Lecture 3

Defining New Types

Which would cause undefined behavior if allowed by compiler?

```
fn move_a_string(s: String) {
}
```

A

```
fn main() {
    let s = String::from("1905");
    move_a_string(s);
    let s2 = s;
}
```

B

```
fn main() {
    let s = String::from("1905");
    let s2 = s;
    move_a_string(s);
}
```

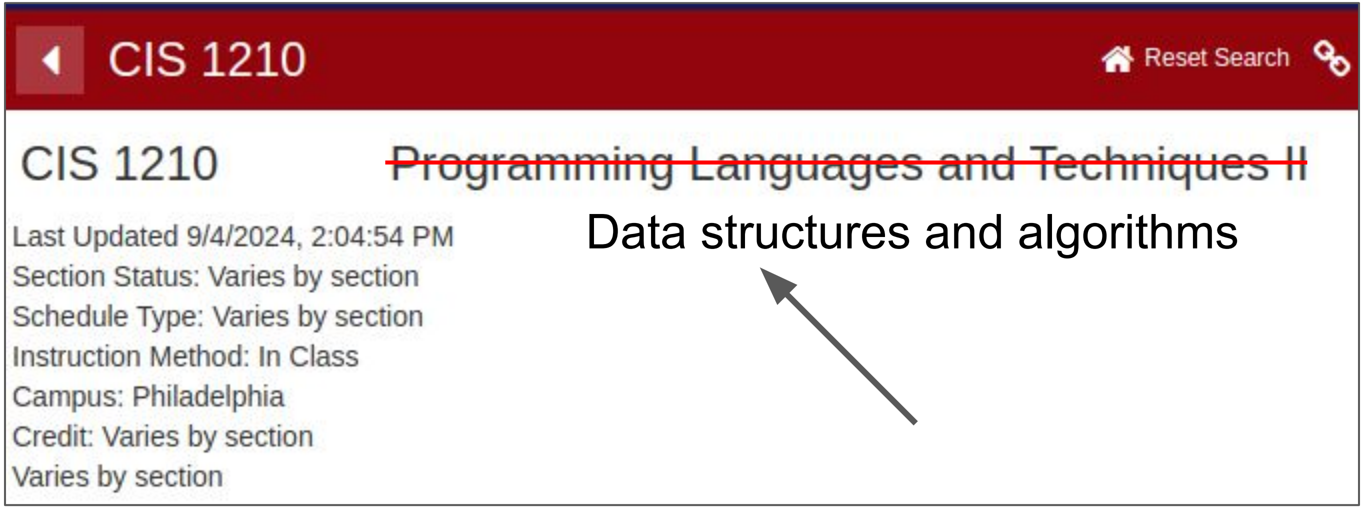
C

```
fn main() {
    let s = String::from("1905");
    let s2 = s;
    println!("{}", s);
    move_a_string(s2);
}
```

D

```
fn main() {
    let s = String::from("1905");
    move_a_string(s);
    println!("{}", s);
}
```

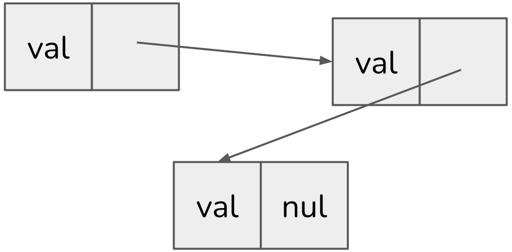
What are we missing?



Today: Data Structures!

Trying to make a linked list

What's a linked list?



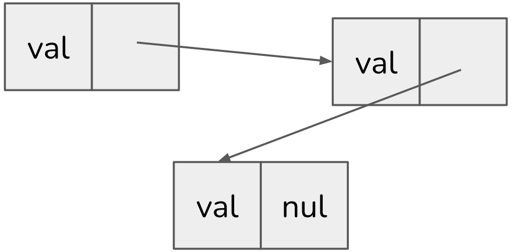
How would you go about implementing a Linked List class in C or C++?

- 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?

