Structs

Goals For Today

- Review
- Modules & pub/paths
- Introduce Structs
- Syntax shortcuts

Don’t Forget!

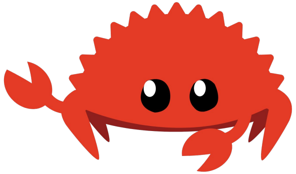
- HW4 Due Today 11:59pm
- MP1 Due 2/23

But First...

🦀Rust🦀

Is there another way to set a &mut String to empty besides using .clear()

Wondering if prairie learn could also tell us why our code doesn't compile.



Some Review of Ownership

There are three major rules of ownership:

1. Each value in Rust has variable called it's **owner**
2. There can only be one **owner**
3. When the owner goes out of scope, the value is dropped

Some Review of Ownership

1. An ampersand (&) represents a *reference*
2. Allows you to refer to some value without taking *ownership* of it
3. We call the action of creating a reference *borrowing*

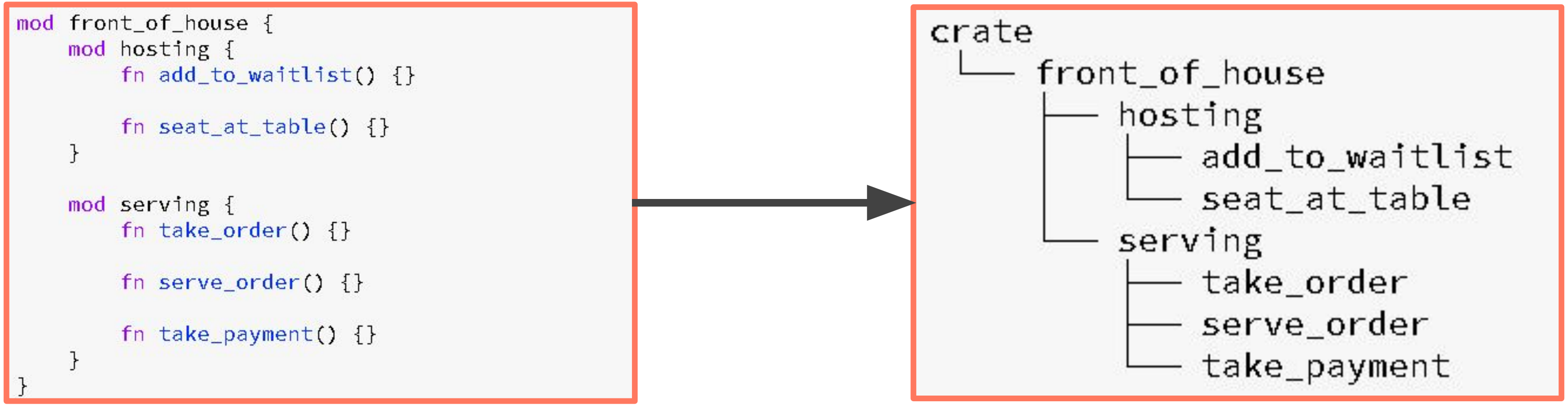
Some Review of Ownership

1. An ampersand (&) represents a *reference*
2. Allows you to refer to some value without taking *ownership* of it
3. We call the action of creating a reference *borrowing*

4. At any given time, you can have either:
   a. one mutable reference using **&mut** or…
   b. An infinite number of immutable references using &

Modules/pub/Paths

Modules in Rust allow us to organize our code and control privacy.



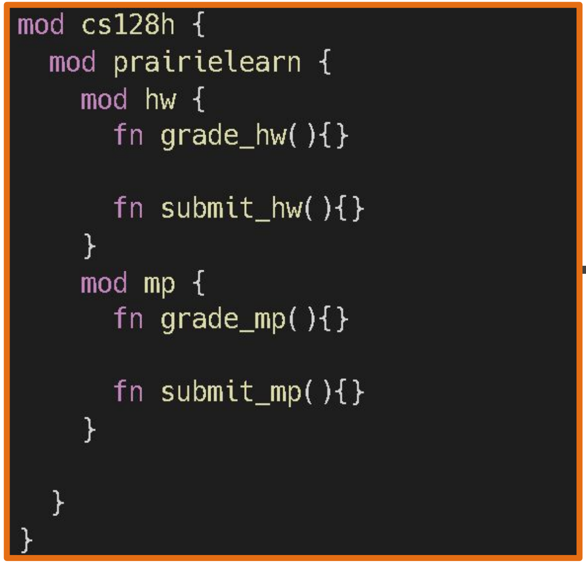
```rust
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

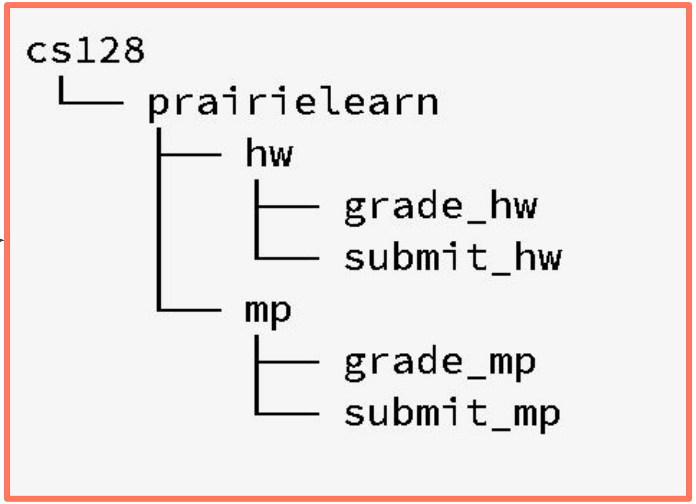
https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Modules/pub/Paths

Modules in Rust allow us to organize our code and control privacy.



```
mod cs128h {
  mod prairielearn {
    mod hw {
      fn grade_hw( ){}

      fn submit_hw( ){}
    }
    mod mp {
      fn grade_mp( ){}

      fn submit_mp( ){}
    }

  }
}
```



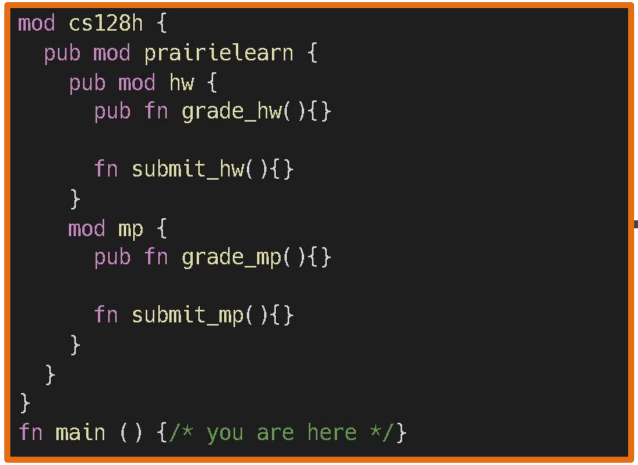
```
cs128
 └── prairielearn
      ├── hw
      │    ├── grade_hw
      │    └── submit_hw
      └── mp
           ├── grade_mp
           └── submit_mp
```

https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

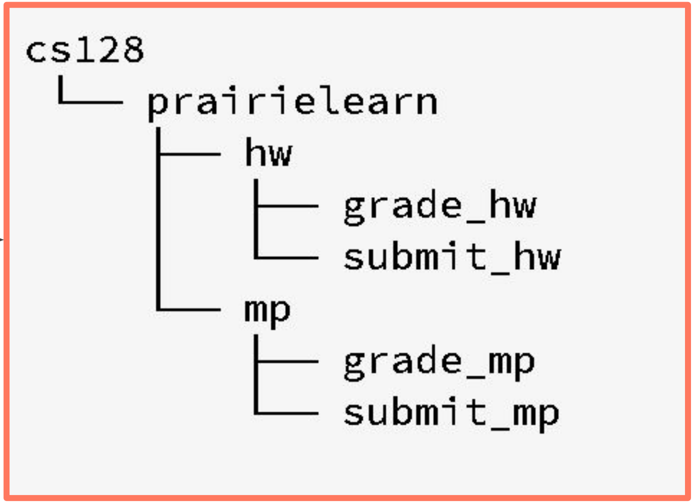
Modules/pub/Paths

Modules in Rust allow us to organize our code and control privacy.

By default, modules are private. So, we can use the **pub** keyword to make them public.



