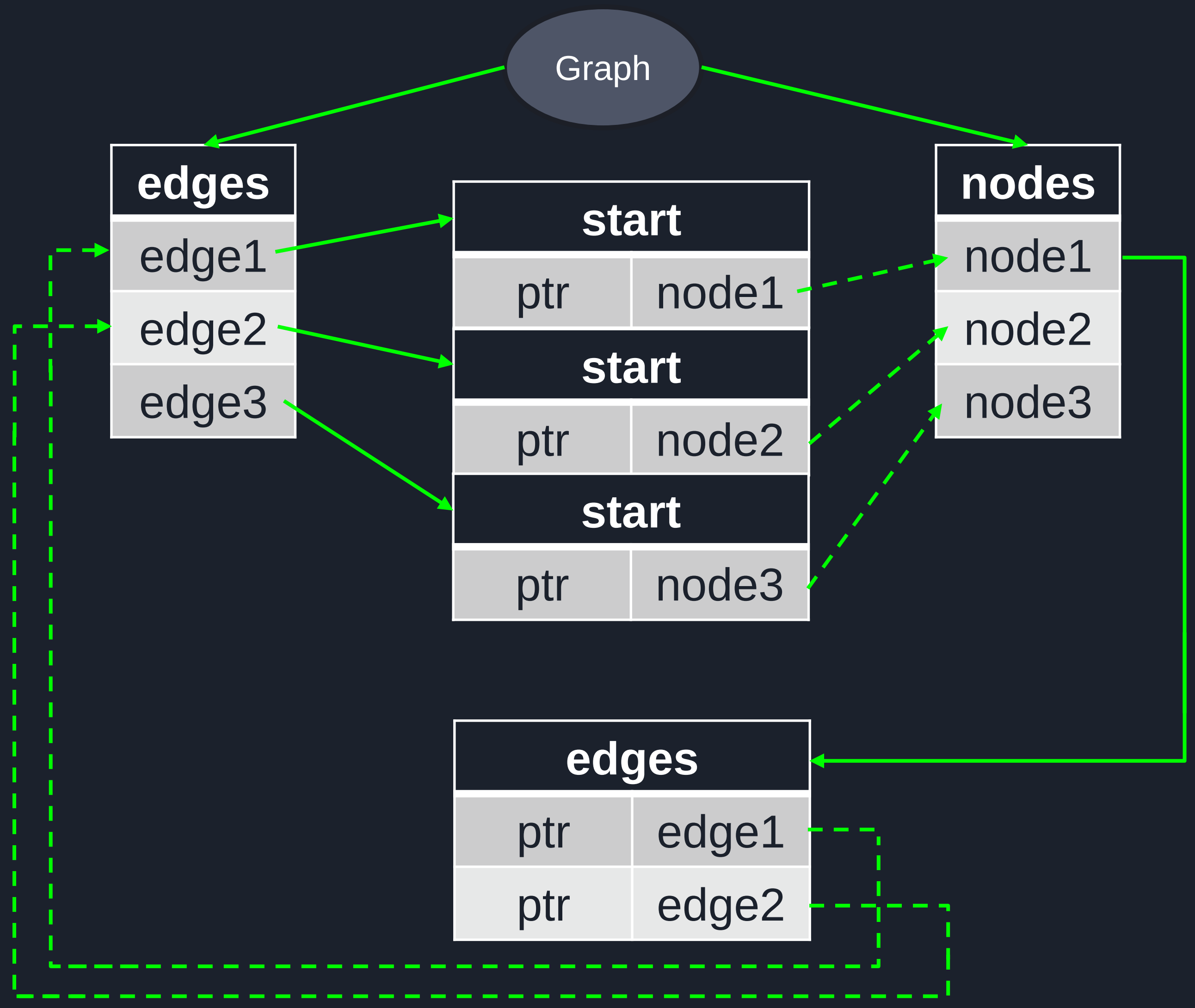
What we actually want:



