Lecture 11

Traits

Goals For Today

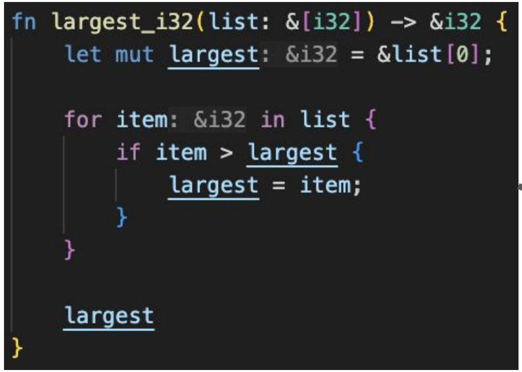
- Generics review
- Traits

Generics Review

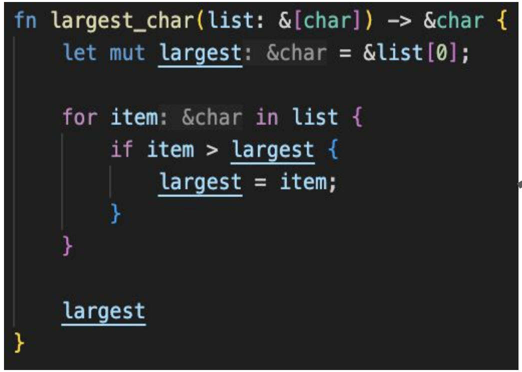
- Abstract stand-ins for concrete types or other properties
- Can be used in the definitions of
  - Structs: Vec<T>
  - Enums: Option<T>, Result<T>
  - Methods
  - Functions

Generics Review (cont)

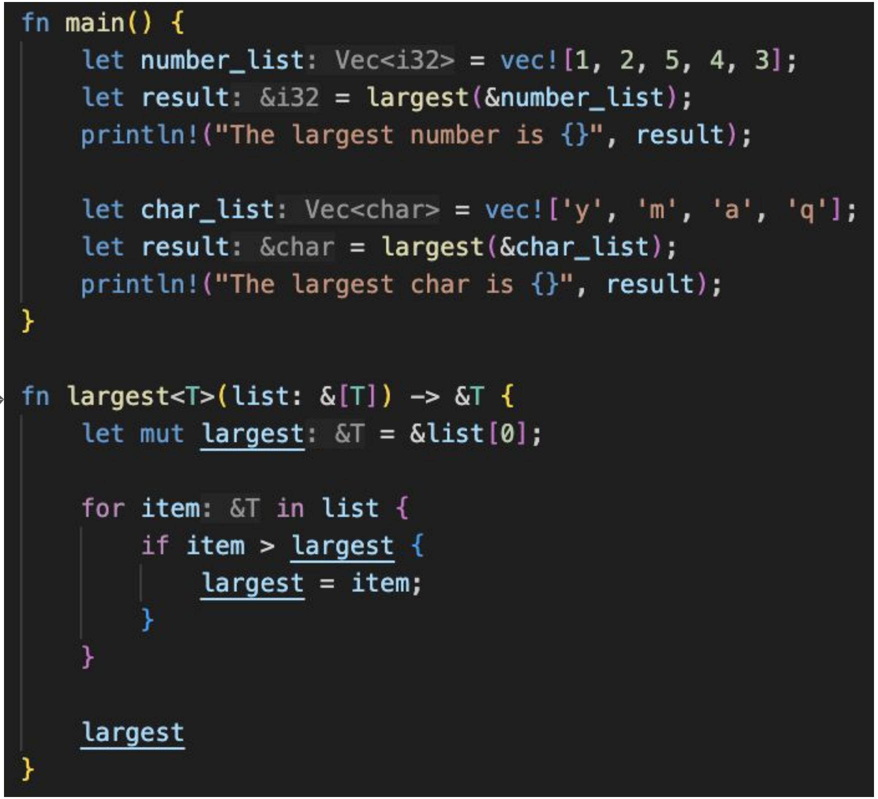
- We can use generics in functions to remove boilerplate:



```rust
fn largest_i32(list: &[i32]) -> &i32 {
    let mut largest: &i32 = &list[0];

    for item: &i32 in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```



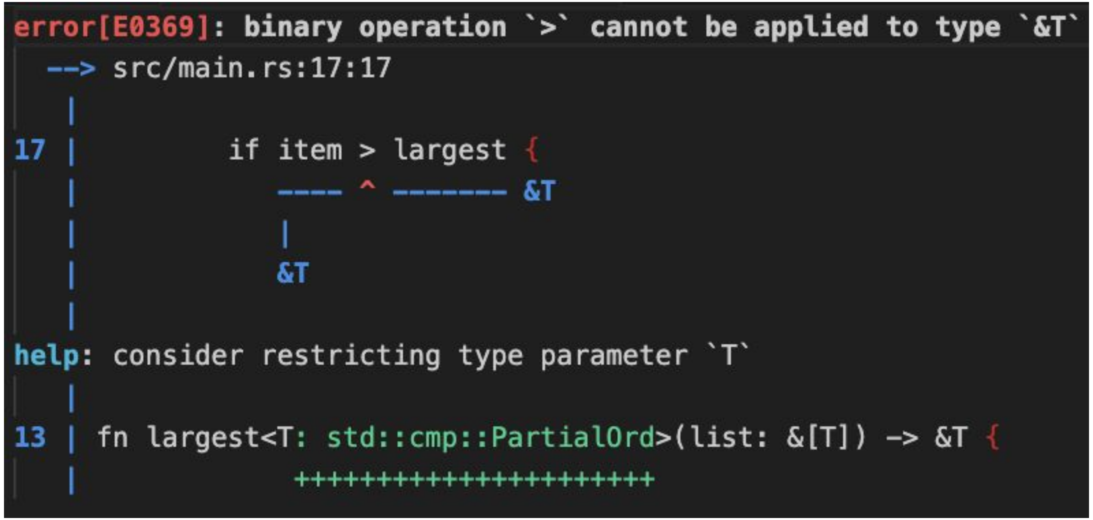
```rust
fn largest_char(list: &[char]) -> &char {
    let mut largest: &char = &list[0];