```
mod cs128h {
  pub mod prairielearn {
    pub mod hw {
      pub fn grade_hw(){}

      fn submit_hw(){}
    }
    mod mp {
      pub fn grade_mp(){}

      fn submit_mp(){}
    }
  }
}
fn main () {/* you are here */}
```

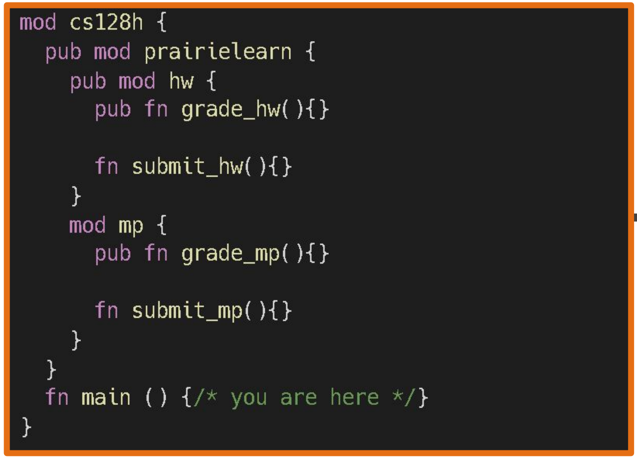


https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

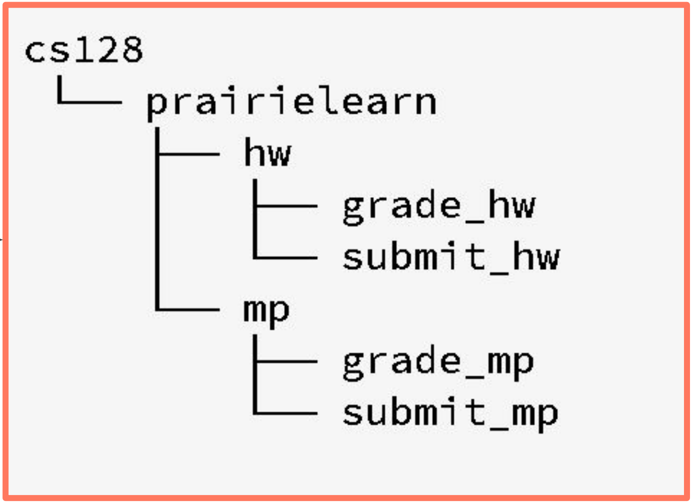
Modules/pub/Paths

Modules in Rust allow us to organize our code and control privacy.

By default, modules are private. So, we can use the **pub** keyword to make them public.



```
mod cs128h {
  pub mod prairielearn {
    pub mod hw {
      pub fn grade_hw(){}

      fn submit_hw(){}
    }
    mod mp {
      pub fn grade_mp(){}

      fn submit_mp(){}
    }
  }
  fn main () {/* you are here */}
}
```

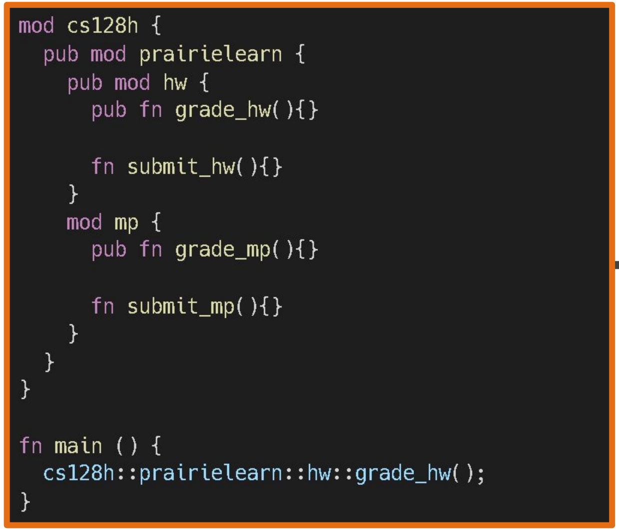


```
cs128
 └── prairielearn
      ├── hw
      │    ├── grade_hw
      │    └── submit_hw
      └── mp
           ├── grade_mp
           └── submit_mp
```

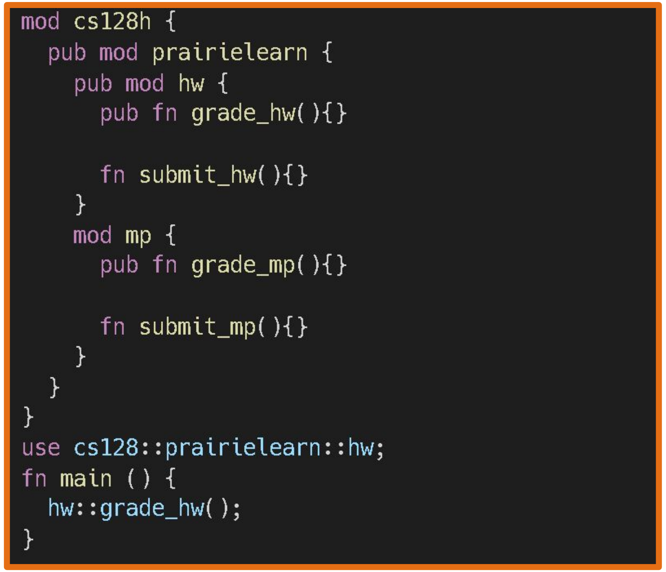
https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Modules/pub/Paths

The **use** keyword can make commonly used paths shorter.



```
mod cs128h {
  pub mod prairielearn {
    pub mod hw {
      pub fn grade_hw(){}

      fn submit_hw(){}
    }
    mod mp {
      pub fn grade_mp(){}

      fn submit_mp(){}
    }
  }
}

fn main () {
  cs128h::prairielearn::hw::grade_hw();
}
```



```
mod cs128h {
  pub mod prairielearn {
    pub mod hw {
      pub fn grade_hw(){}

      fn submit_hw(){}
    }
    mod mp {
      pub fn grade_mp(){}

      fn submit_mp(){}
    }
  }
}
use cs128::prairielearn::hw;
fn main () {
  hw::grade_hw();
}
```

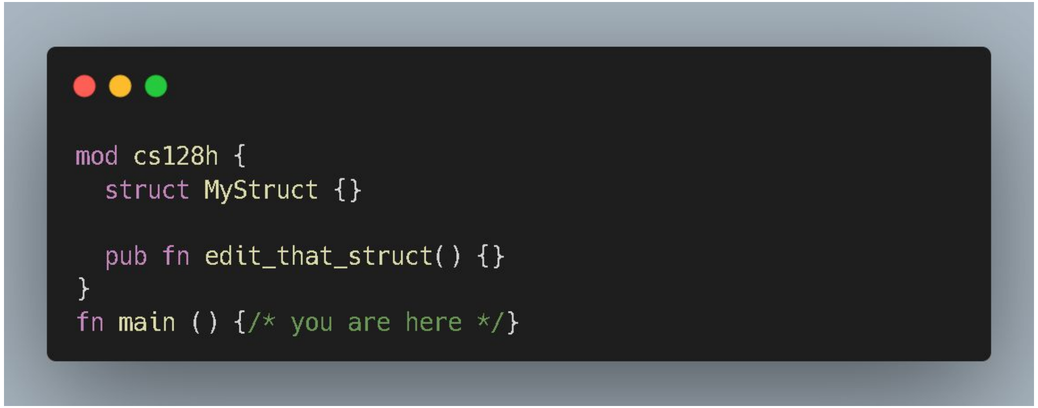
https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Modules/pub/Paths

How to create modules

1. `mod` keyword

src/main.rs



```
mod cs128h {
  struct MyStruct {}

  pub fn edit_that_struct() {}
}
fn main () {/* you are here */}
```

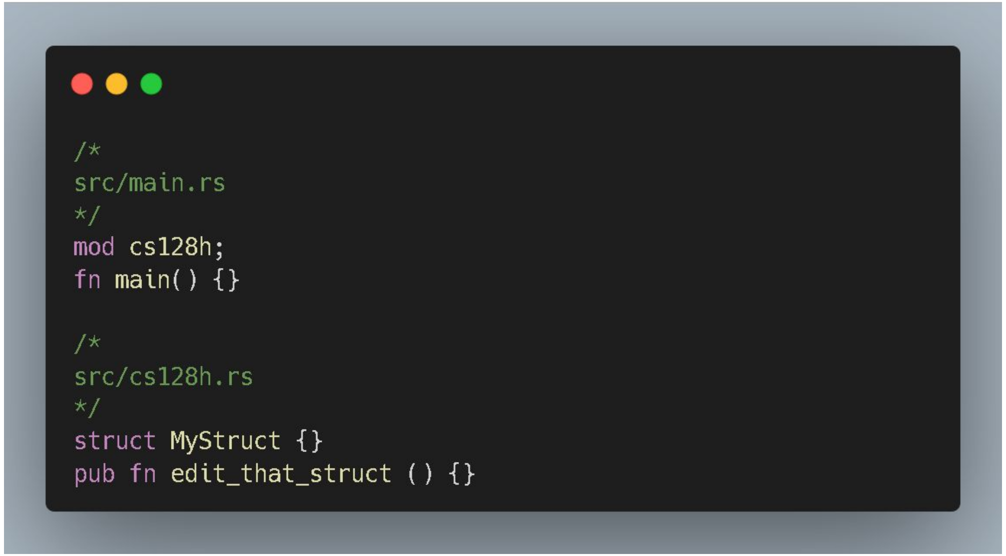
https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Modules/pub/Paths

How to create modules

1. `mod` keyword
2. In a file with the module's name

src/main.rs



