Generics & Smart Pointers in Rust

Lecture 10

Slides: Neil Kaushikkar

Goals For Today

- Answering Your Questions
- Review Struct Methods
- Introduction to Generics
- Introduction to Smart Pointers
- Linked Lists and HW8 hints

Reminders

- HW7 due 10/4 at 11:59 pm CT
- HW8 releasing tonight due 10/6 at 11:59 pm CT
- MP2 due 10/7 at 11:59pm CT

Answering Your Questions!

- “Does **&word[..]** mean **&(*word)[..]** given that 'word' is a **String**?”
  - Not quite - if word is a **String**, you cannot dereference it
  - **&word[..]** means a substring of ‘word’ that covers the entire string
    - **&** borrows the original word
    - **[..]** defines where to start and end the substring

Struct Methods Review

- Very similar to functions:
  - Use the **fn** keyword
  - Contain code that is run somewhere else
- However, they are defined in the context of a **struct** (or an **enum** or **trait**)
  - Use the **impl** keyword to implement methods for your custom type
  - The first parameter is always **self**, which represents the instance of the struct the method is being called on

The self Keyword Review

- **&self** - IMMUTABLE borrow to the current instance
- **&mut self** - MUTABLE borrow to the current instance
- Use dot notation on self to access **struct** fields and call other **struct** methods within **struct** methods

Introduction to Generics

- Generics allow you to generalize types to broader cases
- They help in reducing code duplication
- "Generic type parameters" are typically represented as <T>
    - Describes anything that accepts one or more generic type parameters <T>
- If a type is NOT generic, it is concrete

Reference:

- https://doc.rust-lang.org/book/ch10-00-generics.html

This Might Seem Familiar!

- **Option<T>** - an optional elements of the type **T**
- **Result<T, E>** - a result with some success value of type **T** and some error value of type **E**
- **Vec<T>** - an ordered collection of elements of the same type **T**
- **HashSet<T>** - an unordered collection of *distinct* elements of the same type **T**
    - This will be useful in MP2!
- **HashMap<K, V>** - an unordered collection of *distinct* keys of type **K** and their corresponding values of some type **V**
    - This will be useful in MP3!

Generics in Rust

- Templated custom types (**struct**s & **enum**s)
- Templated functions
- Templated interfaces (**trait**s)

Reference:

- https://doc.rust-lang.org/book/ch10-00-generics.html

Writing Your Own Generic Code

```rust
struct Val {
    val: f64,
}

// impl of Val
impl Val {
    fn value(&self) -> &f64 {
        &self.val
    }
}
```

```rust
struct GenVal<T> {
    gen_val: T,
}

// impl of GenVal for a generic type `T`
impl<T> GenVal<T> {
    fn value(&self) -> &T {
        &self.gen_val
    }
}
```

```rust
fn main() {
    let w = Val { val: 3.0 };

    let x: GenVal<i32> = GenVal { gen_val: 3i32 };
    let y: GenVal<String> = GenVal { gen_val: "abc".to_string() };
    let z: GenVal<f64> = GenVal { gen_val: 3.0 };

    println!("{}, {}, {}, {}", w.value(), x.value(), y.value(), z.value());
}
```

Reference:

- https://doc.rust-lang.org/book/ch10-01-syntax.html

Let’s Write Our Own Option Enum

Smart Pointers

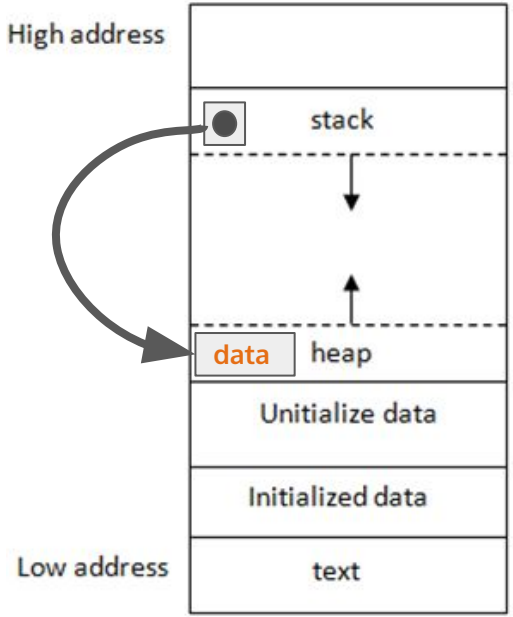
- Pointers are a general concept for variables that contain an address in memory
- Smart pointers are data structures that not only act like a pointer but also have additional metadata and capabilities
  - **String** and **Vec<T>** are examples of smart pointers - manage own memory
  - Smart pointers take ownership of the data they point to
  - When the smart pointer structure goes out of scope, the data it owns is also dropped

