Lecture 10

Generic Data Types

Goals For Today

- Generics
- Smart Pointers

Motivating Example

- In Rust, several types can contain other types
  - Vec<T>
  - [T]
  - HashMap<K, V>
  - Option<T>
  - Result<T>
- How do they do this?

Motivating Example (cont)

- Both functions do essentially the same thing, just with different function signatures
- If want to apply the same logic to other types (ex: u16), need to repeat these same code
- Seems wasteful
- Can we reduce the 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
}
```

Generics

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

Generics in Structs

- Note the struct signature: Point<T>, meaning "struct Point containing members with generic type T"
- x and y can be any type, but they must be of the same type
- What if we want them to be of different types T and U?

```rust
fn main() {
    let integer: Point<i32> = Point { x: 1, y: 2 };
    let float: Point<f32> = Point { x: 1.0, y: 2.0 };
}

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

Generics in Structs (cont)

- Note struct signature: Point<T, U>; meaning "struct Point containing members with generic types T and U"
- Allows x and y to take different types
- Syntax rule: must list all generic types of members in struct definition

```rust
fn main() {
    let wont_work: Point<i32> = Point { x: 1, y: 2.0 };
}

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

```rust
fn main() {
    let will_work: Point<i32, f32> = Point { x: 1, y: 2.0 };
}

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

Generics in Enums

- Enum signature: Number<T>, meaning "enum Number with variant(s) containing value of type T"
- Note: Even if we use a variant not containing T, still need to declare the concrete type for T

```rust
fn main() {
    let finite_num: Number<i32> = Number::Finite(5);
    let infinite_num: Number<u32> = Number::Infinite;
}

0 implementations
enum Number<T> {
    Finite(T),
    Infinite
}
```

Generics in Enums (cont)

- Number<T, U, V>: “enum Number with variants containing values of type T, U, V”
- Syntax rule: must list the generic types of all values, contained in every variants of the enum

```rust
fn main() {
    let finite_1d_num: Number<i32, u32, f32> = Number::OneDimFinite(5);
    let finite_2d_num: Number<i32, u32, f32> = Number::TwoDimFinite(2, 1.0);
    let infinite_num: Number<i32, u32, f32> = Number::Infinite;
}

0 implementations
enum Number<T, U, V> {
    OneDimFinite(T),
    TwoDimFinite(U, V),
    Infinite
}
```

Generics in Enums (cont)

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

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

Generics in Methods

- Syntax: impl<T> Point<T>
- Meaning: methods in this implementation work on Points containing generic type T
- Wait, why need the distinction?

```
fn main() {
    let p: Point<i32> = Point { x: 5, y: 10 };

    // prints out 'p.x = 5'
    println!("p.x = {}", p.x());
}

1 implementation
struct Point<T> {
    x: T,
    y: T,
}

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

Generics in Methods (cont)

- Even though Point<T> contains generic T , we can still create methods for a concrete type
- Note: No longer need to specify generic T in the impl keyword, still need to specify the concrete type for the struct

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

impl<T> Point<T> {
    fn x(&self) -> i32 {
        5
    }
}

impl Point<i32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.pow(exp: 2) as f32 + self.y.pow(exp: 2) as f32).sqrt()
    }
}
```

```rust
fn main() {
    let p_int: Point<i32> = Point { x: 3, y: 4 };
    // p_int_dist = 5
    let p_int_dist: f32 = p_int.distance_from_origin();

    let p_float: Point<f64> = Point { x: 5.0, y: 10.0 };
    // Error: no method distance_from_origin found for Point<f64>
    // method was found for Point<i32>
    let p_float_dist = p_float.distance_from_origin();
}
```

Generics in Functions

- Now that we know generics exist, we can combine these 2 functions! (and express the same logic for other types)

```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 in Functions (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 {
   |             ++++++++++++++++++++++
```

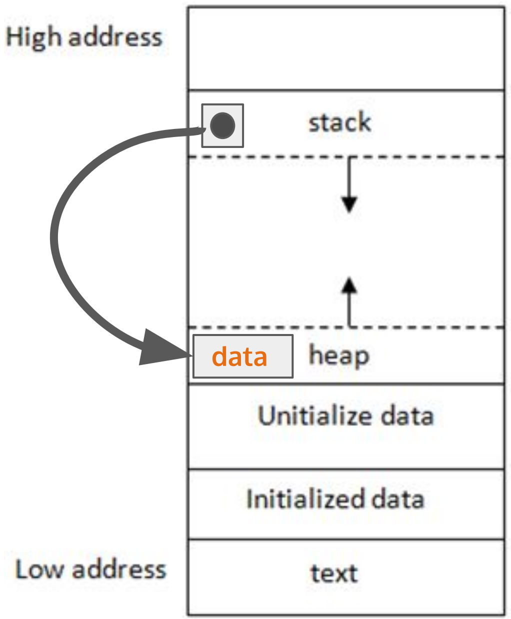
- What if T is HashMap<i32, i32>?
- The concrete type the user inputs into the function might not be comparable