```
/*
src/main.rs
*/
mod cs128h;
fn main() {}

/*
src/cs128h.rs
*/
struct MyStruct {}
pub fn edit_that_struct () {}
```

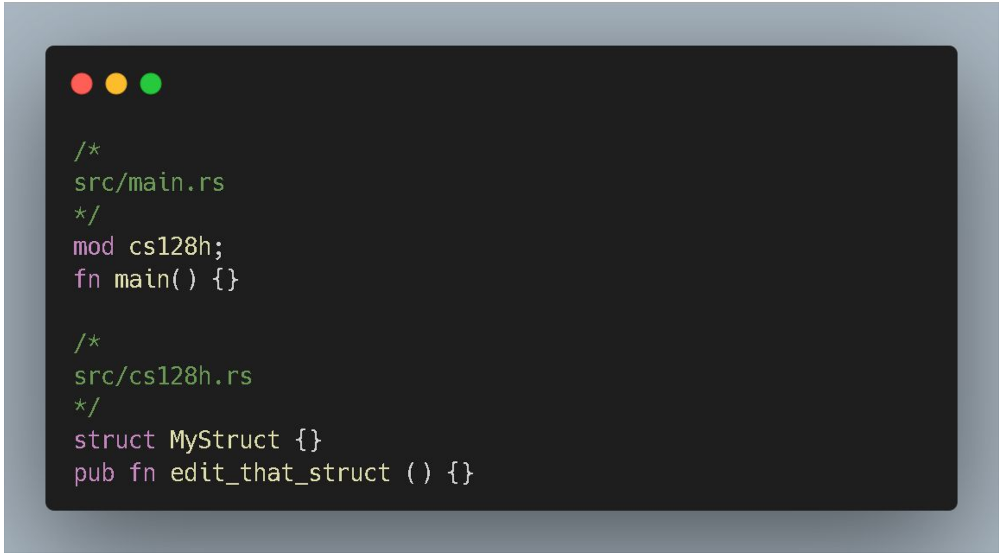
https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Modules/pub/Paths

How to create modules

1. `mod` keyword
2. In a file with the module's name
3. In a folder in the mod.rs file

src/main.rs



```
/*
src/main.rs
*/
mod cs128h;
fn main() {}

/*
src/cs128h.rs
*/
struct MyStruct {}
pub fn edit_that_struct () {}
```

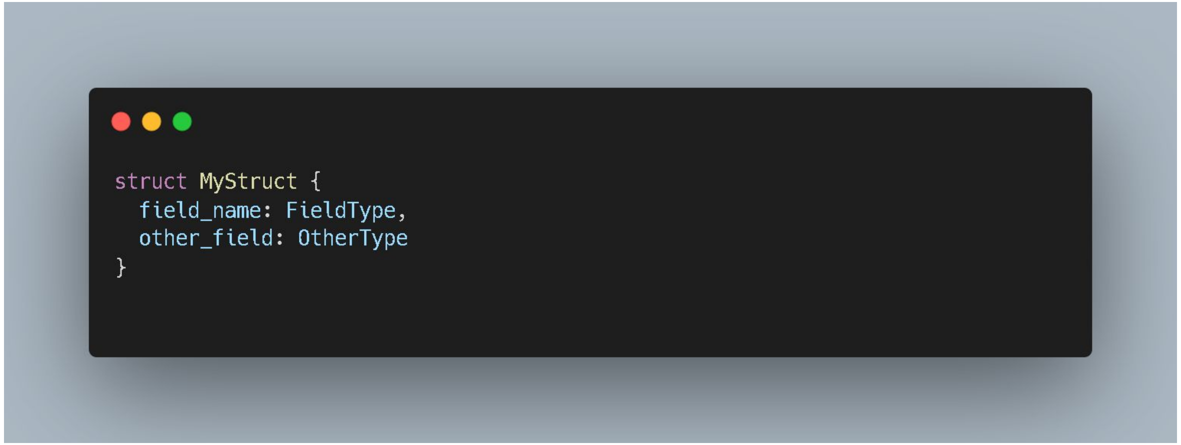
https://doc.rust-lang.org/book/ch07-02-defining-modules-to-control-scope-and-privacy.html

Structs

You've actually been exposed to structs a couple of times by now. They're very similar to tuples, but allow for some more flexibility.

At their heart, structs act as any other data structure, in the future we'll also explore how structs can act more like classes (and have associated functions, behaviors, etc)

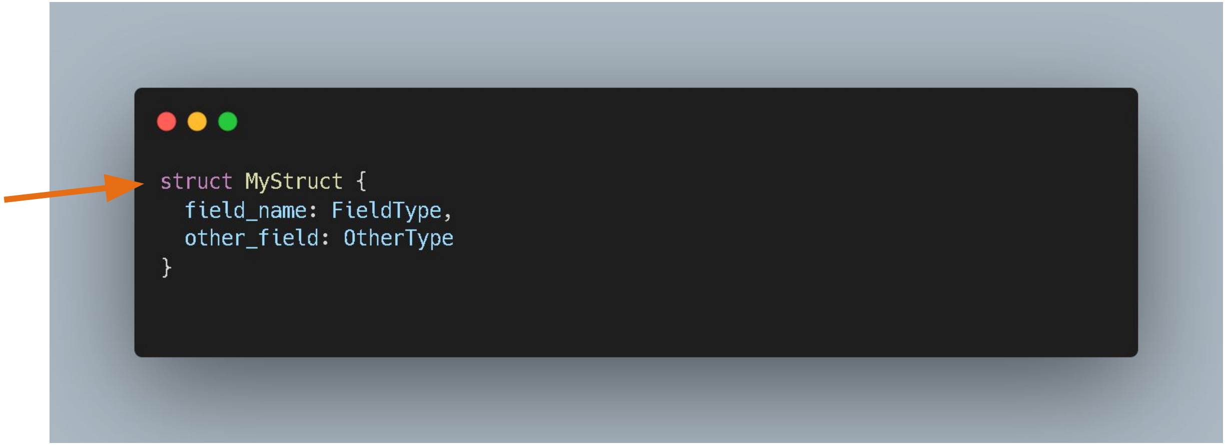
Structs



```
struct MyStruct {
  field_name: FieldType,
  other_field: OtherType
}
```

Structs

Structs are defined with the **struct** keyword.

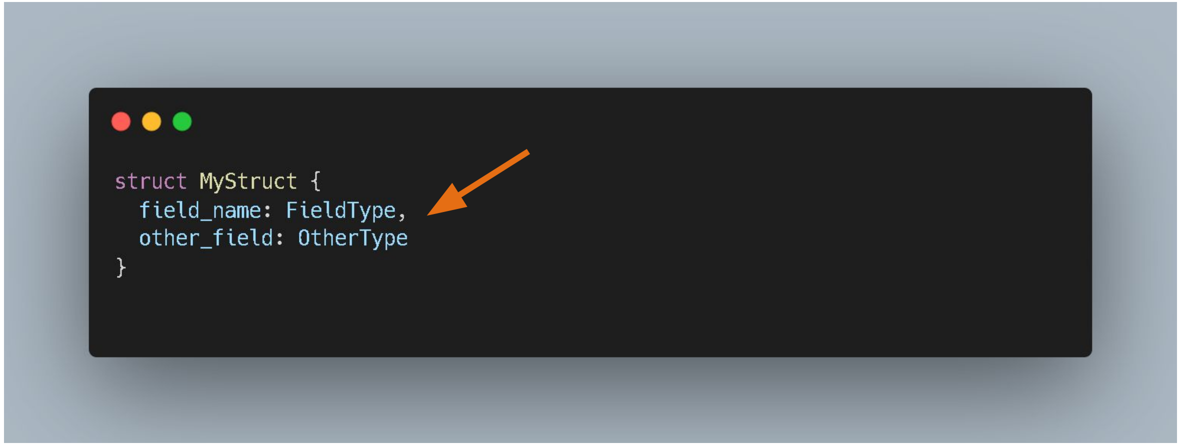


