Intro to Rust Lang

Standard Collections and Generics

Today: Standard Collections and Generics

- Rust's `std::collection` Types
  - `Vec<T>`
  - `String`
  - `HashMap<K, V>`
- Generic Types

Standard Collections

Rust's standard library contains a number of useful data structures called *collections*.

- Most other data types represent a single value, but collections contain multiple values
- Values in collections are (*almost always*) stored on the **heap**
  - The size of each collection does not need to be known at compile time
  - *For more information on other standard library collections, refer to the documentation of the* `std::collections` *module*

Vectors

Review: Vectors

You can create a vector with `new`, and add elements with `push`.

```rust
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);

println!("{:?}", v);
```

Review: `vec!` Macro

Rust provides a *macro* to create vectors easily in your programs.

```
let v = vec![1, 2, 3];

println!("{:?}", v);
```

```
[1, 2, 3]
```

Review: Indexing

You can index into a vector to retrieve a reference to an element.

```rust
let v = vec![1, 2, 3, 4, 5];

let third_ref: &i32 = &v[2];
println!("The third element is {}", third_ref);

let third: i32 = v[2]; // This is only possible because i32 is Copy
println!("The third element is {}", third);
```

`Vec::get()`

You can also use the `get` method to access an optional reference.

```rust
let v = vec![1, 2, 3, 4, 5];

let third: Option<&i32> = v.get(2);
match third {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element."),
}
```

- Using `get` returns `None` if the index is out of bounds instead of panicking

Vec and References

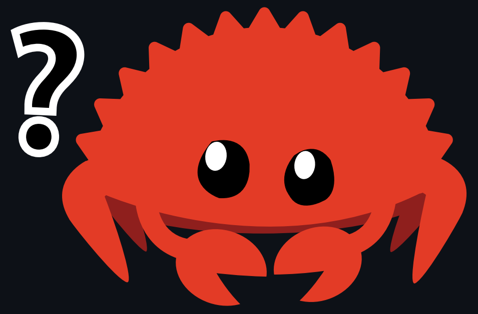
Recall the rules for immutable and mutable references.

```rust
let mut v = vec![1, 2, 3, 4, 5];

let first = &v[0];

v.push(6);

println!("The first element is: {}", first);
```



- You cannot mutate a vector while references to its elements exist
- Appending might resize and reallocate the vector and change its location in memory

`Vec` and References

If we try to run this:

```rust
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0];
v.push(6);
println!("The first element is: {}", first);
```

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:5
  |
3 |     let first = &v[0];
  |                  - immutable borrow occurs here
4 |     v.push(6);
  |     ^^^^^^^^^ mutable borrow occurs here
5 |     println!("The first element is: {}", first);
  |                                          ----- immutable borrow later used here
```

Iterating over a Vector

To access each element in order, we can iterate through the elements with a `for` loop directly, rather than using indices.

```
let v = vec![100, 32, 57];

for elem in &v { // `elem` is a reference to an i32 (&i32)
    println!("{}", elem);
}
```

```
100
32
57
```

Mutable iteration over a Vector

We can also iterate over mutable references to each element to make changes to each element.

```rust
let mut v = vec![100, 32, 57];

for elem in &mut v {  // `elem` is a mutable reference to an i32
    *elem += 50;
}

println!("{:?}", v);
```

```
[150, 82, 107]
```

- We only have a single mutable reference into the vector at a time
    - We pass the borrow checker's rules!

For Loop Sugar

You can also *consume* the vector when you want to loop over it.

```
let v = vec![100, 32, 57];

for i in v {
    println!("{}", i);
}

// println!("{:?}", v); <-- Can't do this anymore!
```

- We'll talk more about this in week 7!

Deref Coercion to `&[T]`

Instead of a function taking a `&Vec<T>` as a parameter, we can take a `&[T]`.