    for item: &char in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```



```rust
fn main() {
    let number_list: Vec<i32> = vec![1, 2, 5, 4, 3];
    let result: &i32 = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list: Vec<char> = vec!['y', 'm', 'a', 'q'];
    let result: &char = largest(&char_list);
    println!("The largest char is {}", result);
}

fn largest<T>(list: &[T]) -> &T {
    let mut largest: &T = &list[0];

    for item: &T in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Generics Review (cont)

- But it looks like Rust is unhappy?

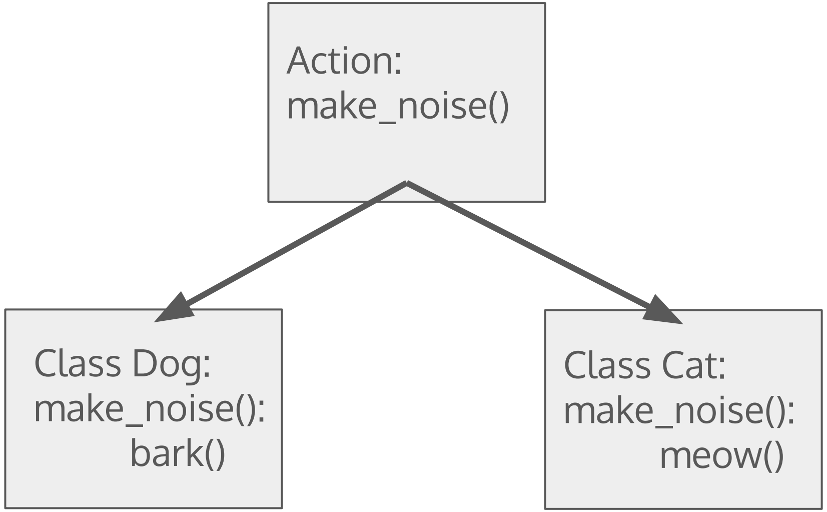


```
error[E0369]: binary operation `>` cannot be applied to type `&T`
  --> src/main.rs:17:17
   |
17 |         if item > largest {
   |            ---- ^ ------- &T
   |            |
   |            &T
   |
help: consider restricting type parameter `T`
   |
13 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
   |             ++++++++++++++++++++++
```

- What if T is HashMap<i32, i32>?
- The concrete type the user inputs into the function might not be comparable

Generics in Functions (cont)

- Needs to specify that our generic type must have a comparison behavior
- Needs to be able to also define abstract actions in addition to our abstract types
- Potential solution: Interfaces from OO languages (e.g: C++, Java)
  - Specify shared actions that classes can define concrete behaviors for
  - Do not store data
- Enter Traits



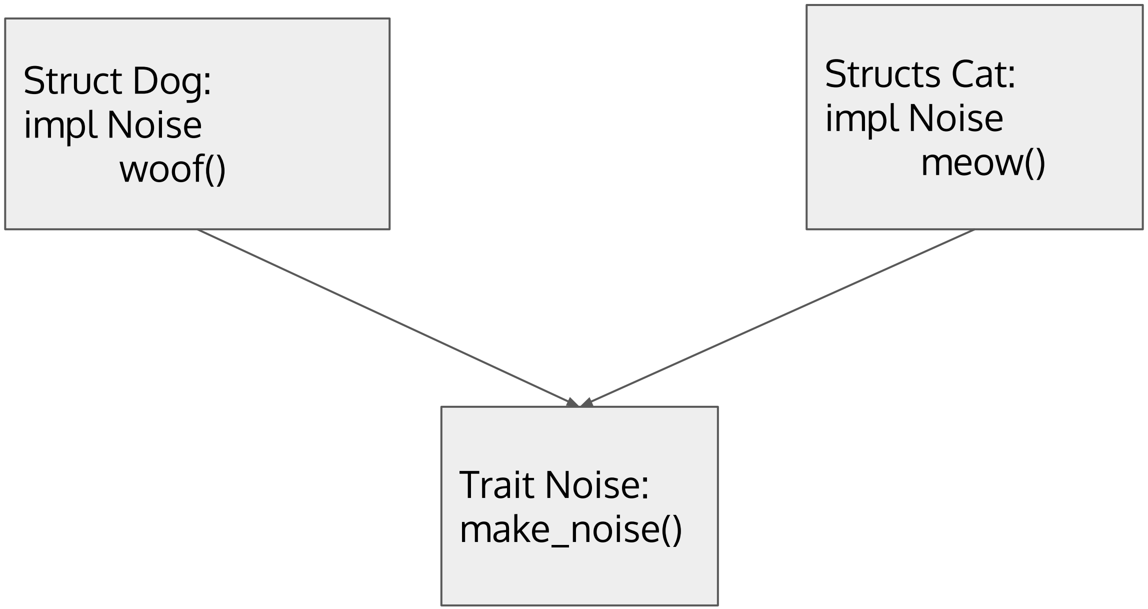
Action:
make_noise()

Class Dog:
make_noise():
bark()

Class Cat:
make_noise():
meow()

Traits

- Flip inheritance upside down
- Structs choose which traits to inherit
- Can filter to accept only structs that have traits



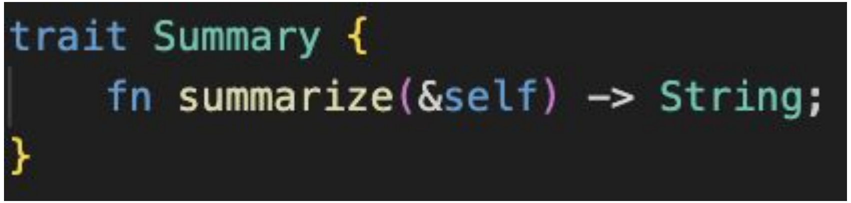
Struct Dog:
impl Noise
woof()

Structs Cat:
impl Noise
meow()

Trait Noise:
make_noise()

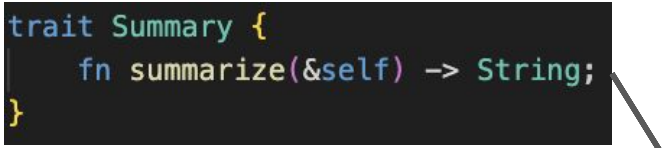
Traits

- Define shared functionality that types can have
- Types have this shared functionality when they implement the trait
- Trait definition: name of trait + set of functions that types can implement

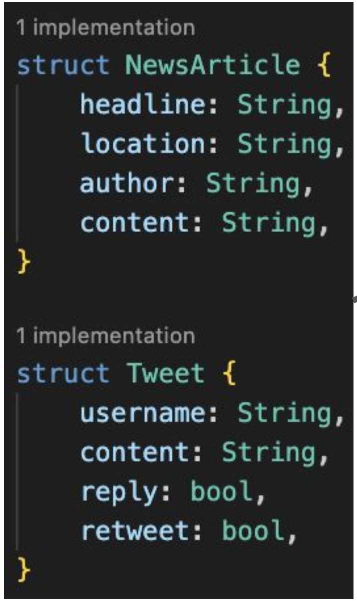