2. Smart Pointers

```
struct Graph {
    edges: Vec<Edge>,
    nodes: Vec<Node>,
}
0 implementations
struct Edge {
    start: Node,
    end: Node
}
0 implementations
struct Node {
    id: usize,
    edges: Vec<Edge>,
}
```

What we actually want:



2. Smart Pointers

- If the compiler can‘t statically prove that the code obeys the rules, it has to reject the code
- It is really easy to write code that brings the compiler to its limit
  - Cyclic data structures
  - Shared Ownership vs No Ownership at all
  - Convoluted code where we can‘t analyze what‘s borrowed, and where, and how

2. Smart Pointers

- If the compiler can‘t statically prove that the code obeys the rules, it has to reject the code
- It is really easy to write code that brings the compiler to its limit
  - Cyclic data structures
  - Shared Ownership vs No Ownership at all
  - Convoluted code where we can‘t analyze what‘s borrowed, and where, and how
- In those situations (and some more) we need to use Smart Pointers

2. Smart Pointers

– Smart Pointers are data structures that act as normal references

2. Smart Pointers

- Smart Pointers are data structures that act as normal references
  - They implement the Deref trait (and sometimes DerefMut)
    → Unary operator * to access the underlying data

2. Smart Pointers

- Smart Pointers are data structures that act as normal references
  - They implement the Deref trait (and sometimes DerefMut)
  - Additionally, they contain extra metadata and implement specific behavior

2. Smart Pointers

- Smart Pointers are data structures that act as normal references
  - They implement the Deref trait (and sometimes DerefMut)
  - Additionally, they contain extra metadata and implement specific behavior
- More generally: Smart Pointers are a design pattern
  - You can easily write your own Smart Pointers, many libraries offer their own variations
  - As with every design pattern, they have pros and cons

2. Smart Pointers

- The most straightforward Smart Pointer is Box<T>

2. Smart Pointers

- The most straightforward Smart Pointer is Box<T>
- Box<T> doesn’t offer much utility, it only puts the encapsulated value on the heap, instead of storing it on the stack

2. Smart Pointers

- The most straightforward Smart Pointer is Box<T>
- Box<T> doesn't offer much utility, it only puts the encapsulated value on the heap, instead of storing it on the stack
- Box<T> owns the underlying data
  → Data is dropped when Box is dropped

2. Smart Pointers

```rust
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

2. Smart Pointers

```
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

Similar to Option and Result, Box is part of the prelude
→ No special imports necessary

2. Smart Pointers

```rust
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

Moves the person into the Heap

2. Smart Pointers

```
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

Stack		
person	name	???
	age	???
boxed	ptr	???

2. Smart Pointers

```rust
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

Stack		
person	name	Peter
	age	27
boxed	ptr	???

2. Smart Pointers

```rust
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

Stack		
person	name	???
	age	???
boxed	ptr	0x1230

Heap		
0x1230	name	Peter
0x1234	age	27

2. Smart Pointers

```
struct Person {
    name: &'static str,
    age: u8
}
▶ Run | Debug
fn main() {
    let person: Person = Person {
        name: "Peter",
        age: 27
    };
    let boxed: Box<Person> = Box::new(person);
}
```

Underlying value is dropped when the box is dropped

Stack

Heap

2. Smart Pointers

- The most straightforward Smart Pointer is Box<T>
- Box<T> doesn't offer much utility, it only puts the encapsulated value on the heap, instead of storing it on the stack
- Box<T> owns the underlying data
- Because of that, Box<T> is often used in recursive data structures
  - It doesn't matter how big the underlying structure is, Box<T> is always pointer sized

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Add indirection by putting the nodes into a Box

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>
Sizeof BinaryTree<i32> = Sizeof<i32> + 2 * Sizeof<Option<Box<BinaryTree<i32>>>>

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>
Sizeof BinaryTree<i32> = Sizeof<i32> + 2 * Sizeof<Option<Box<BinaryTree<i32>>>>
Sizeof BinaryTree<i32> = 4 + 2 * Sizeof<Box<BinaryTree<i32>>>