```
fn largest(list: &Vec<i32>) -> &i32
```

```
fn largest(list: &[i32]) -> &i32
```

- The second is more general and preferred
- We can do this because of *deref coercion*
  - This basically means you can turn a `&Vec<T>` into a `&[T]`
- We'll talk more about this in week 9!

Use Enums to Store Multiple Types

Vectors can only store values of the same type, so we can use enums to store values of different types (variants).

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}

let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

Vectors and Ownership

Vectors own all of their contained elements.

```
let mut v = vec![String::from("rust"), String::from("is")];

let s = String::from("great!");

v.push(s); // move `s` into `v`

// `s` is no longer usable here!
```

- To insert an owned value, it must be *moved* into the vector
    - In other words, ownership must be transferred to the vector

Dropping a Vector

Like any other struct, a vector is dropped when it goes out of scope.

```rust
{
    let v = vec!["rust".to_string(), "is".to_string(), "great!".to_string()];

    stuff(&v); // do stuff with `v`

} // <- `v` goes out of scope and everything it contains is freed
```

- When the vector gets dropped, all of its contents are also dropped
- The borrow checker will ensure that references into the vector cannot be used after it has been dropped

String

What is a String?

- A `String` is essentially a `Vec` of bytes interpreted as text
- We introduced them back in week 2, but now we'll look at them in more depth
- New Rust programmers may be confused by:
  - `String`'s internal UTF-8 encoding
  - Rust's prevention of possible logical errors from the encoding
  - `String`s are not as simple as they may initially seem

What is a String?

- Rust "only" has one string type in the core language, `str`
  - We almost always see it in its borrowed form, `&str`
  - String slices are `&str`
  - String literals are also `&str`
    - References data stored in the program's binary
- `String` is a growable, mutable, owned, UTF-8 encoded string type

Creating a `String`

You can create a `String` with the methods `new`, `to_string`, or `from`.

```
let mut s = String::new(); // empty mutable string

let data = "initial contents"; // string literal

let s = data.to_string(); // string literal into `String`

// the method also works on a literal directly:
let s = "initial contents".to_string();

let s = String::from("initial contents"); // string literal into `String`
```

Strings are UTF-8 Encoded

We can represent any properly encoded data in `String`.

Here are some greetings in different languages!

```
let hello = String::from("السلام عليكم");
let hello = String::from("Dobrý den");
let hello = String::from("Hallo");
let hello = String::from("שָׁלוֹם");
let hello = String::from("नमस्ते");
let hello = String::from("Olá");
let hello = String::from("Привет");
let hello = String::from("Hola");
```

Updating a `String`

We can grow a `String` by using the `push` or `push_str` methods.

```
let mut s = String::from("foo");

s.push('b'); // push a char
s.push_str("ar"); // push a &str

println!("{}", s);
```

```
foobar
```

Updating a `String`

The `push_str` method takes a string slice, because we don't want to take ownership of the string passed in.

```
let mut s1 = String::from("foo");

let s2 = String::from("bar");

s1.push_str(&s2);

println!("s2 is {}", s2); // `s2` is still valid!
```

```
s2 is bar
```

Concatenating Strings

You can combine two strings with `+`:

```
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // note s1 has been moved here and can no longer be used
```

This is syntactic sugar for a function whose signature looks something like this:

```
fn add(self, s: &str) -> String
```

- Notice that `add` takes full ownership of `self`
- Also notice `add` takes `&str` as its second parameter and not `&String`
  - This is the same *deref coercion* as with `&Vec<T>` to `&[T]`!

Concatenating Multiple Strings

```
let s1 = String::from("tic");
let s2 = String::from("tac");
let s3 = String::from("toe");
```

To combine multiple strings, you can use multiple `+`'s:

```
let s = s1 + "-" + &s2 + "-" + &s3;
```

Or you can use the `format!` macro:

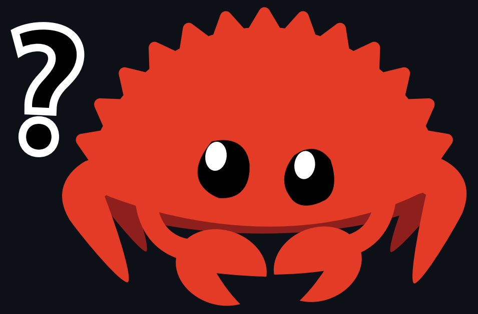
```
let s = format!("{}-{}-{}", s1, s2, s3);

let s = format!("{s1}-{s2}-{s3}"); // relatively new shorthand!
```

Indexing into Strings

This code might seem reasonable from any other programming language like Python or C.