```
trait Summary {
    fn summarize(&self) -> String;
}
```

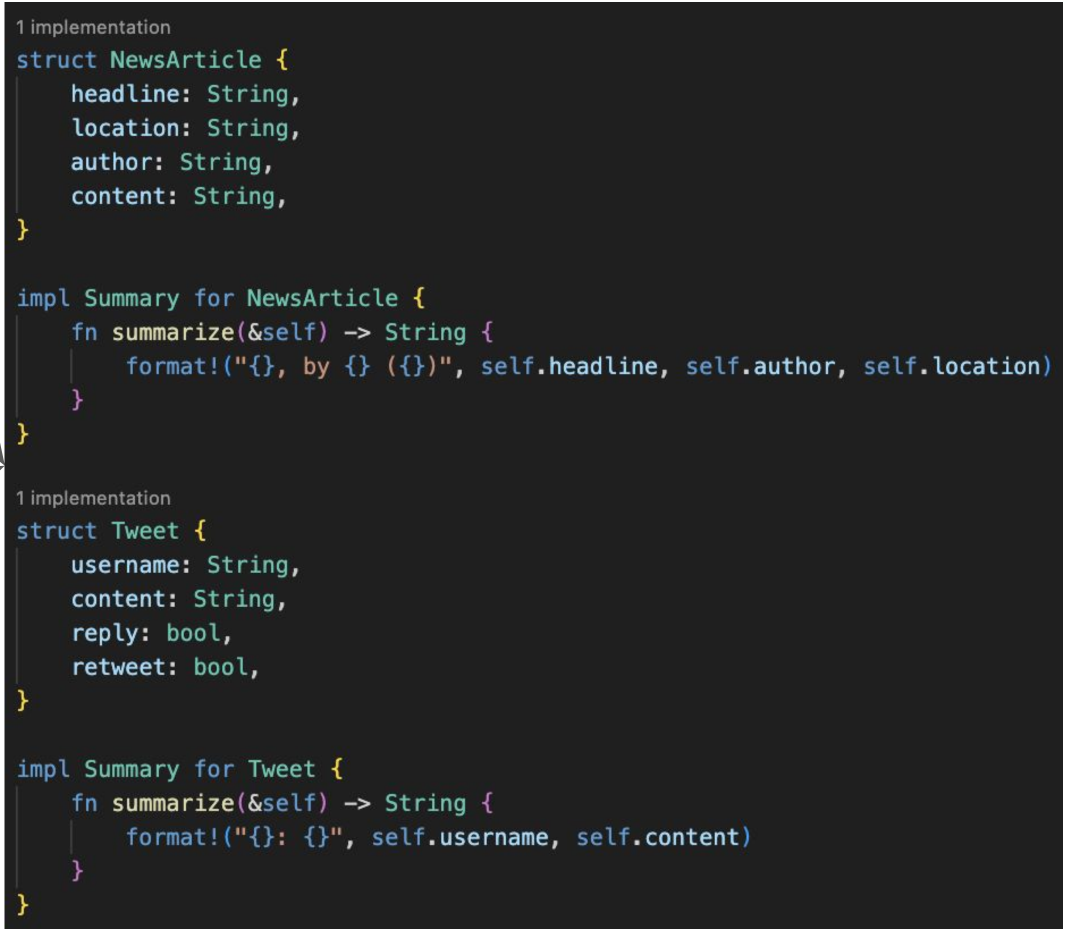
Implementing Traits



```rust
trait Summary {
    fn summarize(&self) -> String;
}
```



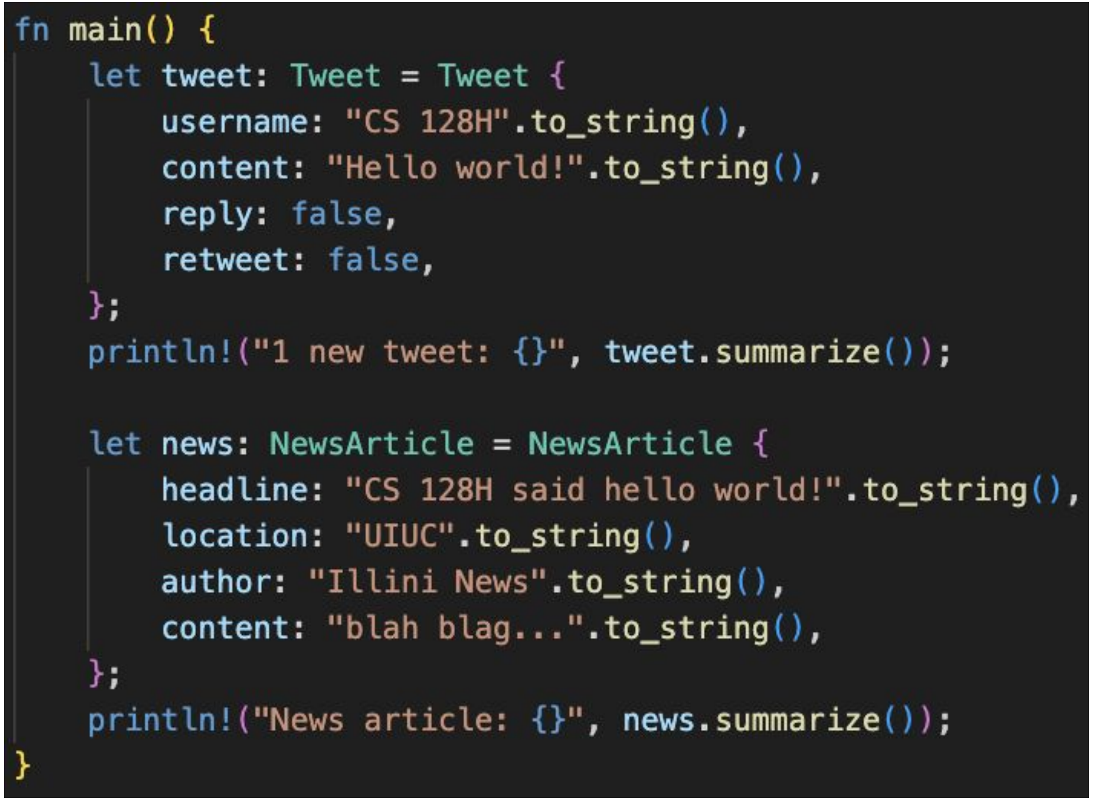
```rust
1 implementation
struct NewsArticle {
    headline: String,
    location: String,
    author: String,
    content: String,
}