```
struct MyStruct {
  field_name: FieldType,
  other_field: OtherType
}
```

Structs

Structs are defined with the **struct** keyword.

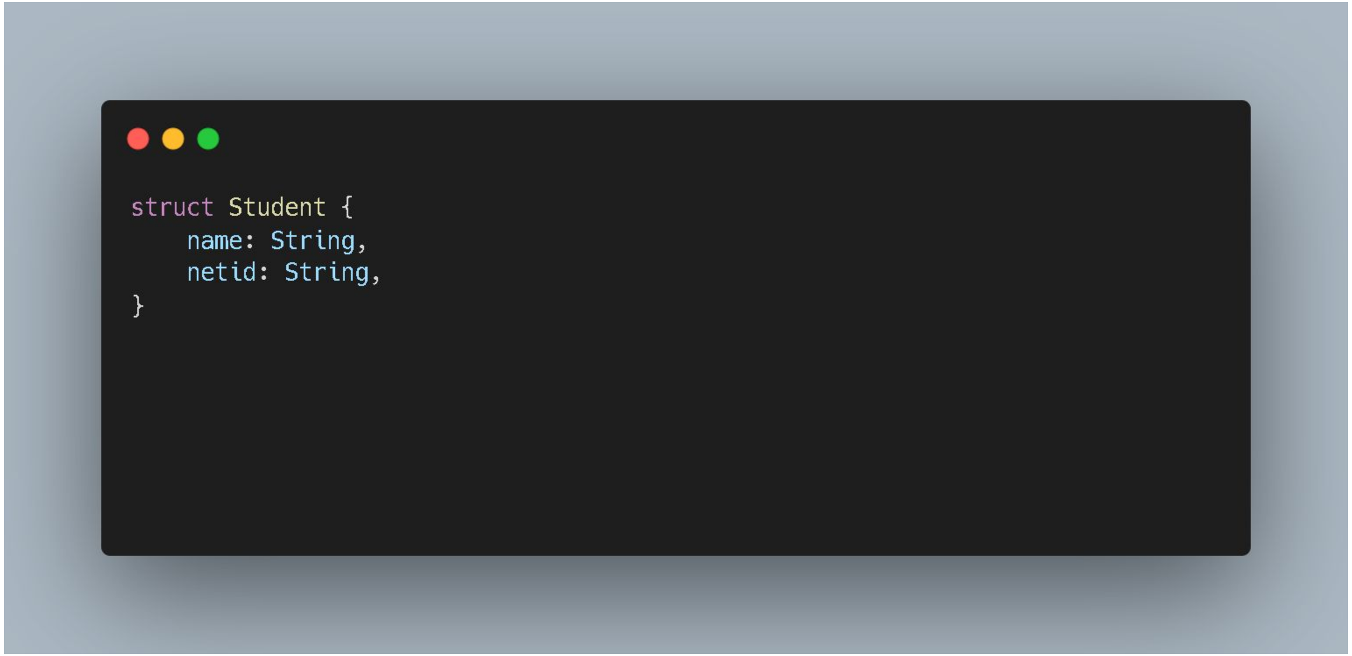
Inside of a struct, we can define certain struct **fields** and their data types.



```
struct MyStruct {
  field_name: FieldType,
  other_field: OtherType
}
```

Structs

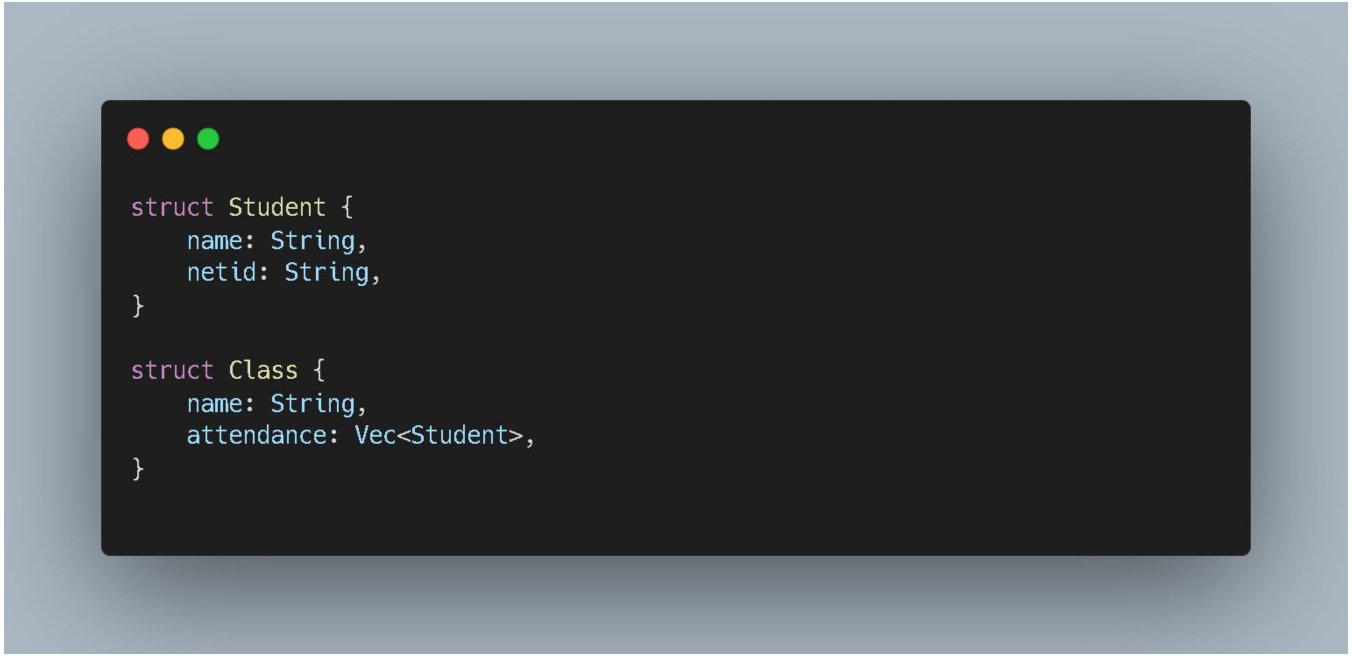
Here’s a real example.