```
let s1 = String::from("hello");
let h = s1[0];
```



- Accessing individual characters in a string by indexing is common in many languages
- However, if you try to access a `String` using an index, you will get an error

Indexing into Strings

```rust
let s1 = String::from("hello");
let h = s1[0];
```

```
error[E0277]: the type `String` cannot be indexed by `{integer}`
 --> src/main.rs:3:13
  |
3 |     let h = s1[0];
  |             ^^^^^ `String` cannot be indexed by `{integer}`
  |
  = help: the trait `Index<{integer}>` is not implemented for `String`
```

- Why won't Rust allow indexing `String`?

Internal Representation of Strings

A `String` is really a wrapper over `Vec<u8>`, or a vector of bytes.

```
let hello = String::from("Hola");
```

- How long is this string?
  - The length of the string is 4
  - The internal vector storing the string `"Hola"` is 4 bytes long
- Simple enough, right?

Internal Representation: Cyrillic

Now suppose we wanted to say "Hello", but in Russian.

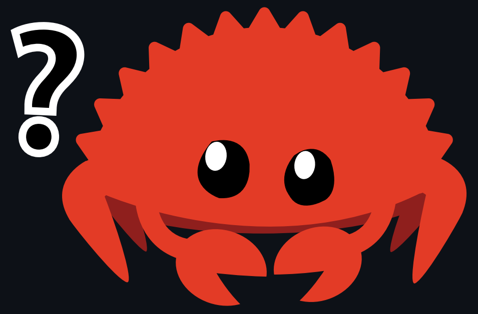
```
let hello = String::from("Привет");
```

- How long is this string?
  - There are 6 distinct characters
  - However, the string's `len` is 12, the number of bytes needed in the internal vector

Internal Representation: UTF-8

Let's revisit some invalid Rust code again.

```
let hello = "Привет";
let answer = &hello[0];
```



- What *should* `answer` be?
  - It can't be `П`, internally it is represented by 2 bytes: `[208, 159]`
  - Do we return `208` instead?
- There isn't any obvious expected behavior here...

Internal Representation: UTF-8

```
let hello = "Привет";
let answer = &hello[0]; // BAD!
```

- Anything we can return here might not be an "expected" result
- The philosophy of Rust is to not compile this code at all
  - Prevents misunderstandings early in the development process
- Further reading on UTF-8: Rust Book Chapter 8.2

Slicing Strings

Instead of indexing with a single number, you can use `[]` with a *range* to create a string slice containing specific bytes.

```
let hello = "Привет";

let s = &hello[0..4]; // `s` == "Пр"
```

Slicing Strings

However, if we try to slice only a part of a character's bytes, Rust panics at runtime.

```
let hello = "Привет";

let s = &hello[0..1];
```

```
thread 'main' panicked at 'byte index 1 is not a char boundary;
it is inside 'П' (bytes 0..2) of `Привет`'
```

- This happens in the same way that an invalid index causes a panic!

Iterating Over Strings

Normally, we want to iterate over individual Unicode scalar values, and we can use the `chars` method.

```
for c in "Пр".chars() {
    println!("{c}");
}
```

```
П
р
```

Iterating Over Strings

Alternatively, if you want the actual raw bytes, you can use the `bytes` method.