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>
Sizeof BinaryTree<i32> = Sizeof<i32> + 2 * Sizeof<Option<Box<BinaryTree<i32>>>>
Sizeof BinaryTree<i32> = 4 + 2 * Sizeof<Box<BinaryTree<i32>>>

Enums are always as big as the biggest variant:
None → 0 bytes
Some(T) -> Sizeof<T>
+1 byte to store which variant it is (the tag),
but Option is optimized, we don‘t have that here

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>
Sizeof BinaryTree<i32> = Sizeof<i32> + 2 * Sizeof<Option<Box<BinaryTree<i32>>>>
Sizeof BinaryTree<i32> = 4 + 2 * Sizeof<Box<BinaryTree<i32>>>
Sizeof BinaryTree<i32> = 4 + 2 * 8 (or 4 on 32bit-systems)

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>
Sizeof BinaryTree<i32> = Sizeof<i32> + 2 * Sizeof<Option<Box<BinaryTree<i32>>>>
Sizeof BinaryTree<i32> = 4 + 2 * Sizeof<Box<BinaryTree<i32>>>
Sizeof BinaryTree<i32> = 4 + 2 * 8
Sizeof BinaryTree<i32> = 20 bytes

2. Smart Pointers

```rust
struct BinaryTree<T> {
    value: T,
    left: Option<Box<BinaryTree<T>>>,
    right: Option<Box<BinaryTree<T>>>,
}
```

Sizeof BinaryTree<T> = Sizeof<value> + Sizeof<left> + Sizeof<right>
Sizeof BinaryTree<i32> = Sizeof<i32> + 2 * Sizeof<Option<Box<BinaryTree<i32>>>>
Sizeof BinaryTree<i32> = 4 + 2 * Sizeof<Box<BinaryTree<i32>>>
Sizeof BinaryTree<i32> = 4 + 2 * 8
Sizeof BinaryTree<i32> = 20 bytes + 4 bytes alignment = 24 bytes :^)

2. Smart Pointers

- The most straightforward Smart Pointer is Box<T>
- Box<T> doesn't offer much utility, it only puts the encapsulated value on the heap, instead of storing it on the stack
- Box<T> owns the underlying data
- Because of that, Box<T> is often used in recursive data structures
  - It doesn't matter how big the underlying structure is, Box<T> is always pointer sized
- Another usecase is when you have a large amount of data, and want to move it a lot

2. Smart Pointers

```rust
struct BigStruct {
    much_data: [i32; 50_000]
}
fn move_data(data: BigStruct) -> BigStruct {
    // A lot of work, pew
    data
}
▶ Run | Debug
fn main() {
    let mut big: BigStruct = BigStruct {
        much_data: [0; 50_000]
    };
    for _ in 0..100_000 {
        big = move_data(big);
    }
}
```

2. Smart Pointers

```rust
struct BigStruct {
    much_data: [i32; 50_000]
}
fn move_data(data: BigStruct) -> BigStruct {
    // A lot of work, pew
    data
}
▶ Run | Debug
fn main() {
    let mut big: BigStruct = BigStruct {
        much_data: [0; 50_000]
    };
    for _ in 0..100_000 {
        big = move_data(big);
    }
}
```

Struct is 200KB big
→ Move 200KB when passing big as argument
→ Move 200KB when putting the return value into big

2. Smart Pointers

```
struct BigStruct {
    much_data: [i32; 50_000]
}
fn move_data(data: Box<BigStruct>) -> Box<BigStruct> {
    // A lot of work, pew
    data
}
▶ Run | Debug
fn main() {
    let mut big: Box<BigStruct> = Box::new(BigStruct {
        much_data: [0; 50_000]
    });
    for _ in 0..100_000 {
        big = move_data(big);
    }
}
```

When working with a Box, we now only need to pass 8 bytes!