```
struct Student {
    name: String,
    netid: String,
}
```

Structs

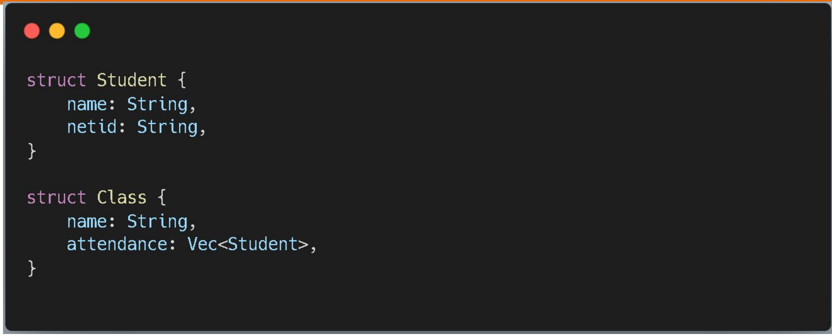
Struct **fields** can contain other **structs**, or more complex data types, or both!



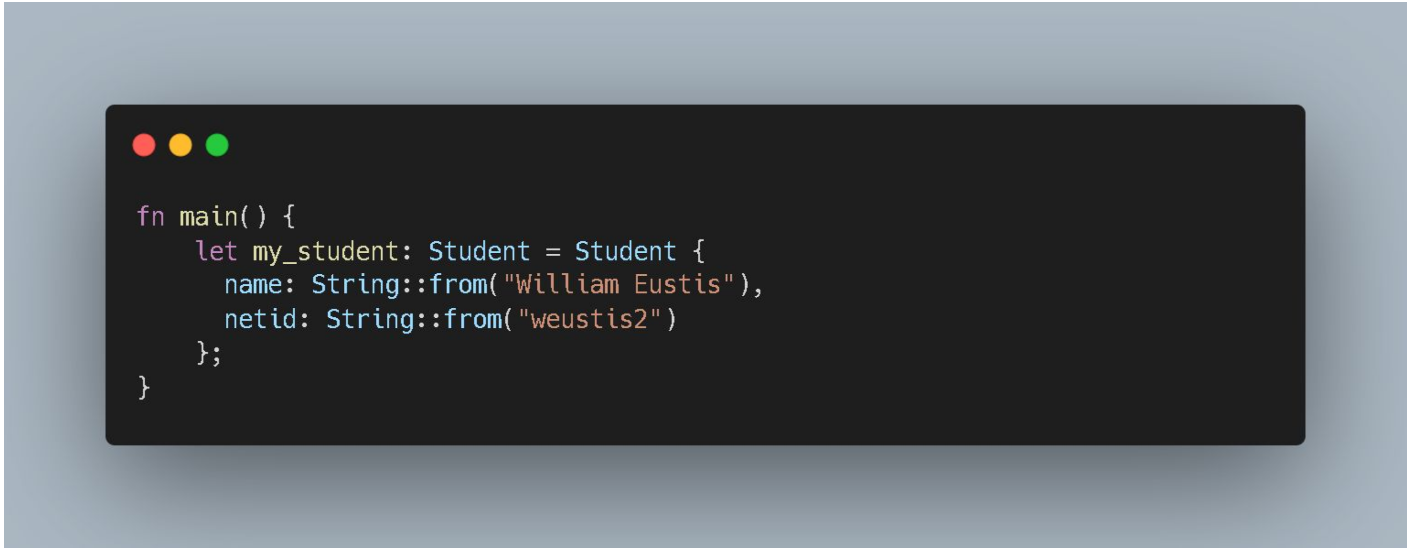
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```

Structs

We can **instantiate** our struct like so:



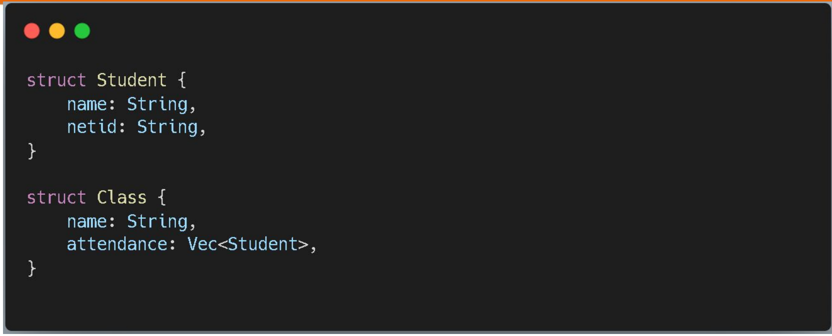
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



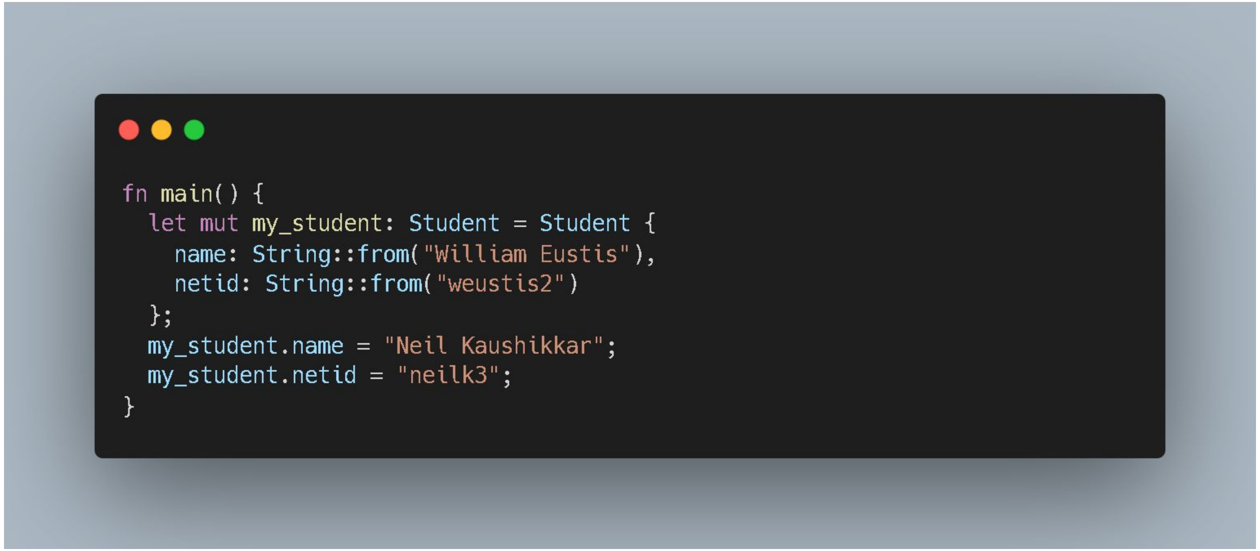
```
fn main() {
    let my_student: Student = Student {
      name: String::from("William Eustis"),
      netid: String::from("weustis2")
    };
}
```

Structs

Structs can also be **mutable.**



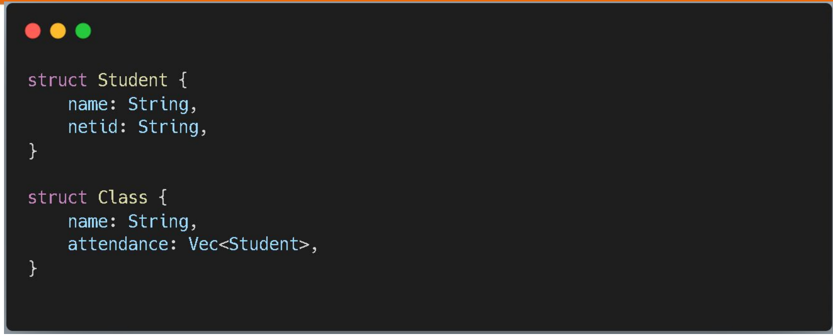
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



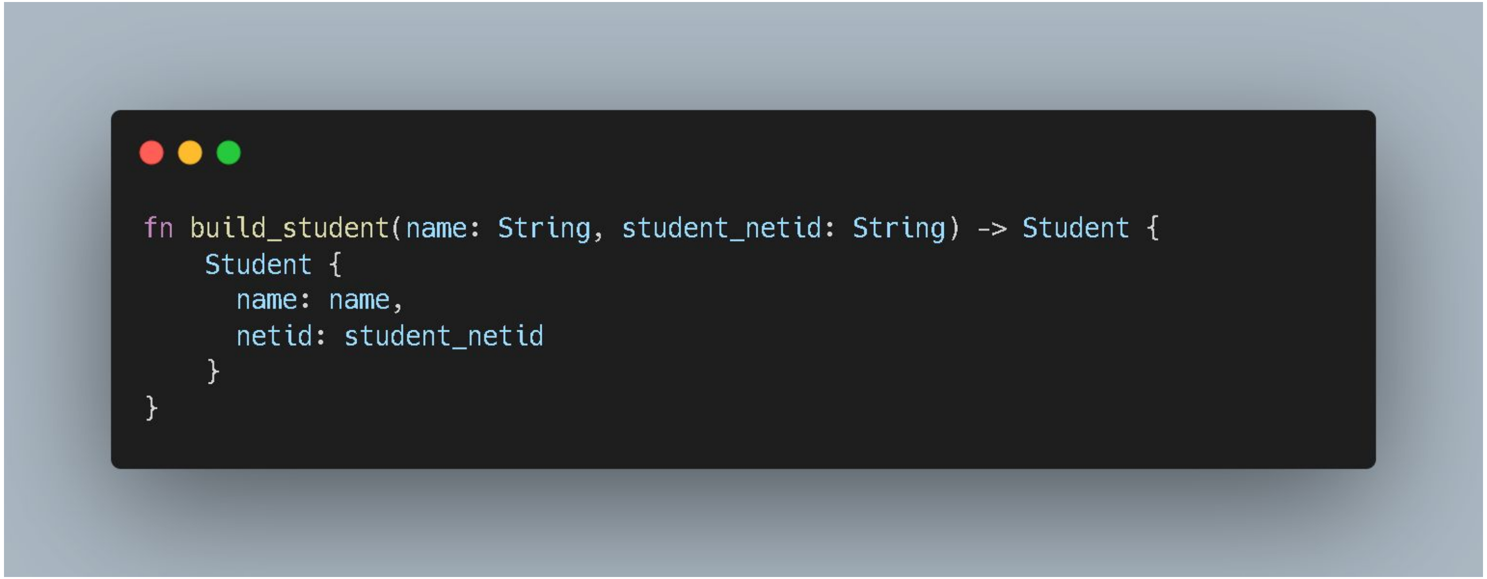
```
fn main() {
  let mut my_student: Student = Student {
    name: String::from("William Eustis"),
    netid: String::from("weustis2")
  };
  my_student.name = "Neil Kaushikkar";
  my_student.netid = "neilk3";
}
```

Structs

We can use functions to simplify things...