1 implementation
struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}
```



```rust
1 implementation
struct NewsArticle {
    headline: String,
    location: String,
    author: String,
    content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

1 implementation
struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

Implementing Traits (cont)

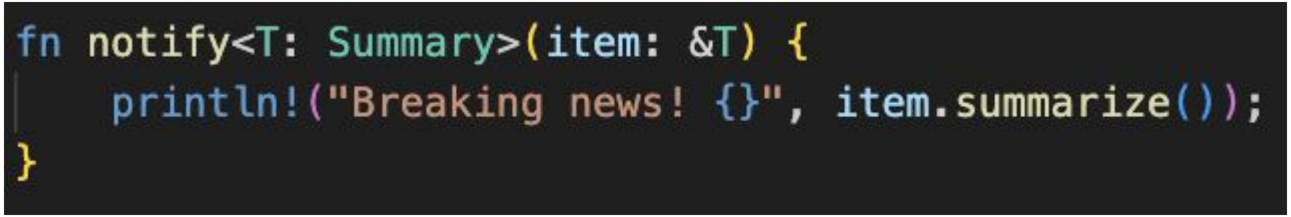
- Now we can summarize news articles and tweets more conveniently!



```rust
fn main() {
    let tweet: Tweet = Tweet {
        username: "CS 128H".to_string(),
        content: "Hello world!".to_string(),
        reply: false,
        retweet: false,
    };
    println!("1 new tweet: {}", tweet.summarize());

    let news: NewsArticle = NewsArticle {
        headline: "CS 128H said hello world!".to_string(),
        location: "UIUC".to_string(),
        author: "Illini News".to_string(),
        content: "blah blag...".to_string(),
    };
    println!("News article: {}", news.summarize());
}
```

Traits and Generics

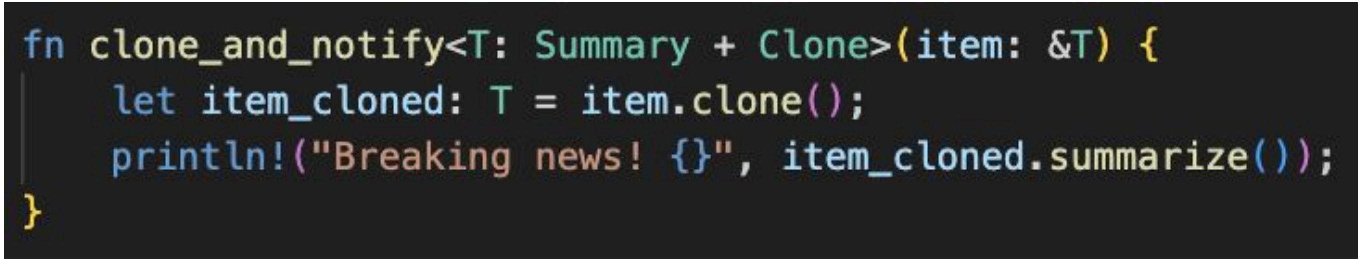
- We can specify trait bounds on generic types
- A trait bound means the generic type must implement a particular trait (has a particular behavior)
- Here, the type of the input to notify() must implement Summary
  - Behavior: the item can be summarized



```rust
fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

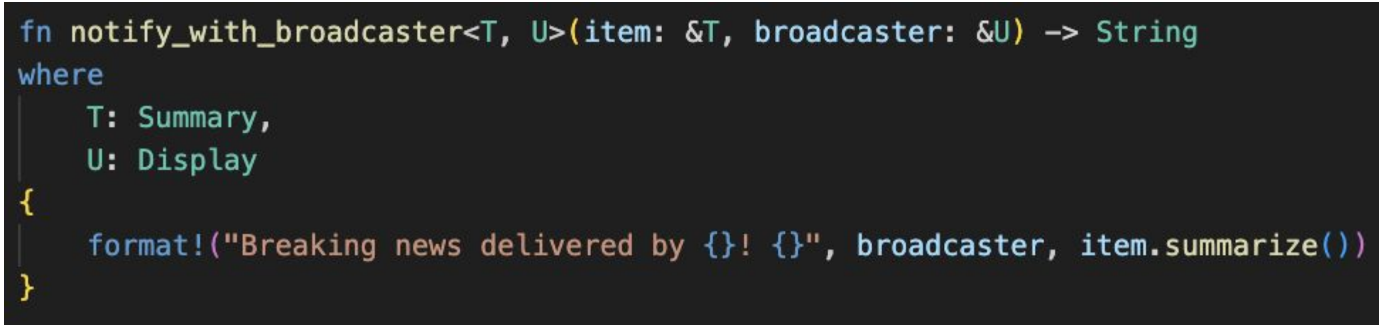
Traits and Generics (cont)

- Can specify multiple trait bounds using the + syntax