Trying to make a linked list

What's a linked list?




```
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 data types in Rust

```
struct Person {
    name: String,
    location: String,
}
fn main() {
    let me = Person {
        name: String::from("paul"),
        location: String::from("Philadelphia")
    };
    println!("{} lives in {}",
        me.name,
        me.location);
}
```

`struct` keyword declares new structs

- each member has a name and a type
- instantiate structs using {}

How to make a Node?

C:

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

Rust:

```
struct Node {
    value: i32,
    next: Node,
}
```

Infinitely sized struct

```
struct Node {
    value: i32,
    next: &Node,
}
```

Borrowing whose data?

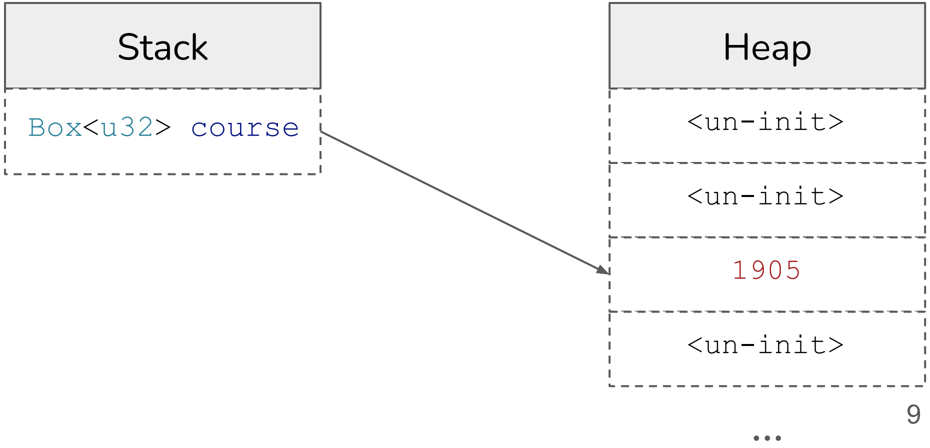
```
struct Node {
    value: i32,
    next: /* pointer to a node...? */
}
```

How to make a Node? Box!

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

- Make a Box of some type T
- a T gets put on heap, Box points to that T
- Box owns that T. When Box goes out of scope, the T is destroyed.

```
fn main() {
    let course = Box::new(1905);
}
```



Single Node List

```
struct Node {
    value: i32,
    next: Box<Node>
}
fn main() {
    let list = Box::new(Node {
        value: 1905,
        next: /* null??*/
    });
}
```

Single Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let list = Box::new(Node {
        value: 1905,
        next: None
    });
}
```

An `Option<T>` is either a `T` or `None`.

```
fn main() {
    let student_grade: Option<char> = Some('A');
    let instructor_grade: Option<char> = None;
}
```

Two Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
   let first = Box::new(Node {
       value: 1905,
       next: None
   });
   let second = Box::new(Node {value: 1200,next: None});
   first.next = second;
}
```

What's wrong with this?

Two Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let second = Box::new(Node {value: 1200,next: None});
    first.next = Some(second);
}
```

Three Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let mut second = Box::new(Node {value: 1200,next: None});
    let third = Box::new(Node {value: 4100,next: None});
    first.next = Some(second);
    second.next = Some(third);
}
```

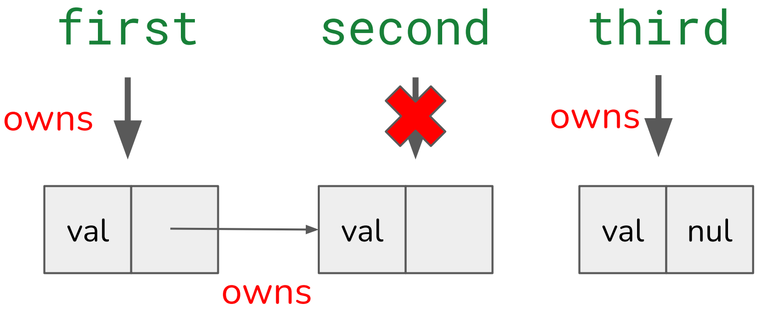
```
error[E0382]: assign to part of moved value: `*second`
 --> list.rs:15:5
   |
13 |     let third = Box::new(Node {value: 4100,next: None});
14 |     first.next = Some(second);
   |                       ------ value moved here
15 |     second.next = Some(third);
   |     ^^^^^^^^^^^ value partially assigned here after move
```

`structs` own their data

- therefore, assigning to a struct member transfers ownership

Three Node List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let mut second = Box::new(Node {value: 1200,next: None});
    let third = Box::new(Node {value: 4100,next: None});
    first.next = Some(second);
    second.next = Some(third);
}
```



Implication: when `first` is dropped:

- 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.