```rust
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```

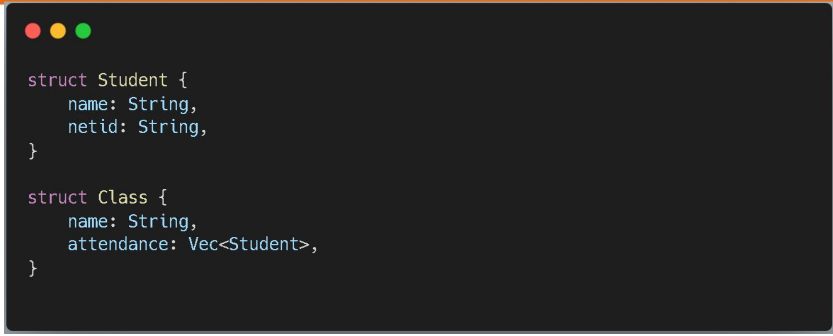


```rust
fn build_student(name: String, student_netid: String) -> Student {
    Student {
      name: name,
      netid: student_netid
    }
}
```

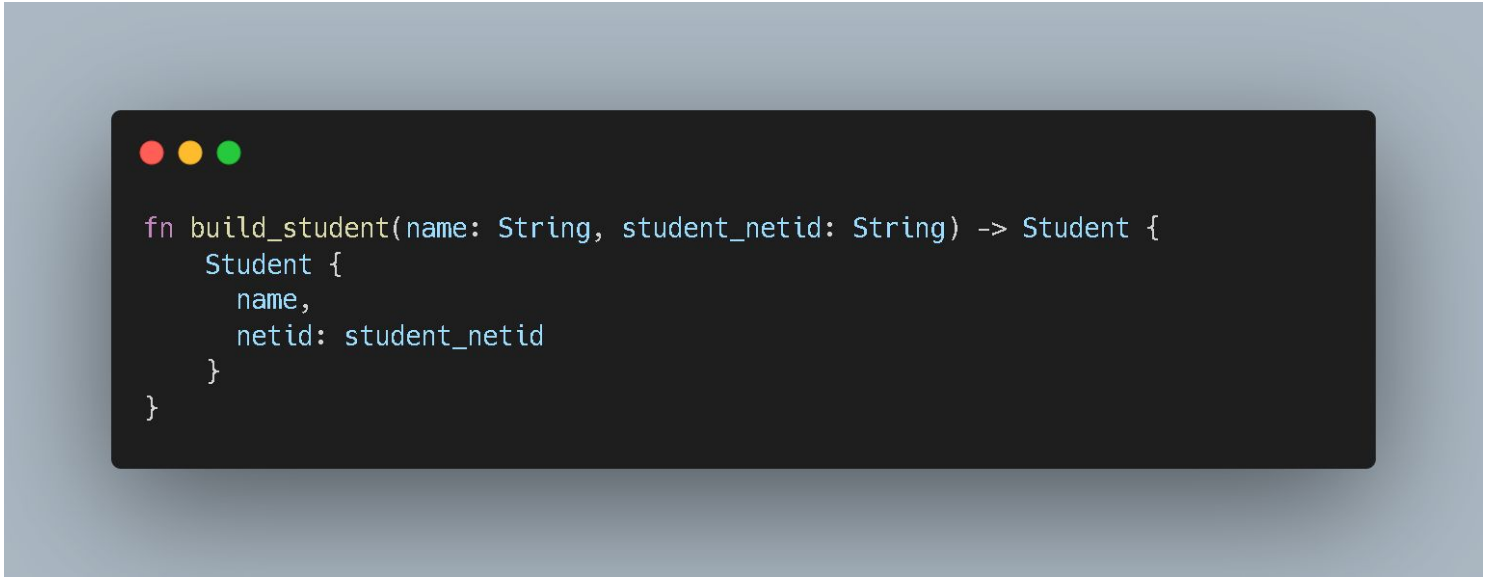
Structs

This can be simplified!

If the variable name and field name are the same, we only need to specify the field name.



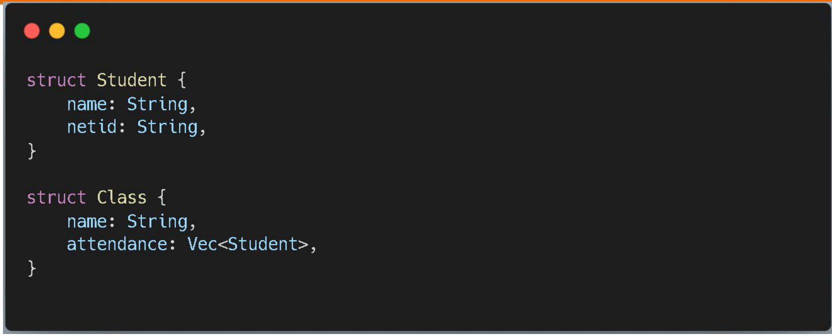
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



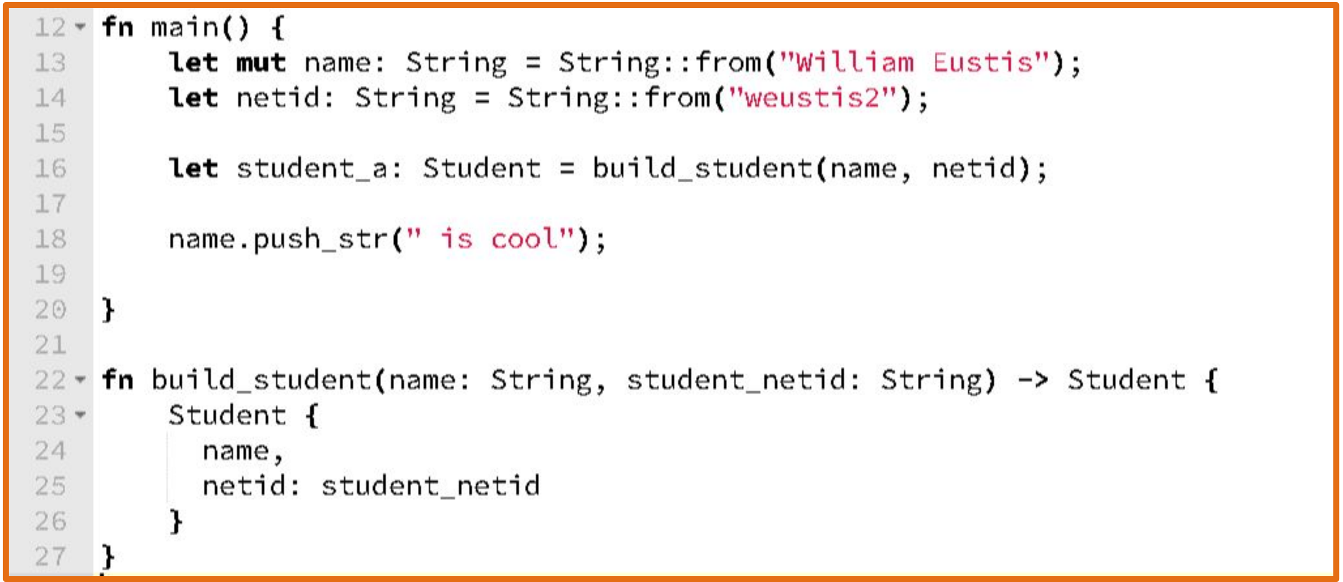
```
fn build_student(name: String, student_netid: String) -> Student {
    Student {
      name,
      netid: student_netid
    }
}
```

Structs

What's happening here with ownership?



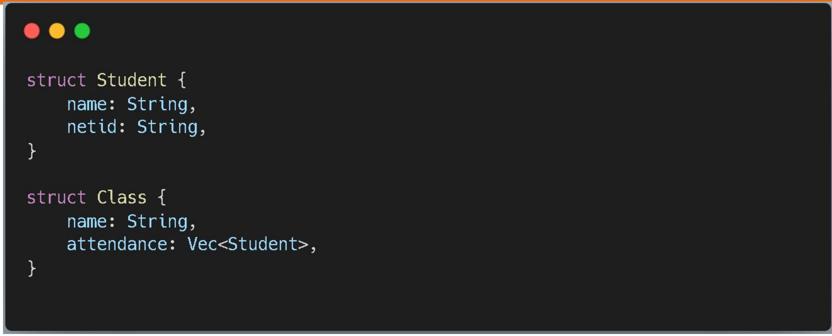
```rust
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



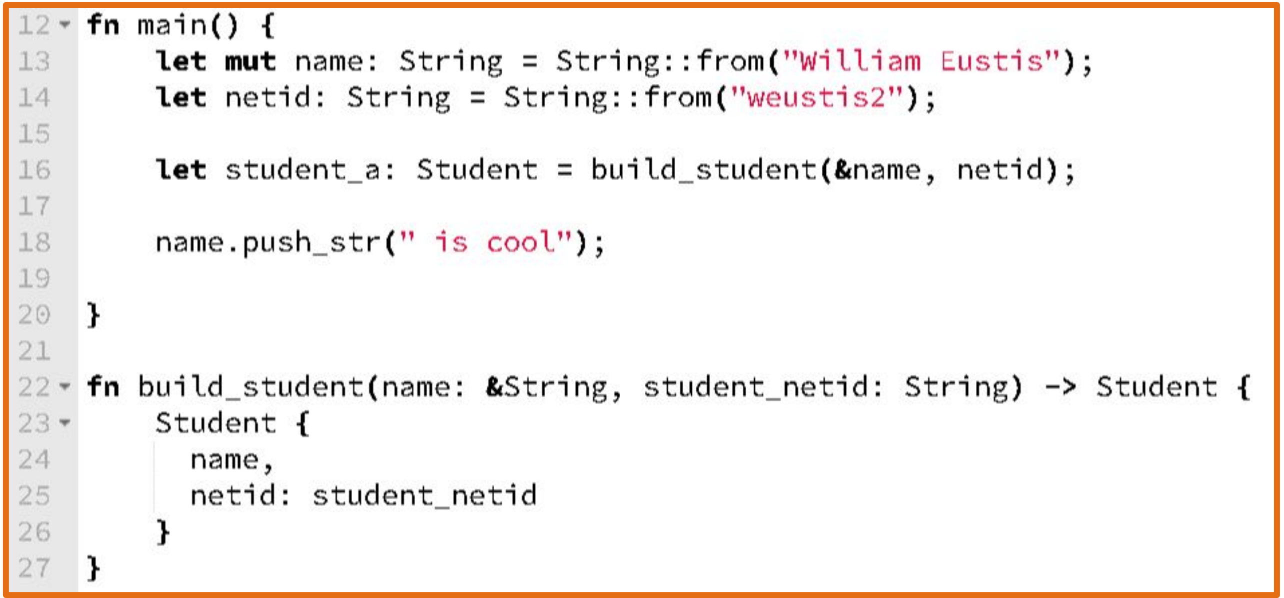
```rust
fn main() {
    let mut name: String = String::from("William Eustis");
    let netid: String = String::from("weustis2");

    let student_a: Student = build_student(name, netid);

    name.push_str(" is cool");