```
for b in "Пр".bytes() {
    println!("{b}");
}
```

```
208
159
209
128
```

Recap: Strings

- Rust chooses to use UTF-8 strings as the default (for both `String` and `&str`)
  - Programmers have to think about handling unicode upfront
  - The complexity brought on by encodings is more apparent in Rust
  - However, this prevents having to deal with non-ASCII characters later!
- The standard library offers many methods for `String` and `&str` types to help handle these complex situations correctly

HashMap

`HashMap<K, V>`

The type `HashMap<K, V>` stores keys with type `K` mapped to values with type `V`.

- Many languages support this kind of data structure, even if they use a different name:
  - Hash
  - Map
  - Object
  - Hash Table
  - Dictionary
  - Associative Array

Creating a Hash Map

We can create a new hash map with `new` and insert entries with `insert`.

```rust
use std::collections::HashMap;

let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
```

- Note that we need to import `HashMap` from the standard library's collections module with `use`
- We'll talk more about `use` in week 6!

Accessing Values in a Hash Map

We can use the `get` method to get the value associated with a key.

```
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

let team_name = String::from("Blue");
let score = scores.get(&team_name).unwrap_or(&0);
```

- The `get` method returns an `Option<&V>`, similar to `Vec::get()`
- We use `unwrap_or(&0)` on the result
  - If it returns `Some(&x)`, we unwrap and get `&x`
  - If it returns `None`, we go to the default case `&0`

Iterating over a Hash Map

We can iterate over each key/value pair using a `for` loop, similar to vectors.

```
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);

for (key, value) in scores {
    println!("{key}: {value}");
}
```

```
Yellow: 50
Blue: 10
```

- *Note that the order is non-deterministic*

Hash Maps and Ownership

Hash maps own their contained data, just like vectors.

```
let field_name = String::from("Favorite color");
let field_value = String::from("Blue");

let mut map = HashMap::new();
map.insert(field_name, field_value);

// field_name and field_value are invalid at this point!
```

Updating a Hash Map

Hash maps only contain one value for each distinct key, so to update we can just insert twice.

```rust
let mut scores = HashMap::new();

scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Blue"), 25);

println!("{:?}", scores);
```

```
{"Blue": 25}
```

- Inserting twice overwrites the existing value for the given key

Accessing a Hash Map with Defaults

- A common pattern when accessing a `HashMap` is:
  - If the key exists, we want to access the value
  - If the key does not exist, insert a default value and then access it
- `HashMap` has a special API called `Entry`

Hash Map Entries

To insert a value if the key does not already exist, you can use the `Entry` API and the method `or_insert`.

```
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);

scores.entry(String::from("Yellow")).or_insert(50);
scores.entry(String::from("Blue")).or_insert(50);

println!("{:?}", scores);
```

```
{"Yellow": 50, "Blue": 10}
```

Hash Map Entries

If you want to update a value, or provide a default if it doesn't yet exist, you can do something similar:

```rust
let text = "hello world wonderful world";

let mut map = HashMap::new();

for word in text.split_whitespace() {
    let count: &mut usize = map.entry(word).or_insert(0);
    *count += 1;
}

println!("{:?}", map);
```

```
{"world": 2, "hello": 1, "wonderful": 1}
```

`hash_map::Entry::or_insert`

The method `or_insert` has the following signature:

```
fn or_insert(self, default: V) -> &mut V
```

- It gives out a mutable reference
  - That reference are guaranteed to point to valid data
  - We need to provide a default, otherwise this data might not exist
- Shorter and more readable than separate conditionals

Recap:

- We covered The Rust Book Chapter 8
- Always refer to the documentation!
  - `Vec<T>` documentation
  - `String` documentation
  - `HashMap<K, V>` documentation

Generics

Generics

So what was the deal with the `T` in `Vec<T>`, and `K, V` in `HashMap<K, V>`?

- We refer to these as *generic* types
- Think of it as being able to fill in any type you want in place of `T`
- Generics allow us to replace specific types with a placeholder that represents multiple types
    - Removes code duplication

Removing Code Duplication

Let's say we want to find the largest number in a list.

```rust
let number_list = vec![34, 50, 25, 100, 65];

let mut largest = &number_list[0];

// Pretend the list always has at least 1 element.
for number in &number_list {
    if number > largest {
        largest = number;
    }
}

println!("The largest number is {}", largest);
```

Removing Code Duplication

What if we have multiple lists? We then have to do multiple searches.