```rust
fn clone_and_notify<T: Summary + Clone>(item: &T) {
    let item_cloned: T = item.clone();
    println!("Breaking news! {}", item_cloned.summarize());
}
```

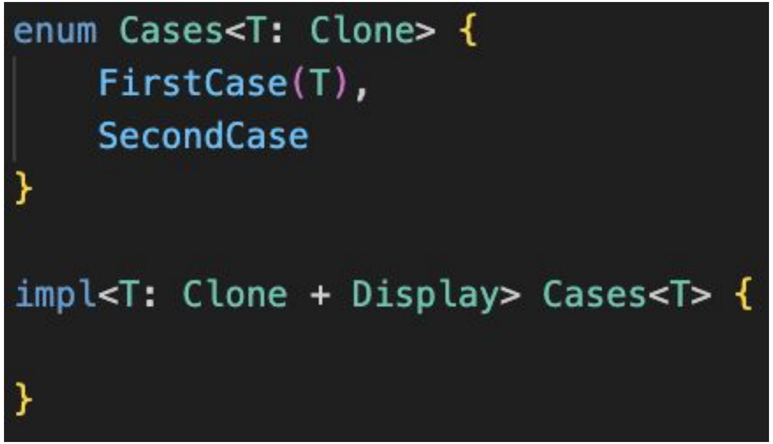
- Can also define trait bounds using a where clause



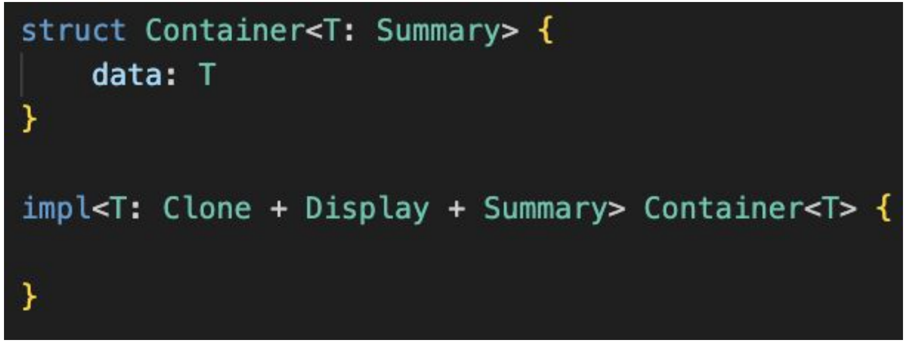
```rust
fn notify_with_broadcaster<T, U>(item: &T, broadcaster: &U) -> String
where
    T: Summary,
    U: Display
{
    format!("Breaking news delivered by {}! {}", broadcaster, item.summarize())
}
```

Traits and Generics (cont)

- Trait bounds can be specified for generic types anywhere they appear
- Note: If a generic type has a trait bound in the struct/enum definition, the implement block must specify the same trait bound for said generic type