}

fn build_student(name: String, student_netid: String) -> Student {
    Student {
        name,
        netid: student_netid
    }
}
```

Structs

What’s happening here with ownership?



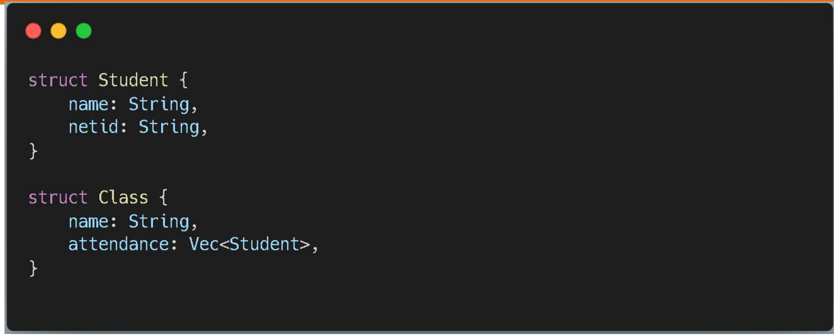
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



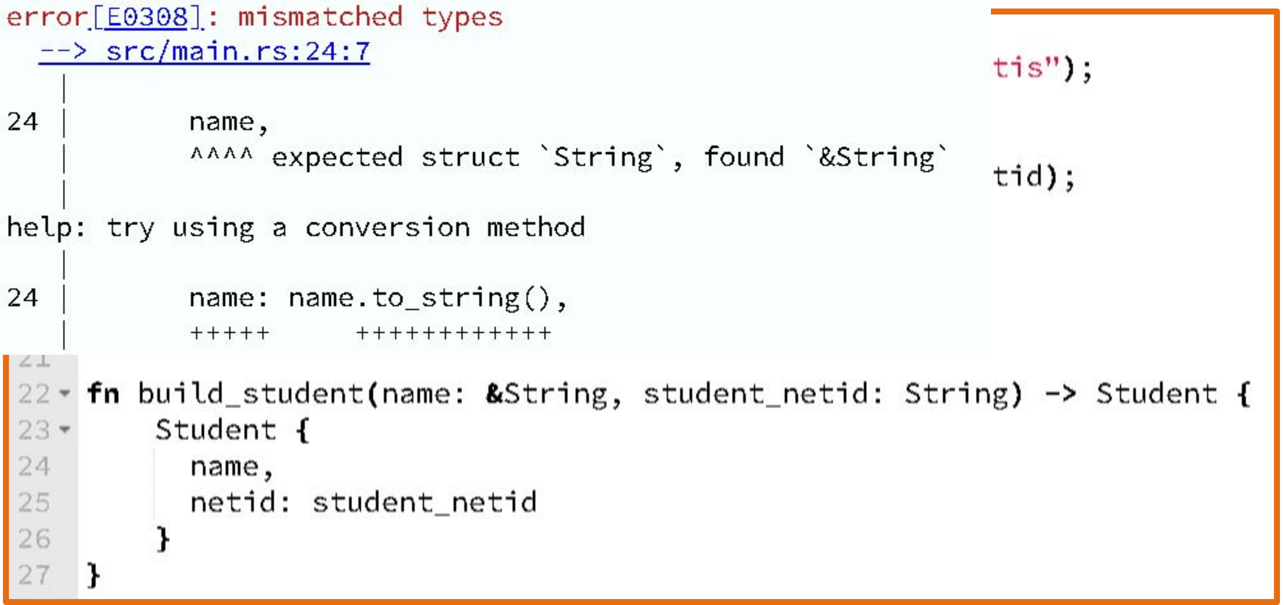
```
fn main() {
    let mut name: String = String::from("William Eustis");
    let netid: String = String::from("weustis2");

    let student_a: Student = build_student(&name, netid);

    name.push_str(" is cool");

}

fn build_student(name: &String, student_netid: String) -> Student {
    Student {
        name,
        netid: student_netid
    }
}
```

Structs

What's happening here with ownership?



```rust
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



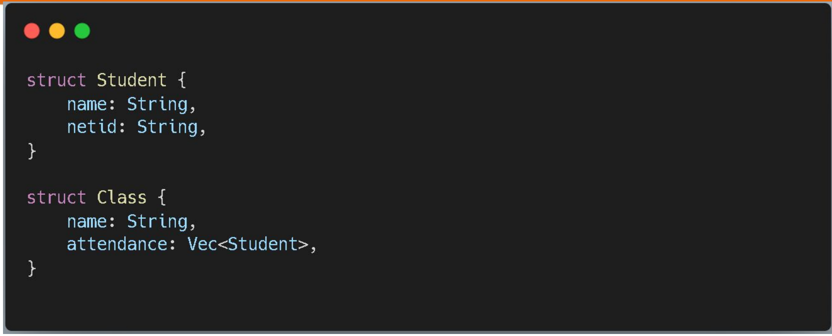
```
error[E0308]: mismatched types
  --> src/main.rs:24:7
   |
24 |         name,
   |         ^^^^ expected struct `String`, found `&String`
   |
help: try using a conversion method
   |
24 |         name: name.to_string(),
   |         +++++      ++++++++++++
```

```
tis");

tid);
```

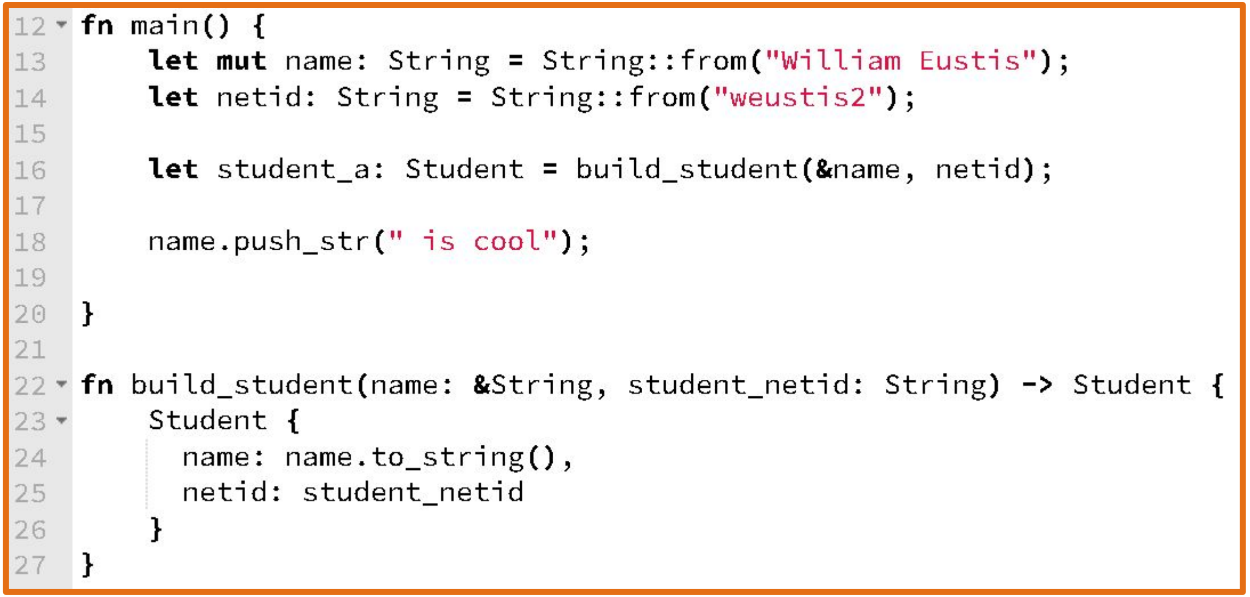
```rust
fn build_student(name: &String, student_netid: String) -> Student {
    Student {
      name,
      netid: student_netid
    }
}
```

Structs

What’s happening here with ownership?