Reference:

- https://doc.rust-lang.org/book/ch15-00-smart-pointers.html

The Box Smart Pointer

- Boxes allow you to store data on the heap rather than the stack
- The pointer to the heap data is on the stack
- When the pointer on the stack goes out of scope, the data on the heap is freed



Reference:

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

Using Boxes

Using Boxes

- Create a box smart pointer using **Box::new(data: T)**
- Make the box variable **mut**able if you wish to modify the underlying data
- Dereference the smart pointer to get access to the data pointed at
    - HOWEVER, if you are calling a function on the data, you DO NOT need to dereference the box smart pointer (Rust will do it for you)

Reference:

- https://doc.rust-lang.org/std/boxed/struct.Box.html

Intro to Boxes

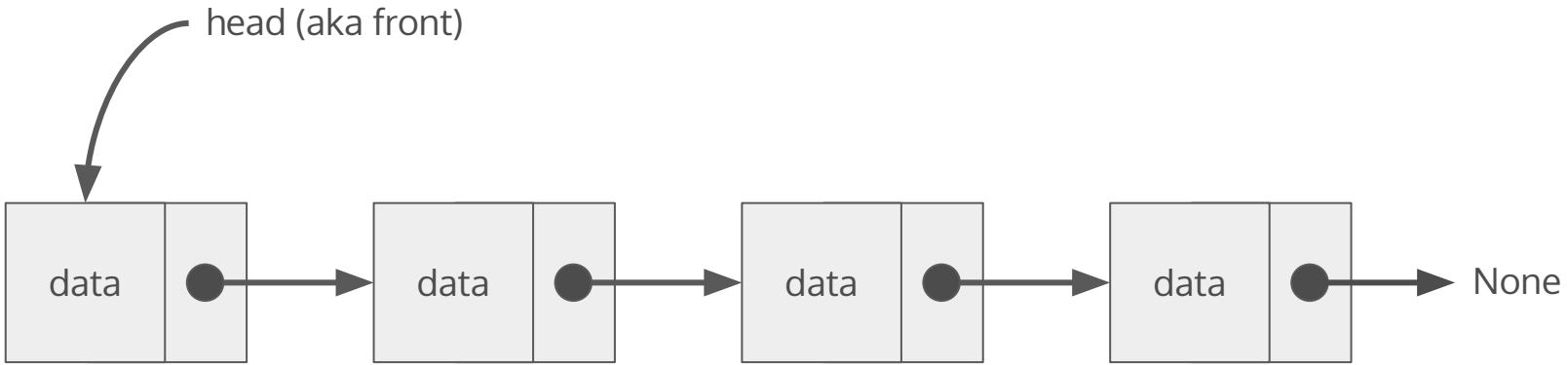
The Box Smart Pointer

- Used when you have a type whose size can't be known at compile time and you want to use a value of that type in a context that requires an exact size
- Also used when you want to own a value and you care only that it's a type that implements a particular trait rather than being of a specific type
    - May cover more about this in the OOP Special Topics Lectures

Reference:

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

Linked Lists (Wrong Rust Implementation)