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references
- Rc<T> is used for Shared Ownership

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references
- Rc<T> is used for Shared Ownership
- Rc<T> handles Ownership at Runtime, instead of Compiletime

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references
- Rc<T> is used for Shared Ownership
- Rc<T> handles Ownership at Runtime, instead of Compiletime
  - Every time you clone a Rc<T>, the reference count increases by one

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references
- Rc<T> is used for Shared Ownership
- Rc<T> handles Ownership at Runtime, instead of Compiletime
  - Every time you clone a Rc<T>, the reference count increases by one
  - Every time you drop a Rc<T>, the reference count decreases by one

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references
- Rc<T> is used for Shared Ownership
- Rc<T> handles Ownership at Runtime, instead of Compiletime
    - Every time you clone a Rc<T>, the reference count increases by one
    - Every time you drop a Rc<T>, the reference count decreases by one
    - The underlying data is dropped when the reference count reaches 0

2. Smart Pointers

- Another useful Smart Pointer is Rc<T>
- Rc stands for Reference Counted, and does exactly what you might think it does → It counts references
- Rc<T> is used for Shared Ownership
- Rc<T> handles Ownership at Runtime, instead of Compiletime
- Because of Shared Ownership, the underlying data is immutable by default → But we can fix that later

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

2. Smart Pointers

```rust
use std::rc::Rc;
```

Rc is part of the standard library, needs to be imported first

```rust
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

2. Smart Pointers

```rust
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Create a reference counted version of the original
→ Ownership is moved into the Rc → Heap

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Create a copy of the Rc

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Note: It‘s Rust-idiomatic to write method calls of Rc as associated function

Create a copy of the Rc

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Note: It's Rust-idiomatic to write method calls of Rc as associated function
→ rc_orig.clone() might make you think that you clone the underlying data

Create a copy of the Rc

2. Smart Pointers

```rust
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Note: It's Rust-idiomatic to write method calls of Rc as associated function
→ rc_orig.clone() might make you think that you clone the underlying data
→ But you're just cloning the Rc-Metadata

Create a copy of the Rc

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Returns the current reference count

2. Smart Pointers

```rust
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Stack	
original	???
rc_orig	???
other	???

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Stack	
original	OurData<15>
rc_orig	???
other	???

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Stack	
original	???
rc_orig	0x1000
other	???

Heap		
0x1000	data	OurData<15>
0x1008	count	1

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Note: The original data was moved!

Stack	
original	???
rc_orig	0x1000
other	???

Heap		
0x1000	data	OurData<15>
0x1008	count	1

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Stack	
original	???
rc_orig	0x1000
other	0x1000

Heap		
0x1000	data	OurData<15>
0x1008	count	2

2. Smart Pointers

```rust
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

```
Inside: 2
```

Stack	
original	???
rc_orig	0x1000
other	0x1000

Heap		
0x1000	data	OurData<15>
0x1008	count	2

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Call to drop(other)
→ Decrement count

Stack	
original	???
rc_orig	0x1000
other	???

Heap		
0x1000	data	OurData<15>
0x1008	count	1

2. Smart Pointers

```rust
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

```
Inside: 2
Outside: 1
```

Stack	
original	???
rc_orig	0x1000
other	???

Heap		
0x1000	data	OurData<15>
0x1008	count	1

2. Smart Pointers

```rust
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Call to drop(rc_orig)
→ Decrement count

Stack	
original	???
rc_orig	0x1000
other	???

Heap		
0x1000	data	OurData<15>
0x1008	count	0

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Call to drop(rc_orig)
→ Decrement count
→ count is zero → drop data

Stack	
original	???
rc_orig	0x1000
other	???

Heap		
0x1000	data	???
0x1008	count	0

2. Smart Pointers