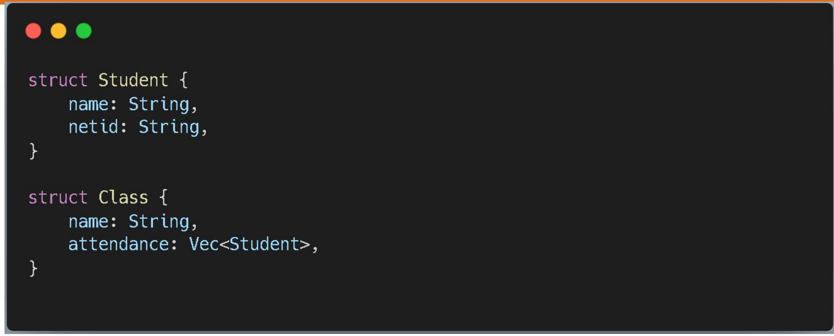
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



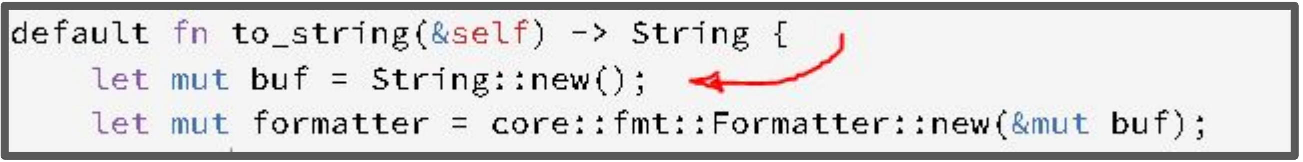
```
fn main() {
    let mut name: String = String::from("William Eustis");
    let netid: String = String::from("weustis2");

    let student_a: Student = build_student(&name, netid);

    name.push_str(" is cool");

}

fn build_student(name: &String, student_netid: String) -> Student {
    Student {
      name: name.to_string(),
      netid: student_netid
    }
}
```

Structs

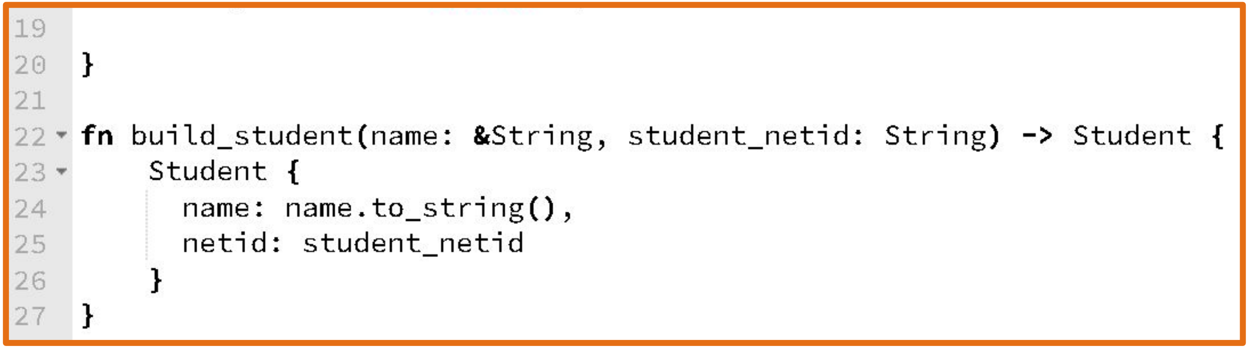
What's happening here with ownership?



```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```



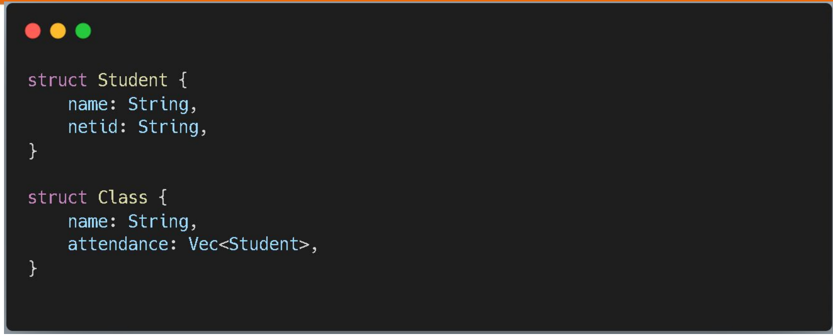
```
default fn to_string(&self) -> String {
    let mut buf = String::new();
    let mut formatter = core::fmt::Formatter::new(&mut buf);
```



```

}

fn build_student(name: &String, student_netid: String) -> Student {
    Student {
      name: name.to_string(),
      netid: student_netid
    }
}
```

Structs

What’s happening here with ownership?



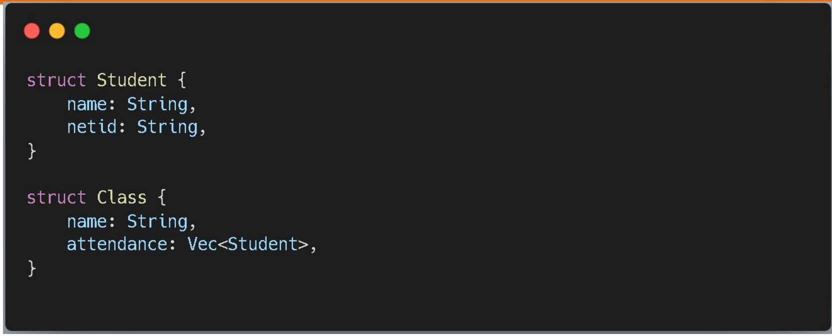
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```

If we want to maintain ownership, we’ll need to change the struct to take a reference to a String. (&String)

But to do this, we must define the **lifetime** of the data. We may cover this in the special topics lectures during the last couple of weeks.

Structs

What's happening here with ownership?



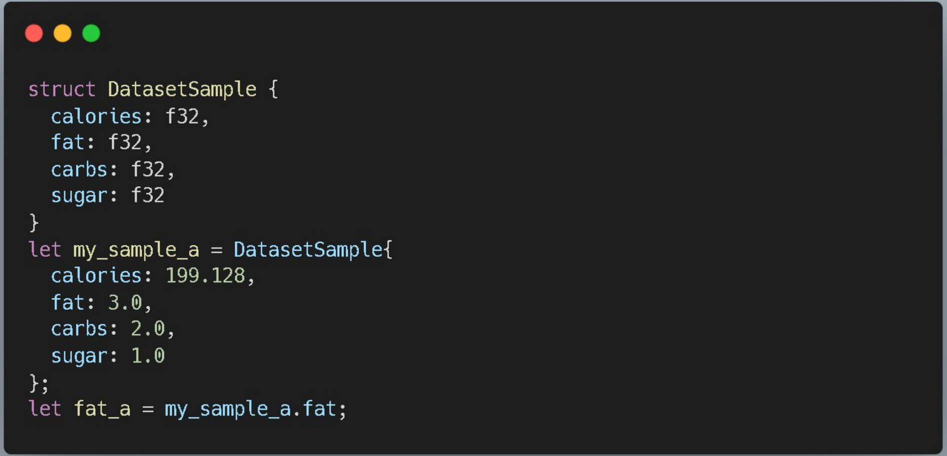
```
struct Student {
    name: String,
    netid: String,
}

struct Class {
    name: String,
    attendance: Vec<Student>,
}
```

If we want to maintain ownership, we'll need to change the struct to take a reference to a String. (&String)

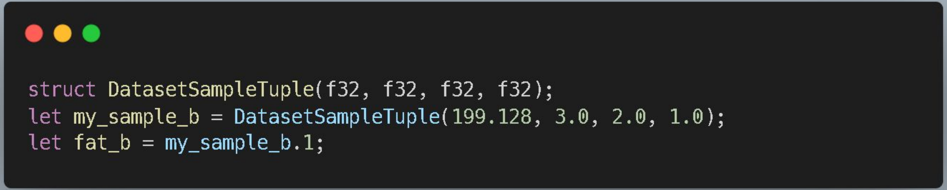
But to do this, we must define the **lifetime** of the data. We may cover this in the special topics lectures during the last couple of weeks.

Special Structs

Tuple structs allow us to not have to name the **fields**.



```rust
struct DatasetSample {
  calories: f32,
  fat: f32,
  carbs: f32,
  sugar: f32
}
let my_sample_a = DatasetSample{
  calories: 199.128,
  fat: 3.0,
  carbs: 2.0,
  sugar: 1.0
};
let fat_a = my_sample_a.fat;
```



```rust
struct DatasetSampleTuple(f32, f32, f32, f32);
let my_sample_b = DatasetSampleTuple(199.128, 3.0, 2.0, 1.0);
let fat_b = my_sample_b.1;
```

That’s All Folks!

:^)