```
enum Cases<T: Clone> {
    FirstCase(T),
    SecondCase
}

impl<T: Clone + Display> Cases<T> {

}
```

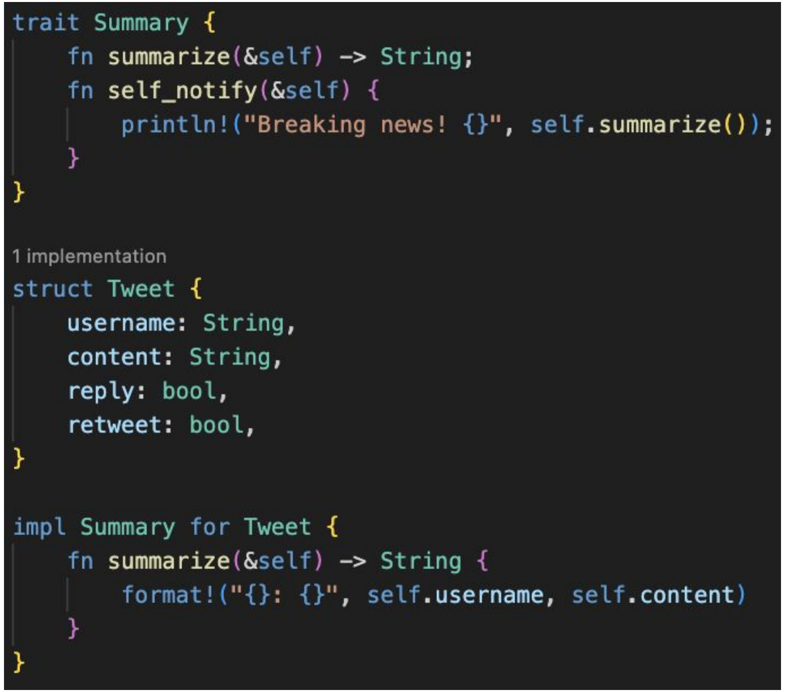


```
struct Container<T: Summary> {
    data: T
}

impl<T: Clone + Display + Summary> Container<T> {

}
```

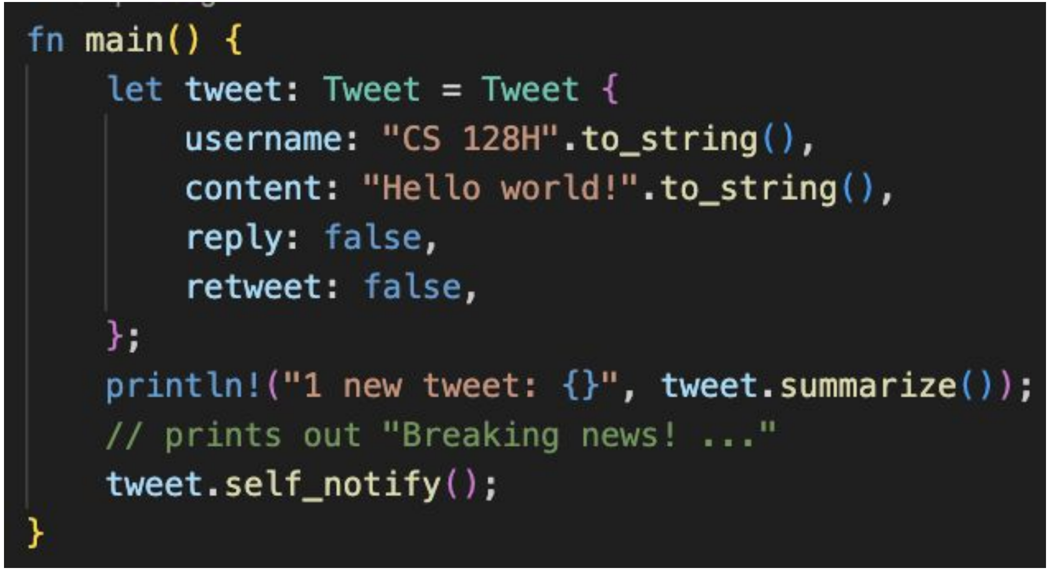
Provided methods in Traits

- Types sometimes don’t have to define every methods in a trait
- These non-required methods are called provided methods

- Implemented in the traits definition, types may or may not re-implement them



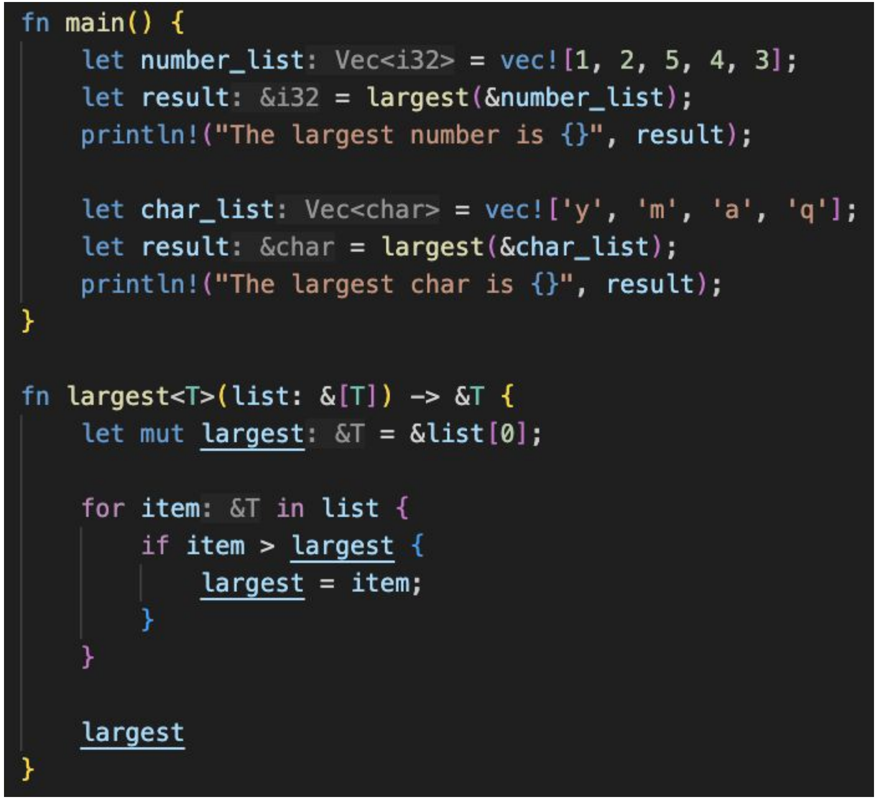
```rust
trait Summary {
    fn summarize(&self) -> String;
    fn self_notify(&self) {
        println!("Breaking news! {}", self.summarize());
    }
}

1 implementation
struct Tweet {
    username: String,
    content: String,
    reply: bool,
    retweet: bool,
}

impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```



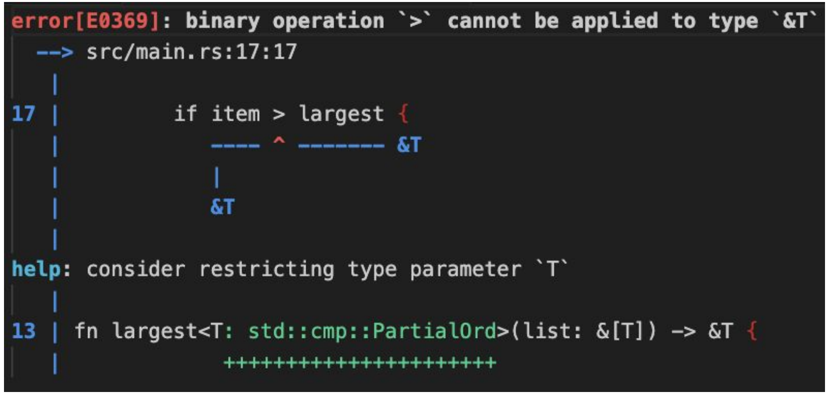
```rust
fn main() {
    let tweet: Tweet = Tweet {
        username: "CS 128H".to_string(),
        content: "Hello world!".to_string(),
        reply: false,
        retweet: false,
    };
    println!("1 new tweet: {}", tweet.summarize());
    // prints out "Breaking news! ..."
    tweet.self_notify();
}
```

The Motivating Example

- Problem: need to specify that T has a comparison behavior
- With Traits, we have a tool for this
- Is there a comparison trait?



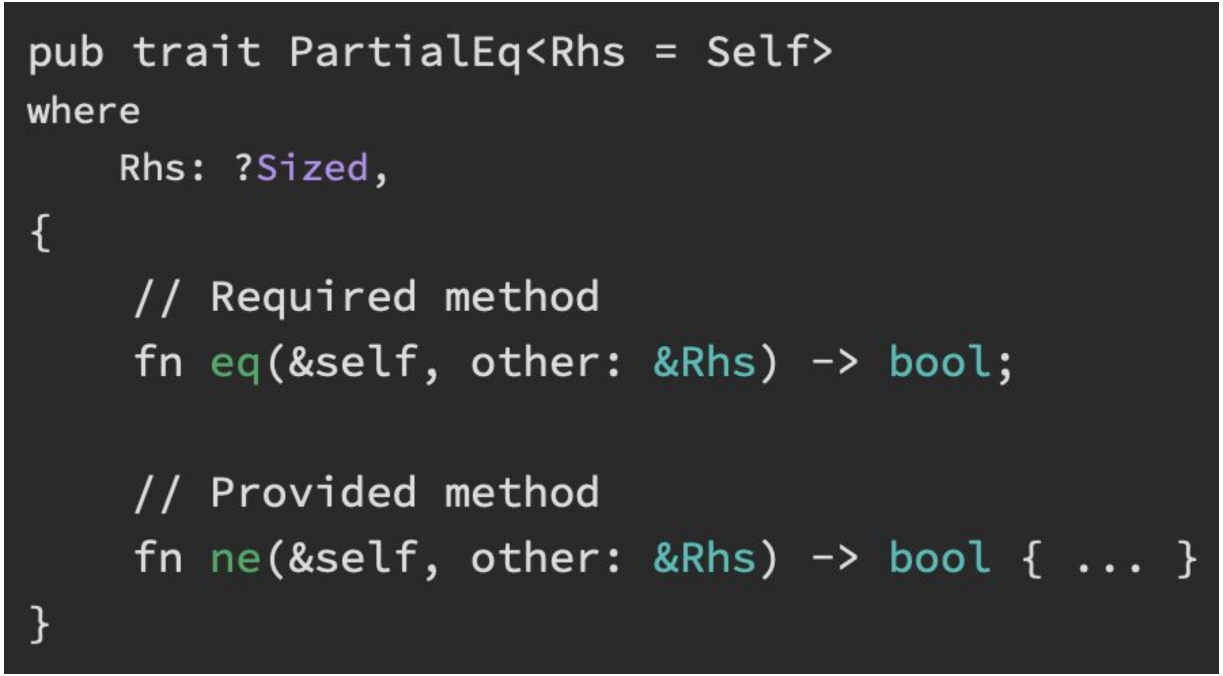
```
fn main() {
    let number_list: Vec<i32> = vec![1, 2, 5, 4, 3];
    let result: &i32 = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list: Vec<char> = vec!['y', 'm', 'a', 'q'];
    let result: &char = largest(&char_list);
    println!("The largest char is {}", result);
}

fn largest<T>(list: &[T]) -> &T {
    let mut largest: &T = &list[0];

    for item: &T in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```