```
use std::rc::Rc;
0 implementations
struct OurData {
    data: i64,
}
▶ Run | Debug
fn main() {
    let original: OurData = OurData { data: 15 };
    let rc_orig: Rc<OurData> = Rc::new(original);
    {
        let other: Rc<OurData> = Rc::clone(self: &rc_orig);
        println!("Inside: {}", Rc::strong_count(&rc_orig));
    }
    println!("Outside: {}", Rc::strong_count(&rc_orig));
}
```

Stack

2. Smart Pointers

- Rc<T> by itself is very useful, but it's quite limited
  - Because of aliasing, you can only ever borrow immutably, but not mutably

2. Smart Pointers

- Rc<T> by itself is very useful, but it's quite limited
- In many situations you actually want a mutable reference counted value

2. Smart Pointers

- Rc<T> by itself is very useful, but it's quite limited
- In many situations you actually want a mutable reference counted value
- This is where RefCell<T> enters the field

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability
  - Normally you can only modify references via &mut T → Inherited Mutability

```rust
let mut a: i32 = 5;
let b: &mut i32 = &mut a;
*b = 12;
```

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability
  - Normally you can only modify references via &mut T → Inherited Mutability

```rust
let mut a: i32 = 5;
let b: &mut i32 = &mut a;
*b = 12;
```

Need mutable values and references
→ Checked at compile time

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability
  - Normally you can only modify references via &mut T → Inherited Mutability
  - Interior Mutability allows you to modify references via &T

```rust
let a: Cell<i32> = Cell::new(5);
a.set(val: 12);
println!("a = {}", a.get());
```

2. Smart Pointers

– The third often used Smart Pointer is RefCell<T>
– RefCell<T> belongs to a group of data structures called Cells
– All Cells utilize a mechanism called Interior Mutability
  – Normally you can only modify references via &mut T → Inherited Mutability
  – Interior Mutability allows you to modify references via &T

Not mutable!

```
let a: Cell<i32> = Cell::new(5);
a.set(val: 12);
println!("a = {}", a.get());
```

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability
  - Normally you can only modify references via &mut T → Inherited Mutability
  - Interior Mutability allows you to modify references via &T

Not mutable!

```rust
let a: Cell<i32> = Cell::new(5);
a.set(val: 12);
println!("a = {}", a.get());
```

Yet...

```
a = 12
```

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability
- All Cells are equally unique and important, I will only cover RefCell<T> today
  - Cell<T>
  - RefCell<T>
  - OnceCell<T>

2. Smart Pointers

- The third often used Smart Pointer is RefCell<T>
- RefCell<T> belongs to a group of data structures called Cells
- All Cells utilize a mechanism called Interior Mutability
- All Cells are equally unique and important, I will only cover RefCell<T> today
- RefCell<T> uses Interior Mutability to allow Borrow Checking at Runtime

2. Smart Pointers

- RefCell<T> uses Interior Mutability to allow Borrow Checking at Runtime
- Similar to how Rc<T> keeps track of references, RefCell<T> keeps track of current borrows

2. Smart Pointers

- RefCell<T> uses Interior Mutability to allow Borrow Checking at Runtime
- Similar to how Rc<T> keeps track of references, RefCell<T> keeps track of current borrows
- Using borrow() and borrow_mut(), you can borrow the underlying data of RefCell<T>

2. Smart Pointers

- RefCell<T> uses Interior Mutability to allow Borrow Checking at Runtime
- Similar to how Rc<T> keeps track of references, RefCell<T> keeps track of current borrows
- Using borrow() and borrow_mut(), you can borrow the underlying data of RefCell<T>
- Borrows are valid until the end of the scope
  - Rule of thumb: Keep scopes as short as possible, use separate blocks {} if necessary

2. Smart Pointers

- RefCell<T> uses Interior Mutability to allow Borrow Checking at Runtime
- Similar to how Rc<T> keeps track of references, RefCell<T> keeps track of current borrows
- Using borrow() and borrow_mut(), you can borrow the underlying data of RefCell<T>
- Borrows are valid until the end of the scope
- If you violate the Borrow Checking rules, you'll get a panic at runtime!

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

Create RefCell

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

Immutable :^)

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

Immutable borrow in this block

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

The data is 15

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

Mutable borrow, update the data

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

Immutable borrow, print the data again

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
    let orig: Data = Data { data: 15 };
    let rc_orig: RefCell<Data> = RefCell::new(orig);
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

HOWEVER!

2. Smart Pointers

```
struct Data {
    data: i32
}
fn ref_cell() {
```

```
The data is 15
thread 'main' panicked at src\main.rs:22:30:
already mutably borrowed: BorrowError
```

```
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

HOWEVER!