```
let number_list = vec![34, 50, 25, 100, 65];
let mut largest = &number_list[0];
for number in &number_list {
    if number > largest {
        largest = number;
    }
}
println!("The largest number is {}", largest);

let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];
let mut largest = &number_list[0];
for number in &number_list {
    if number > largest {
        largest = number;
    }
}
```

Removing Code Duplication

Instead, we can make a function called `largest`.

```rust
fn largest(list: &[i32]) -> &i32 {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    println!("The largest number is {}", largest(&number_list));

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];
    println!("The largest number is {}", largest(&number_list));
```

Remove Function Duplication

What if we also wanted to find the largest character in a slice?

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

- This is effectively the same as finding the largest number in a list
- We would still need to write a new function in addition to `largest`
- Can we remove a *function* that has been duplicated?

Generic Functions

We can define a function as generic over some type `T` with a tag `<T>`:

```
fn largest<T>(list: &[T]) -> &T
```

- This function is generic over `T`
- This function takes in a slice of `T` as input
- This function returns a reference to `T`
- `T` can be *any** type!

Generic Functions

Generic types can have any name, not just `<T>`:

```
fn largest<T>(list: &[T]) -> &T
```

```
fn largest<Key>(list: &[Key]) -> &Key
```

```
fn largest<Smile>(list: &[Smile]) -> &Smile
```

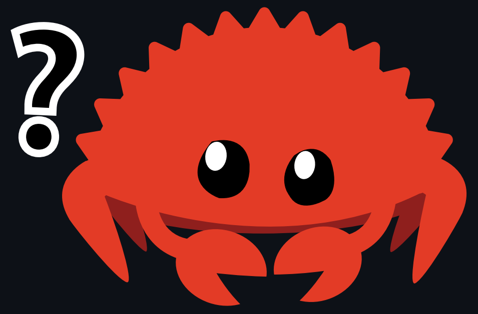
- All of these essentially mean the same thing!
    - Last one is *frowned* upon since it might seem like a struct or enum

Generic Functions

Let's try to modify our old function directly:

```
fn largest<T>(list: &[T]) -> &T {
    let mut largest = &list[0];
    for item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    println!("The largest number is {}",
             largest(&[34, 50, 25, 100, 65]));
    println!("The largest char is {}",
             largest(&['y', 'm', 'a', 'q']));
}
```



Generic Functions

We get an error:

```
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:4:17
  |
4 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++
```

```
error[E0369]: binary operation `>` cannot be applied to type `&T`
 --> src/main.rs:4:17
  |
4 |         if item > largest {
  |            ---- ^ ------- &T
  |            |
  |            &T
  |
help: consider restricting type parameter `T`
  |
1 | fn largest<T: std::cmp::PartialOrd>(list: &[T]) -> &T {
  |             ++++++++++++++++++++++
```

- We cannot compare two `&T` to each other
- We've stated that `T` can be *any* type, regardless of if `T` is a type that cannot actually be compared
- Let's just follow the compiler's advice for now!

Generic Functions

Once we make that change, this works!

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

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
The largest number is 100
The largest char is y
```

Sneak Peek: Traits

```rust
use std::cmp::PartialOrd;

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

    for item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

- We'll talk about type restrictions with *traits* next week!
- For now, all you need to know is that we need the `PartialOrd` trait to enable

Generic Structs

We can define structs to contain a generic type using the `<T>` syntax as well!

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

Generic Structs

Observe that this declaration defines both the `x` field and the `y` field to be of the same type.

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

Generic Structs

If we try to compile this, we get an error

```
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer,
                                             found floating-point number
```

Generic Structs

If we want a struct that allows different generic fields to have different types, we need to define another generic type.

```rust
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

- Note that they can still be the same!

Generic Enums

Recall the `Option<T>` type:

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

- This is a generic enum over `T`!

Generic Enums

Enums can be generic over multiple types, just like structs.