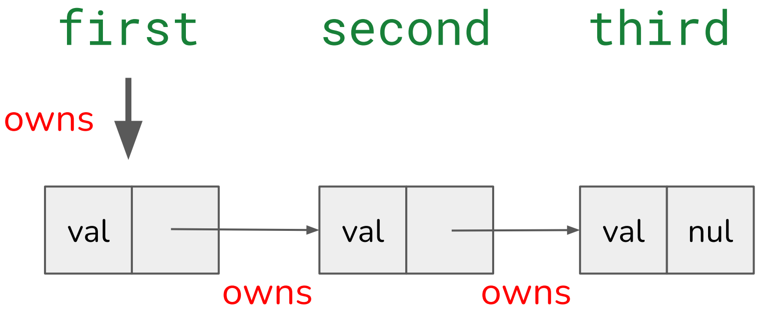
"Chain of ownership"

Three Node List Second Attempt

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let mut first = Box::new(Node {
        value: 1905,
        next: None
    });
    let mut second = Box::new(Node {value: 1200,next: None});
    let third = Box::new(Node {value: 4100,next: None});
    second.next = Some(third);
    first.next = Some(second);

}
```

swap order



Traversing List

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

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let curr = /* ?? */;
    while curr != /* NULL ? */ {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let curr = /* ?? */;
    while curr != /* NULL ? */ {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

What should `curr` be?

- Can't use pointers
- Don't want to take ownership

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

`curr` has type `Option<&Box<Node>>`

- contains either a reference to a box containg `Node` or `None`

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

```
error[E0609]: no field `value` on type `Option<&Box<_>>`
 --> list.rs:11:30
   |
11 |         println!("{}", *curr.value);
   |
```

Traversing List

```rust
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

```rust
loop {
    match curr {
        Some(node) => {
            println!("{}",
node.value);
            curr = node.next.as_ref();
        },
        None => break
    }
}
```

Traversing List

```
struct Node {
    value: i32,
    next: Option<Box<Node>>
}
fn main() {
    let first: Box<Node> = todo!();
    let mut curr = Some(&first);
    while curr != None {
        println!("{}", curr.value);
        curr = curr.next;
    }
}
```

```
loop {
    match curr {
        Some(node) => {
            println!("{}",
node.value);
            curr = node.next;
        },
        None => break
    }
}
```

Separating functionality

```
std::list<int> myList;
myList.push_front (200);
myList.push_front (300);
myList.pop_back ();
```

Goal: associate functionality with data by writing methods like `push_front`

Separating functionality

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

Separating functionality

```
struct LinkedList {
    head: Option<Box<Node>>,
    length: usize, // optional
}
impl LinkedList {
    fn new() -> LinkedList {
        LinkedList {
            head: None,
            length: 0,
        }
    }
}
```

`impl` blocks:

- write functions associated with a type
- accessible as `LinkedList::new()`

Constructors:

- don't exist in Rust
- By convention, provide a `new` function to create instances of your type

Separating functionality

```
struct LinkedList {
    head: Option<Box<Node>>,
    length: usize, // optional
}
impl LinkedList {
    fn new() -> LinkedList {
        LinkedList {
            head: None,
            length: 0,
        }
    }
    fn len() -> usize {
        length
    }
}
```

```
error[E0425]: cannot find value `length` in this scope
  --> list.rs:14:9
   |
14 |         length
   |         ^^^^^^^
   |
```

Separating functionality

```
struct LinkedList {
    head: Option<Box<Node>>,
    length: usize, // optional
}
impl LinkedList {
    fn new() -> LinkedList {
        LinkedList {
            head: None,
            length: 0,
        }
    }
    fn len(&self) -> usize {
        self.length
    }
}
```

Methods

- Just functions that take a `self` parameter
- Can take `self`, `&self`, or `&mut self`

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

Quiz: `self`, `&self`, or `&mut self`

```
impl String {
    fn pop_last(???)
}

impl String {
    fn to_uppercase(???) -> String
}

impl String {
    fn suffix(???) -> &str
}

impl u32 {
    fn increment(???)
}
```

Structs

Declared with `struct` keyword

- Can't contain themselves directly, use a Box to break up recursion
- Initialized with brackets (`Node {value:1}`)

Declare functions associated with a struct using an `impl` block

- **associated functions**: don't take a `self` parameter and are called like `Node::new()`
- **methods**: take `self`, `&self`, or `&mut self` and are called like `list.len()`;

By convention, provide a `new` function that acts as a constructor

`Box` owns a value allocated on the heap

- When the box goes out of scope, the value is deallocated
- Auto-deref `Box<T>` into `&T` or `&mut T`

Structs have ownership of their values

- Accessing a struct element can move data out of the struct
- Assigning to a struct element can move data into that struct