```
error[E0369]: binary operation `>` cannot be applied to type `&T`
  --> src/main.rs:17:17
   |
17 |         if item > largest {
   |            ---- ^ ------- &T
   |            |
   |            &T
   |
help: consider restricting type parameter `T`
   |
13 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
   |             ++++++++++++++++++++++
```

PartialEq and PartialOrd

- std::cmp::PartialEq:
- Defines the == and != operators
- Type only needs to implement eq(), ne() simply returns !eq()

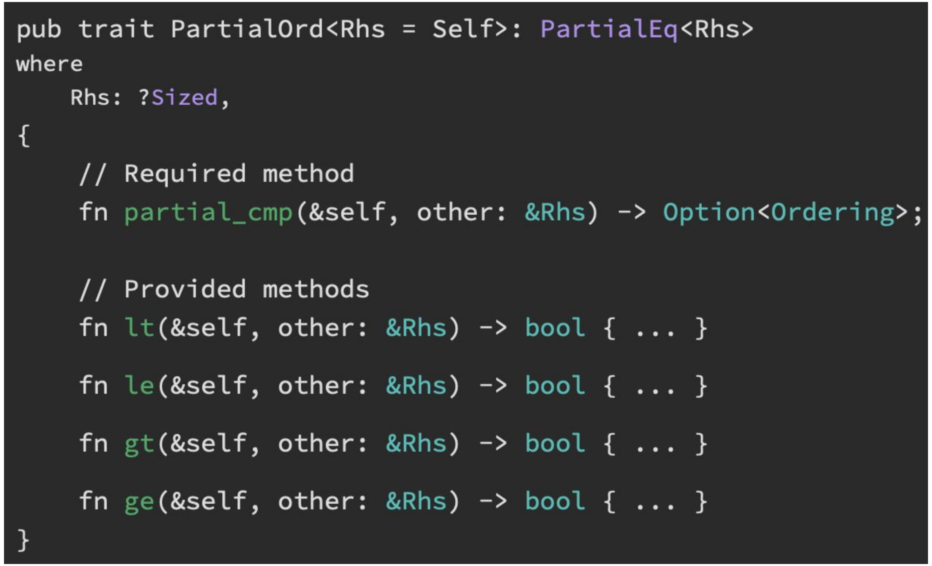


```
pub trait PartialEq<Rhs = Self>
where
    Rhs: ?Sized,
{
    // Required method
    fn eq(&self, other: &Rhs) -> bool;

    // Provided method
    fn ne(&self, other: &Rhs) -> bool { ... }
}
```

Reference: https://doc.rust-lang.org/std/cmp/trait.PartialEq.html

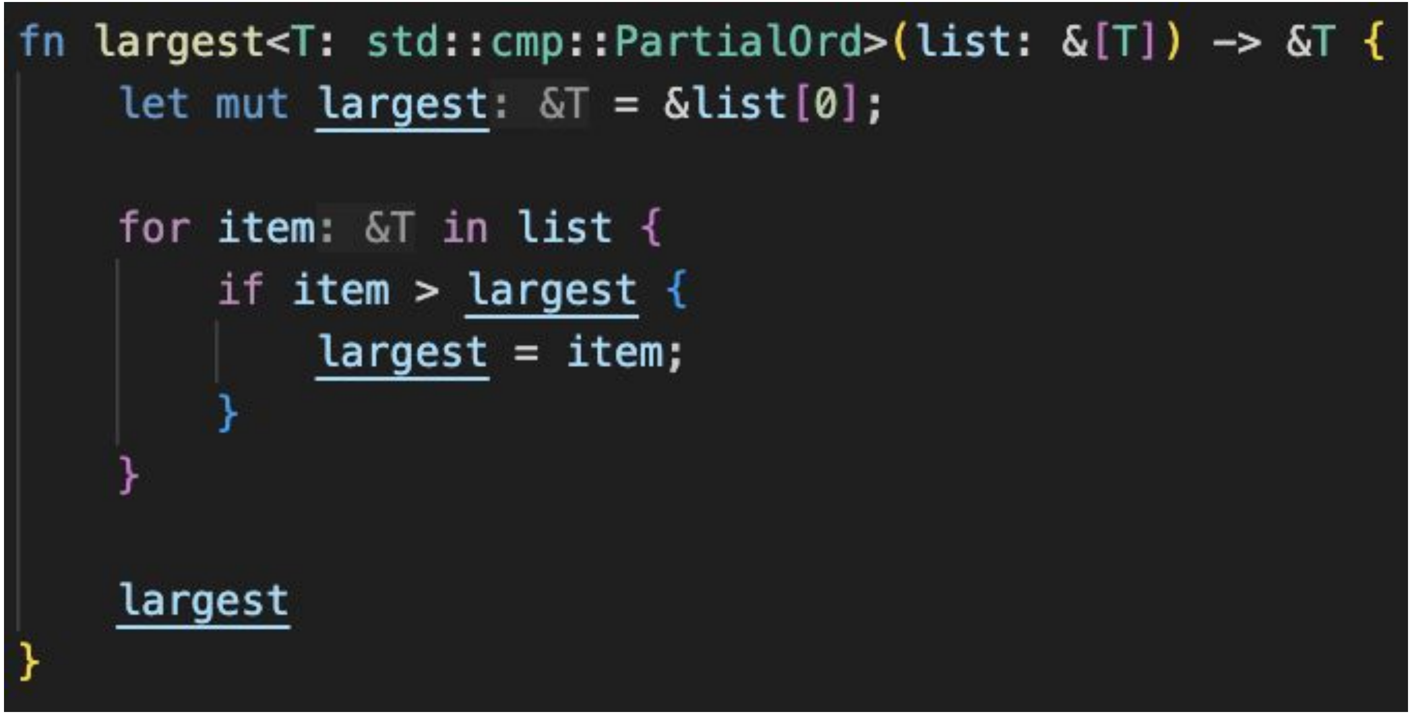
PartialEq and PartialOrd

- std::cmp::PartialOrd:
  - Defines <, <=, >, >=
  - Type only needs to implement partial_cmp(), which defines <, >, and ==
  - Note the interface-like syntax PartialOrd: PartialEq
  - Means types implementing PartialOrd must also implement PartialEq
  - PartialEq is used to ensure correctness of partial_cmp()
  - Summary: Types implementing PartialOrd are comparable
  - The solution to our problem!