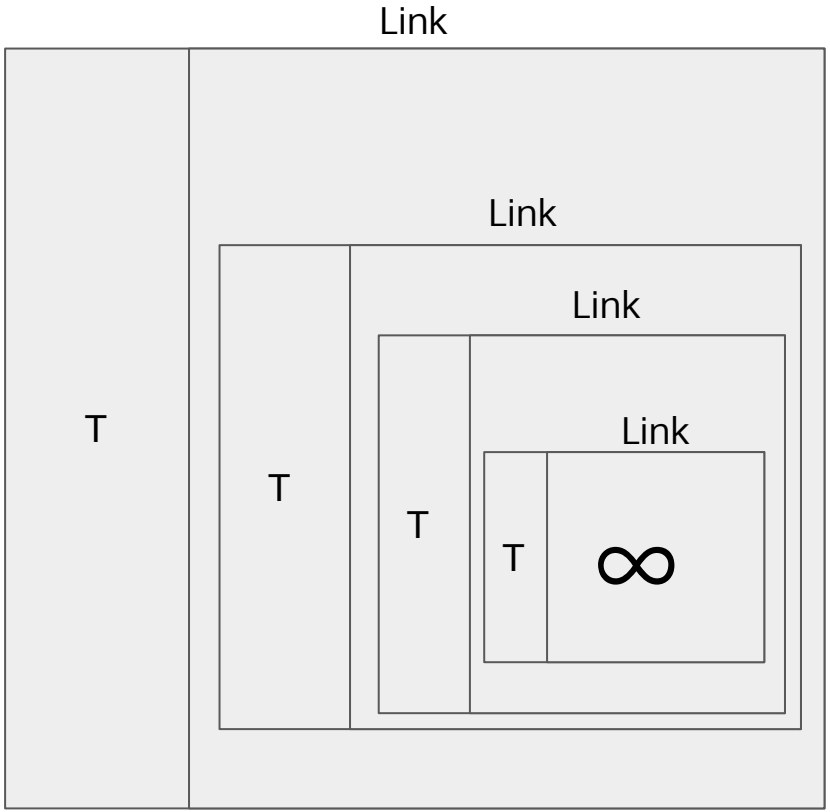



```rust
pub struct Link<T: std::fmt::Display> {
    thing: T,
    next: Option<Link<T>>,
}
```

Linked Lists (Wrong Rust Implementation)

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

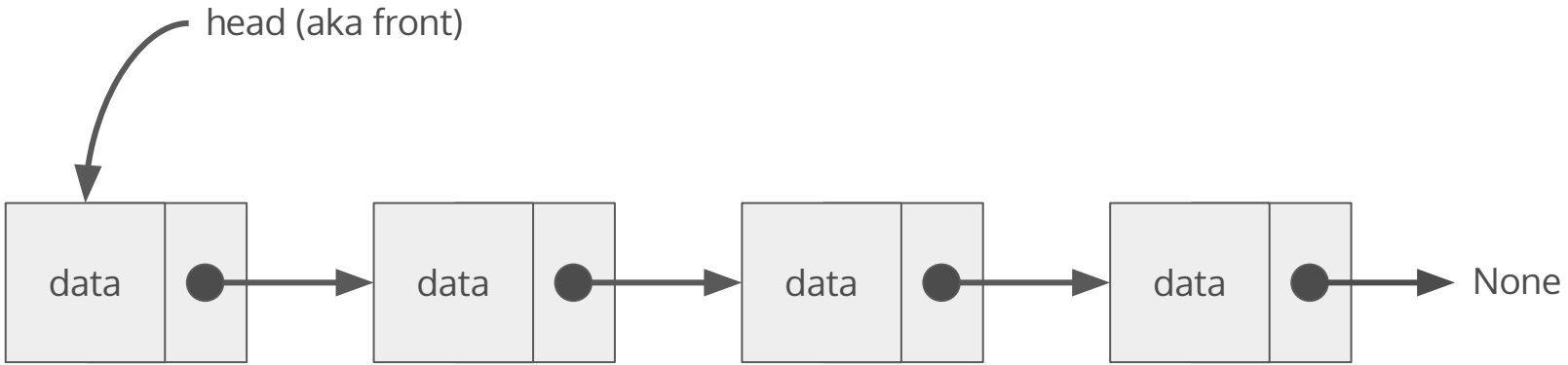
```
error[E0072]: recursive type `Link` has infinite size
  --> src/bin/list.rs:8:1
   |
8  | pub struct Link<T: std::fmt::Display> {
   | ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ recursive type has infinite size
9  |     thing: T,
10 |     next: Option<Link<T>>,
   |           -------------- recursive 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>>>,
}
```

Linked Lists & HW8 Head Start

That’s All Folks!