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

- This enum is generic over `T` and `E`, with each stored in a variant
- `Result<T, E>` is a very common type in the standard library that we will talk about next week!

Generic Methods

Methods on structs can also be generic.

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };
    println!("p.x = {}", p.x());
}
```

Generic Methods

```
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
```

- Observe that we have to declare `T` after the `impl` as well as after `Point`
- This is to specify that we're implementing methods on the *type* `Point<T>`
- This is different from implementing methods on the *type* `Point<f32>`

Generic `impl`

We could have made an implementation specific to `Point<f32>`:

```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

- This code means that `Point<f32>` will have an additional `distance_from_origin` method on top of the methods defined for `Point<T>`

Generic `impl`

Going back to the `Point<T, U>` example:

```rust
struct Point<T, U> {
    x: T,
    y: U,
}
```

We could implement methods for when `x` is `i32`, but with no restrictions on `y`.

```rust
impl<U> Point<i32, U> {
    fn get_sum_x(&self, other: Point<i32, U>) -> i32 {
        self.x + other.x
    }
}
```

Generic `impl`

However, this actually restricts the type of `other` to have the same generic type parameters `<i32, U>`.

```
impl<U> Point<i32, U> {
    fn get_sum_x(&self, other: Point<i32, U>) -> i32 {
        self.x + other.x
    }
}

fn main() {
    let p1 = Point { x: 5, y: 3.2 }; // y is f64
    let p2 = Point { x: 5, y: 4.4 }; // y is also f64
    println!("{}", p1.get_sum_x(p2))
}
```

- Note that `U` has to be the same in both `self` and `other`

Generic `impl`

To solve this, we can make the method generic over another type:

```
impl<U> Point<i32, U> {
    fn get_sum_x<V>(&self, other: Point<i32, V>) -> i32 {
        self.x + other.x
    }
}

fn main() {
    let p1 = Point { x: 5, y: 3.2 };             // y is f64
    let p2 = Point { x: 5, y: String::new() }; // y is String
    println!("{}", p1.get_sum_x(p2))
}
```

Here's another example of a generic `impl`:

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);
    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Performance of Generics

- The good news is that there is *zero* overhead to using generics!
    - The work is done at compile-time instead of runtime.
- Rust accomplishes this with *monomorphization*

Monomorphization

Let's look at how this works by using the standard library's generic `Option<T>`:

```
let integer = Some(5);
let float = Some(5.0);
```

- The compiler will identify which types `T` can take on by find all instances of `Option<T>`, in this case `i32` and `f64`
- It creates monomorphized versions of `Option` specific to those types

Monomorphization

The compiler will generate something similar to the following:

```rust
enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```


- All extra work is performed at compile time!

Recap: Generics

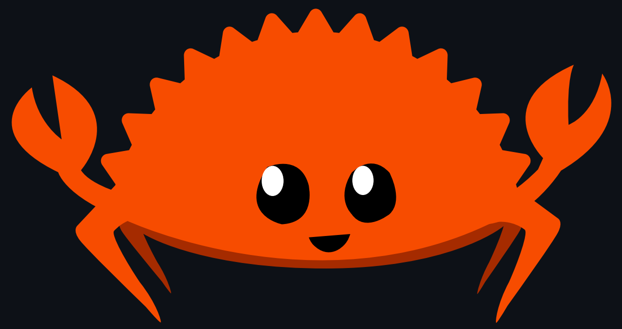
- Generics allow us to reduce code duplication
- Monomorphization means we do not incur any runtime cost!

Homework 4

- You'll be implementing two collection data structures:
  - `MultiSet`
    - A collection that stores unordered values and tracks multiplicity
  - `MultiMap`
    - A collection that maps keys to any number of values
- Make sure you are familiar with the API for `HashMap` and `Entry`!

Next Lecture: Errors and Traits

Thanks for coming!



Slides created by:

Connor Tsui, Benjamin Owad, David Rudo,
Jessica Ruan, Fiona Fisher, Terrance Chen