```rust
pub trait PartialOrd<Rhs = Self>: PartialEq<Rhs>
where
    Rhs: ?Sized,
{
    // Required method
    fn partial_cmp(&self, other: &Rhs) -> Option<Ordering>;

    // Provided methods
    fn lt(&self, other: &Rhs) -> bool { ... }

    fn le(&self, other: &Rhs) -> bool { ... }

    fn gt(&self, other: &Rhs) -> bool { ... }

    fn ge(&self, other: &Rhs) -> bool { ... }
}
```

Reference: https://doc.rust-lang.org/std/cmp/trait.PartialOrd.html



```
fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
    let mut largest: &T = &list[0];

    for item: &T in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Common Traits in the Standard Library

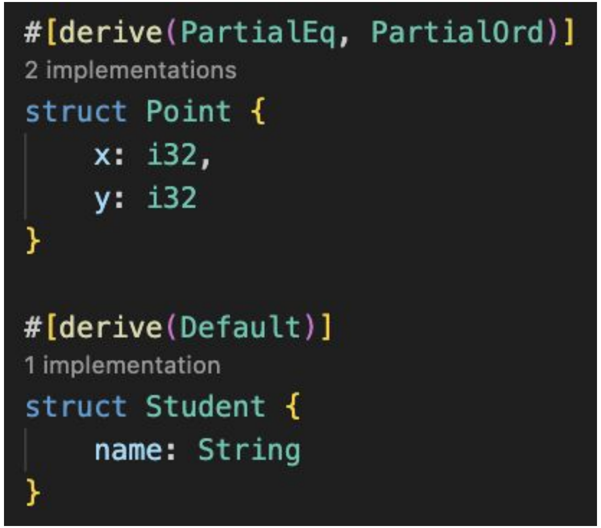
- Display: allows formatting a value as a string
  - Implicitly implement the ToString trait, which defines the to_string() method
  - Thus, prefers implementing Display for converting values to strings
- FromStr: counterpart to ToString, converts string to type value
- Clone: allows cloning a value
  - Defines the clone() method
- Default: defines the default value for a type
  - Allows creating default value using TypeName::default()
- Borrow: Type U implements Borrow<T> means U can be borrowed as T
  - String implements Borrow<str>
- Hash: allows hashing a value
  - Required for use with the HashMap and HashSet data structures

Common Traits in the Standard Library

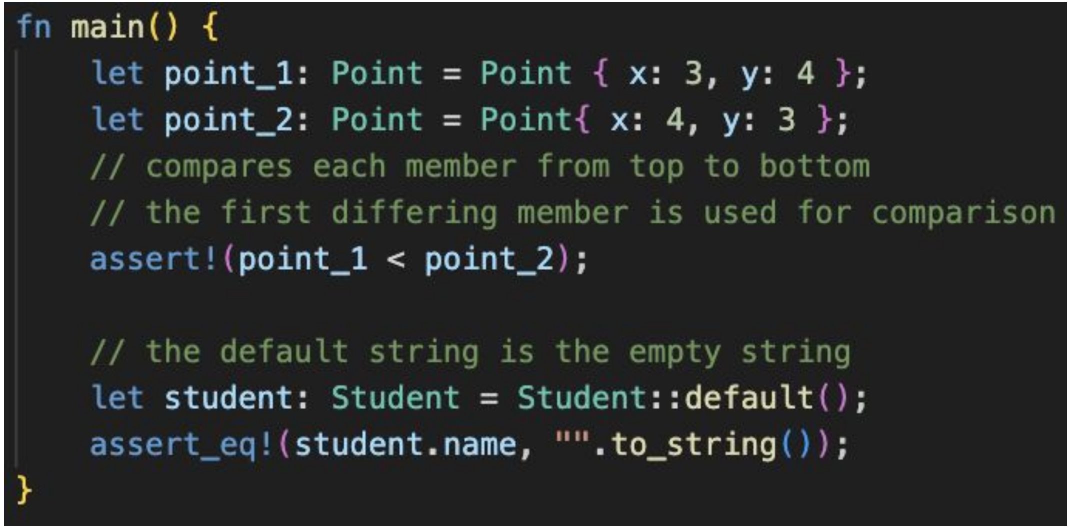
- std::Iter::IntoIterator: converts a value into an iterator
  - Defines into_iter(), iter() and iter_mut()
- Allows the syntax for item in collection { }
- In fact, the for loop in Rust is always tied to the IntoIterator trait
  - for i in 0..vec.len() { } really means for i in [0, .., vec.len() - 1] { } which uses this trait

Deriving Traits

- Some traits can be derived, meaning if every members of a struct implement a trait, we can use #[derive()] to automatically implement said trait on the struct



```
#[derive(PartialEq, PartialOrd)]
2 implementations
struct Point {
    x: i32,
    y: i32
}

#[derive(Default)]
1 implementation
struct Student {
    name: String
}
```



```
fn main() {
    let point_1: Point = Point { x: 3, y: 4 };
    let point_2: Point = Point{ x: 4, y: 3 };
    // compares each member from top to bottom
    // the first differing member is used for comparison
    assert!(point_1 < point_2);

    // the default string is the empty string
    let student: Student = Student::default();
    assert_eq!(student.name, "".to_string());
}
```

- Some other derive-able traits: Clone, Copy, Hash

Announcements

HW 9 is released (due 3/12 11:59 PM)