2. Smart Pointers

```rust
struct Data {
    data: i32
}
fn ref_cell() {
```

We just can‘t escape it... :(

```
The data is 15
thread 'main' panicked at src\main.rs:22:30:
already mutably borrowed: BorrowError
```

```rust
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
    let mut mut_borrow: RefMut<Data> = rc_orig.borrow_mut();
    mut_borrow.data = 100;
    {
        let borrow: Ref<Data> = rc_orig.borrow();
        println!("The data is {}", *borrow);
    }
}
```

Mutable borrow in this block!

Borrow Checker violation!

2. Smart Pointers

- Using a combination of Smart Pointers, we can now implement fancy data structures

2. Smart Pointers

```rust
struct Graph {
    edges: Vec<Rc<RefCell<Edge>>>,
    nodes: Vec<Rc<RefCell<Node>>>,
}
1 implementation
struct Edge {
    start: Rc<RefCell<Node>>,
    end: Rc<RefCell<Node>>,
}
2 implementations
struct Node {
    id: usize,
    edges: Vec<Rc<RefCell<Edge>>>,
}
```

2. Smart Pointers

```
struct Graph {
    edges: Vec<Rc<RefCell<Edge>>>,
    nodes: Vec<Rc<RefCell<Node>>>,
}
1 implementation
struct Edge {
    start: Rc<RefCell<Node>>,
    end: Rc<RefCell<Node>>,
}
2 implementations
struct Node {
    id: usize,
    edges: Vec<Rc<RefCell<Edge>>>,
}
```

Edges and Nodes are now shared

2. Smart Pointers

```rust
struct Graph {
    edges: Vec<Rc<RefCell<Edge>>>,
    nodes: Vec<Rc<RefCell<Node>>>,
}
1 implementation
struct Edge {
    start: Rc<RefCell<Node>>,
    end: Rc<RefCell<Node>>,
}
2 implementations
struct Node {
    id: usize,
    edges: Vec<Rc<RefCell<Edge>>>,
}
```

Edges and Nodes are now shared
→ RefCell<> to modify them (e.g. to add an Edge to a Node)

2. Smart Pointers

```rust
impl Node {
    fn new_rc(id: usize) -> Rc<RefCell<Self>> {
        let n: Node = Self {
            id,
            edges: Vec::new()
        };
        Rc::new(RefCell::new(n))
    }
}
impl Drop for Node {
    fn drop(&mut self) {
        println!("Dropping node with id={}", self.id);
    }
}
```

2. Smart Pointers

```rust
impl Node {
    fn new_rc(id: usize) -> Rc<RefCell<Self>> {
        let n: Node = Self {
            id,
            edges: Vec::new()
        };
        Rc::new(RefCell::new(n))
    }
}
impl Drop for Node {
    fn drop(&mut self) {
        println!("Dropping node with id={}", self.id);
    }
}
```

Helper function to easily create smart Nodes

2. Smart Pointers

```rust
impl Node {
    fn new_rc(id: usize) -> Rc<RefCell<Self>> {
        let n: Node = Self {
            id,
            edges: Vec::new()
        };
        Rc::new(RefCell::new(n))
    }
}
impl Drop for Node {
    fn drop(&mut self) {
        println!("Dropping node with id={}", self.id);
    }
}
```

This is called when we drop a Node

2. Smart Pointers

```
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
}
```

2. Smart Pointers

```
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
}
```

Create two nodes with id=0 and id=1

2. Smart Pointers

```rust
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
}
```

```
graph>cargo run
    Finished dev [unopt
     Running `target\de
Dropping node with id=1
Dropping node with id=0

C:\Users\pfhau\GithubPr
```