Smart Pointers

- Pointers (raw pointer): variables that contain (point to) an address in memory
- Smart pointers: act like pointers but with additional capabilities
  - Example: String and Vec<T> - manage own memory, knows the length of collections they store
  - Take ownership of data they point to
  - When smart pointer goes out of scope, data is also dropped
- Due to ownership rules, safe Rust does not allow raw pointers
- All pointer-like types in Rust are smart pointers

The Box smart pointer

- Box<T> allows you to store data on the heap
  - Unlike Vec<T>/String, said data does not need to be a collection
- Raw pointer on the stack (like all containers/data structures).
- When pointer on the stack goes out of scope, data on the heap is freed (dropped)



Reference:

- https://doc.rust-lang.org/book/ch15-01-box.html

Using Boxes

- Create a Box smart pointer using Box::new(data: T)
- Make the Box mutable if you want to modify the stored data
- Dereference (*) the smart pointer to access the data (just like raw pointers in C++)
  - Do not need to dereference when calling functions on the data, Rust does it for you
- Sounds familiar?

```
// this int lives on the stack
let stack_int: i32 = 5;

// this int lives on the heap, with a stack pointer to it
let heap_int: Box<i32> = Box::new(4);
// prints out: `Value of heap int: 4`;
println!("Value of heap int: {}", *heap_int);
```

Boxes vs References

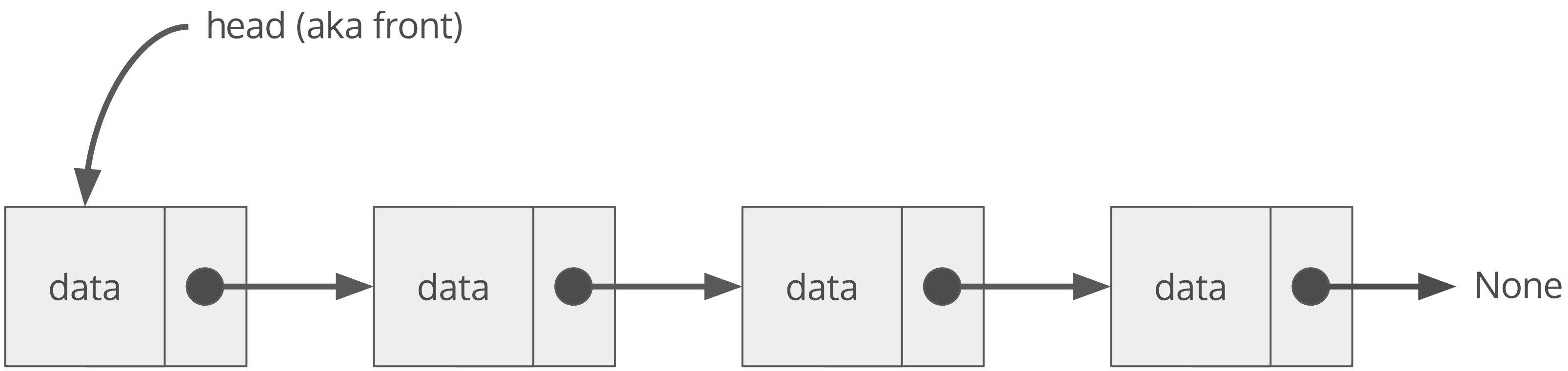
- Both act like pointers, both ensure that the data they point to can’t be null
- Box:
  - Owns the data it points to
- References:
  - Borrow the data they point to
  - Data is owned by some other variable
- Want nullable pointers anyway?
  - Option<Box<T>>
  - Option<&T>

CS128 Honors

Why Boxes?