Other structs you might see: tuple structs

```
struct Point { x: i32, y: i32 }
fn main() {
    let p = Point { x: 1, y: 2 };
    let x = p.x;
    let y = p.y;
    match p {
        Point { x: x_coord, y: y_coord } =>
{
            println!("{}, {}", x, y);
        }
    }
}
```

```
struct Point(i32, i32)
fn main() {
    let p = Point(1, 2);
    let x = p.0;
    let y = p.1;
    match p {
        Point(x, y) => {
            println!("{}, {}", x, y);
        }
    }
}
```

Struct field names are optional–structs without field names are “tuple structs”

Other structs you might see: wrapper types

```
impl f32 {
    fn to_centimeters(self) -> f32 {
        self * 2.54
    }
}
```

```
error[E0390]: cannot define inherent `impl` for
primitive types
--> wrapper.rs:1:1
  |
1 | impl f32 {
  | ^^^^^^^^
```

```
struct Inches(f32);
impl Inches {
    fn to_centimeters(self) -> f32 {
        self.0 * 2.54
    }
}
```

Wrap an existing type in a struct

- Separate functionality (e.g. distinguish inches from centimeters at the type leve)
- Add functionality to primitive types

Making our own Option

`Option`: a type that is a value *OR* no value

`struct`s: a type that is a value *AND* another value (and another and another…)

No way to implement `Option` with `struct`

- Need a new language construct…

Making our own Option

```
enum NumOption {
    Some(u32),
    None
}

fn main() {
    let id = NumOption::Some(5);
    match id {
        NumOption::Some(i) =>
            println!("{} is some", i),
        NumOption::None =>
            println!("None")
    }
}
```

Enums!

- Better than C enums -> can contain data
- Like OCaml `type` keyword

`NumOption` can be in one of two states:

- `Some`, in which case a value of type `u32` is guaranteed to be present
- None, in which case no values are present

Access different constructors using `::` syntax

Destructure using pattern matching

Making our own Option

```rust
enum NumOption {
    Some(u32),
    None
}

impl NumOption {
    fn subtract_one(&mut self) {
        match self {
            NumOption::Some(i) => *i -= 1,
            NumOption::None => {}
        }
    }
}
```

Enums can have methods/associated functions as well

Quiz time

```
enum Tree {
   Node(Tree, Tree),
   Leaf(u32),
}
```

What don't you like?

Option’s Cousin: Result

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

`Result` has either a success value of type `T`, or an error value of type `E`.

- `E` contains data (often an error message) that clarifies what the exact error was

```
fn create(path: String) -> Result<File, IoError>

impl f32 {
    fn from_str(src: &str) -> Result<f32, ParseFloatError>
}
```

Preferred over `Option` when more context for the error is needed

Quiz: Result or Option?

```
fn divide(numerator: f32, denominator: f32) -> ????<f32>

fn binary_search(haystack: &[i32], needle: i32) -> ???<usize>

fn write(path: String, contents: &[u8]) -> ???<usize>

fn first_char(s: &str) -> ???<char>
```

Error Handling Woes

```
fn main() -> Result<(), &str> {
    let mut file = match File::create("foo.txt") {
        Ok(file) => file,
        Err(_) => return Err("Failed to create file"),
    };
    match file.write_all(b"Hello, world!") {
        Ok(_) => {},
        Err(_) => return Err("Failed to write to file"),
    };
    match file.flush() {
        Ok(_) => {},
        Err(_) => return Err("Failed to flush file"),
    };
    return Ok(());
}
```

So much code just to open and write to a file!

- Most of it's error handling
- There must be a better way...

Error Handling Woes

```
fn main() -> Result<(), &'static str> {
    let mut file = File::create("foo.txt").unwrap();
    file.write_all(b"Hello, world!").unwrap();
    file.flush().unwrap();
    return Ok(());
}
```

Just `unwrap` it all

- Method available on `Option` and `Result`
- Returns the inner type or aborts the program if not available (i.e. `None` or `Err`)

About print...

So far, we've seen magic printing with `println!` But if we use our own types...

```rust
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{}", id);
}
```

```
error[E0277]: `Id` doesn't implement `std::fmt::Display`
--> print.rs:7:20
  |
7 |     println!("{}", id);
  |                    ^^ `Id` can't be formatted with the default
formatter
```

But implementing print functions is boilerplate, just print every member right?

Deriving traits

So far, we've seen magic printing with `println!` But if we use our own types...

```
#[derive(Debug)]
struct Id {
    id: u32
}

fn main() {
    let id = Id { id: 1905 };
    println!("{:?}", id);
}
```

Use `#[derive(...)]` to automatically implement functionality

- e.g. printing, hashing

```
> rustc print.rs && ./print
Id { id: 1905 }
```

Acknowledgements

Inspiration for these slides drawn from cs110L at Stanford