2. Smart Pointers

```rust
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
}
```

```
graph>cargo run
    Finished dev [unopt
     Running `target\de
Dropping node with id=1
Dropping node with id=0

C:\Users\pfhau\GithubPr
```

Variables are always dropped in the reverse order they're defined in

2. Smart Pointers

```rust
impl Edge {
    fn new_rc(start: &Rc<RefCell<Node>>, end: &Rc<RefCell<Node>>)
    -> Rc<RefCell<Self>> {
        let e: Edge = Edge {
            start: Rc::clone(self: start),
            end: Rc::clone(self: end)
        };
        let re: Rc<RefCell<Edge>> = Rc::new(RefCell::new(e));
        start.borrow_mut().edges.push(Rc::clone(self: &re));
        end.borrow_mut().edges.push(Rc::clone(self: &re));
        re
    }
}
```

2. Smart Pointers

```rust
impl Edge {
    fn new_rc(start: &Rc<RefCell<Node>>, end: &Rc<RefCell<Node>>)
    -> Rc<RefCell<Self>> {
        let e: Edge = Edge {
            start: Rc::clone(self: start),
            end: Rc::clone(self: end)
        };
        let re: Rc<RefCell<Edge>> = Rc::new(RefCell::new(e));
        start.borrow_mut().edges.push(Rc::clone(self: &re));
        end.borrow_mut().edges.push(Rc::clone(self: &re));
        re
    }
}
```

Create smart Edge

2. Smart Pointers

```rust
impl Edge {
    fn new_rc(start: &Rc<RefCell<Node>>, end: &Rc<RefCell<Node>>)
    -> Rc<RefCell<Self>> {
        let e: Edge = Edge {
            start: Rc::clone(self: start),
            end: Rc::clone(self: end)
        };
        let re: Rc<RefCell<Edge>> = Rc::new(RefCell::new(e));
        start.borrow_mut().edges.push(Rc::clone(self: &re));
        end.borrow_mut().edges.push(Rc::clone(self: &re));
        re
    }
}
```

Borrow original nodes, and add the Edge to them

2. Smart Pointers

```rust
impl Edge {
    fn new_rc(start: &Rc<RefCell<Node>>, end: &Rc<RefCell<Node>>)
    -> Rc<RefCell<Self>> {
        let e: Edge = Edge {
            start: Rc::clone(self: start),
            end: Rc::clone(self: end)
        };
        let re: Rc<RefCell<Edge>> = Rc::new(RefCell::new(e));
        start.borrow_mut().edges.push(Rc::clone(self: &re));
        end.borrow_mut().edges.push(Rc::clone(self: &re));
        re
    }
}
```

No mut anywhere, yet this still works :^)

Borrow original nodes, and add the Edge to them

2. Smart Pointers

```rust
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
    let mut e1: Rc<RefCell<Edge>> = Edge::new_rc(start: &n1, end: &n2);
}
```

2. Smart Pointers

```rust
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
    let mut e1: Rc<RefCell<Edge>> = Edge::new_rc(start: &n1, end: &n2);
}
```

Create new Edge between both Nodes

2. Smart Pointers

```rust
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
    let mut e1: Rc<RefCell<Edge>> = Edge::new_rc(start: &n1, end: &n2);
}
```

```
graph>cargo run
    Finished dev [unopt
     Running `target\de

C:\Users\pfhau\GithubPr
```

2. Smart Pointers

```rust
pub fn main() {
    let mut n1: Rc<RefCell<Node>> = Node::new_rc(id: 0);
    let mut n2: Rc<RefCell<Node>> = Node::new_rc(id: 1);
    let mut e1: Rc<RefCell<Edge>> = Edge::new_rc(start: &n1, end: &n2);
}
```

```
graph>cargo run
    Finished dev [unopt
     Running `target\de

C:\Users\pfhau\GithubPr
```

Congratulations!

We created a memory leak!

3. Next time

- Fixing memory leaks in Rust
- Declarative Macros