- When you have a type whose size can't be known at compile time and want to use a value of that type in a context requiring an exact size (any context involving the stack)
  - Conceptually, the size of the stack pointer will be treated as said exact size
  - Also why Vec<T>/String exist!
- Example use cases:
  - Implementing a Linked List

Linked Lists (Wrong Rust Implementation)

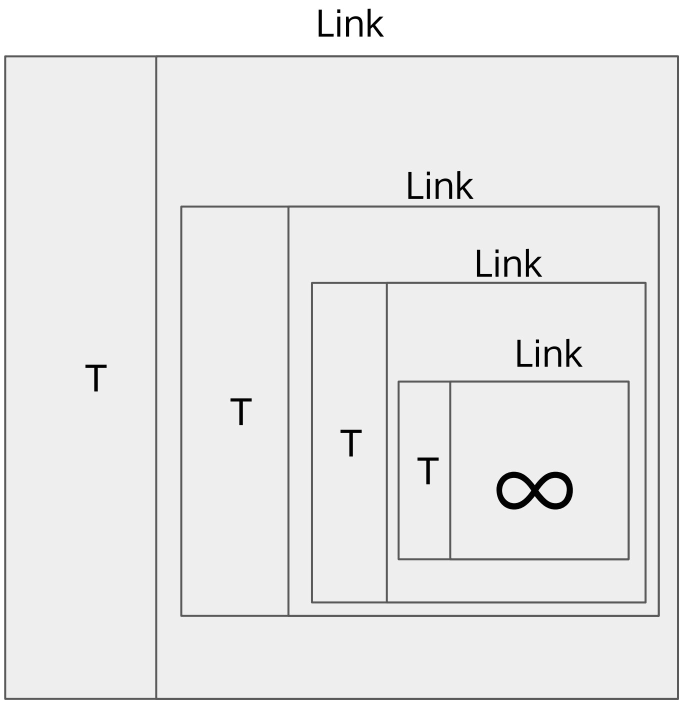



```rust
pub struct Link<T: std::fmt::Display> {
    thing: T,
    next: Option<Link<T>>,
}
```

Linked Lists (Wrong Rust Implementation)

```rust
pub struct Link<T: std::fmt::Display> {
    thing: T,
    next: Option<Link<T>>,
}
```

Rust can’t know the exact size of Link and Link lives on stack by default -> forbidden

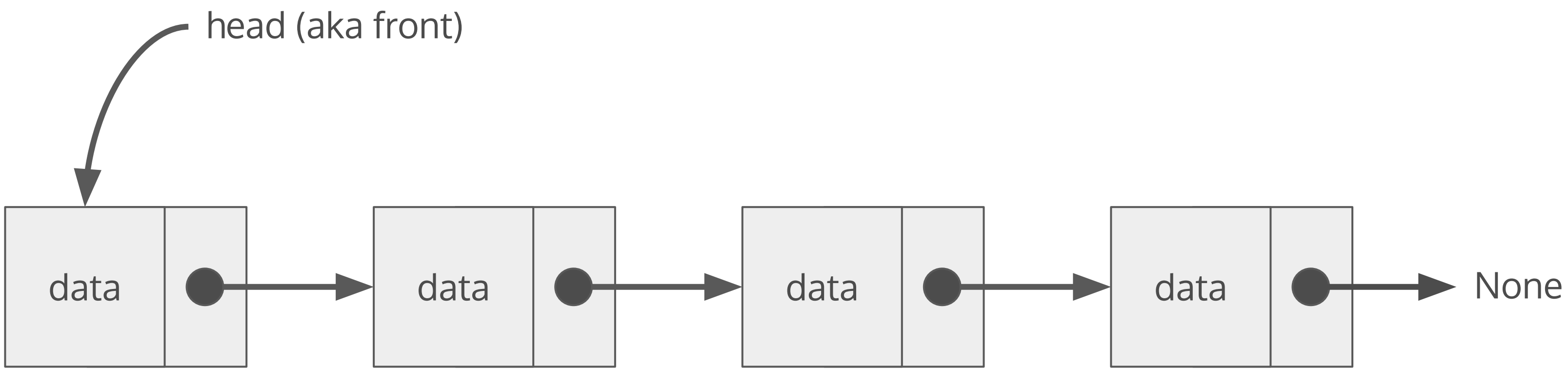



```
error[E0072]: recursive type `Link` has infinite size
  --> src/bin/list.rs:8:1
   |
8  | pub struct Link<T: std::fmt::Display> {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ recursive type has infinite size
9  |     thing: T,
10 |     next: Option<Link<T>>,
   |           --------------- recursive without indirection
   |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to make `Link` representable
   |
10 |     next: Box<Option<Link<T>>>,
   |           ++++               +
```

Reference:

- https://doc.rust-lang.org/book/ch15-01-box.html

Linked Lists (Correct Rust Implementation)




```rust
pub struct Link<T> {
    thing: T,
    next: Option<Box<Link<T>>>,
}
```

Due to the indirection, Rust will determine the exact size as the size of the first Box's stack pointer!

Announcements

HW 8 has been released (due 3/07 at 11:59 PM)

MP 2 has been released (due 3/14 at 11:59 PM)