# Smart Pointers

Mustakimur R. Khandaker

# Introduction

A pointer is a variable that contains an address in memory. This address refers to, or “points at,” some other data.

- The most common kind of pointer in Rust is a reference.

Smart pointers are data structures that not only act like a pointer but also have additional metadata and capabilities.

- For example: String in Rust.

In Rust, the different smart pointers defined in the standard library provide functionality beyond that provided by references.

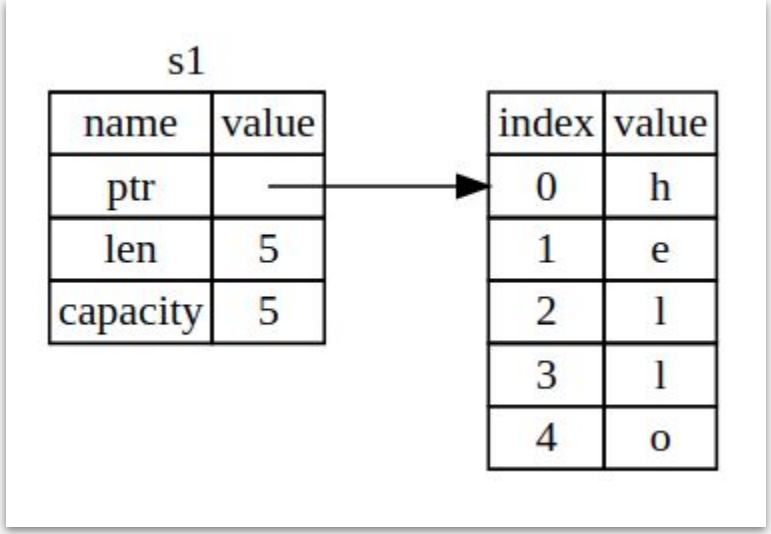
Smart pointers are usually implemented using structs.

- In contrast to ordinary struct, smart pointers implement the Deref and Drop traits.

This lecture will include:

- Implement Deref and Drop to create custom Smart Pointers.
- Box<T> for allocating values on the heap.
- Rc<T>, a reference counting type that enables multiple ownership.

# Box<T>



# Storing Data into Heap

Boxes allow us to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.

- No performance overhead.
- Use cases:
  - When we have a type whose size is unknown at compile time but we want a value of that type requires an exact size.
  - When we have a large amount of data and we want to transfer ownership but don't want to copy the data.

```
fn code1() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Just like any owned value, when a box goes out of scope, as **b** does at the end of main, it will be deallocated. The deallocation happens for the box (stored on the stack) and the data it points to (stored on the heap).

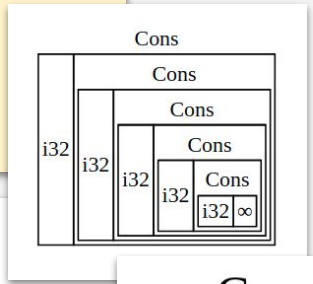
# Recursive Types of Boxes

At compile time, Rust needs to know how much space a type takes up. One type whose size can't be known at compile time is a recursive type, where a value can have as part of itself another value of the same type.

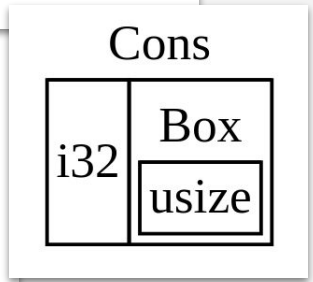
- Boxes have a known size, so by inserting a box (indirection) in a recursive type definition, we can have recursive types.

```
enum List {
    Cons(i32, List),
    Nil,
}
```

```
enum List {
    Cons(i32, Box<List>),
    Nil,
}
```




```
fn code3() {
    //let list = List::Cons(1, List::Cons(2, List::Cons(3, List::Nil)));
    let list = List::Cons(
        1,
        Box::new(List::Cons( 2, Box::new(List::Cons( 3, Box::new(List::Nil))))),
    );
}
```



# Pointer Dereference

# Regular and Box Dereference

A regular reference is a type of pointer, and one way to think of a pointer is as an arrow to a value stored somewhere else.

- We can derefer a reference by using a * before it.

```
fn code4() {
    let x = 5;
    let y = &x;

    println!("x = {} and *y = {}", x, *y);

    //assert_eq!(5, y);
}
```

```
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

```
fn code5() {
    let mut x = 5;
    let mut y = &mut x;

    //assert_eq!(5, y);

    /* let mut a = 10;
    y = &mut a; */
    *y = 10;

    println!("*y = {}", *y);
    println!("x = {}", x);
    //println!("x = {} and *y = {}", x, *y);
}
```

# Custom Smart Pointers

# Deref Trait

Implementing the Deref trait allows us to customize the behavior of the dereference operator, * (as opposed to the multiplication or glob operator).

- By implementing Deref in such a way that a smart pointer can be treated like a regular reference, we can write code that operates on references and use that code with smart pointers too.

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

impl<T> MyBox<T> {
    fn new(x: T, y: T) -> Self {
        Self { x, y }
    }
}
```

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        //&self.x
        &self.y
    }
}
```

```
fn code7() {
    let a = 5;
    let b = 10;
    let y = MyBox::new(a, b);

    assert_eq!(5, a);

    assert_eq!(5, *y);
    //*(y.deref())
}
```

```
without imple of Deref:
    error[E0614]: type `MyBox<{integer}>` cannot be dereferenced
       --> src/main.rs:126:19
        |
    126 |     assert_eq!(5, *y);
        |                   ^^
```

# Implicit Deref Coercions

**Deref coercion** is a convenience that Rust performs on arguments to functions and methods.

- Deref coercion works only on types that implement the Deref trait (DerefMut for mutable reference).
- For example, deref coercion can convert &String to &str because String implements the Deref trait such that it returns str.

```
fn hello(name: &str) {
    println!("Hello, {}!", name);
}

fn code8() {
    let m = MyBox::new(String::from("Rust"), String::from("C/C++"));

    hello(&(*m)[..]);
    //hello(&m);
}
```

Rust does deref coercion when it finds types and trait implementations in three cases:

- From &T to &U when T: Deref<Target=U>
- From &mut T to &mut U when T: DerefMut<Target=U>
- From &mut T to &U when T: Deref<Target=U>

# Drop Trait

Drop trait lets us customize what happens when a value is about to go out of scope. Compiler will call it automatically but would not decide its behavior.

- Release resources like files or network connections.
- When a Box<T> is dropped it will deallocate the space on the heap that the box points to.

```
struct CustomSmartPointer {
    data: String,
}

impl Drop for CustomSmartPointer {
    fn drop(&mut self) {
        println!("Dropping ... `{}`!", self.data);
    }
}
```

```
fn code9() {
    let c = CustomSmartPointer {
        data: String::from("c instance"),
    };
    let d = CustomSmartPointer {
        data: String::from("d instance"),
    };
    println!("instances created.");
}
```

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.60s
     Running `target/debug/drop-example`
instances created.
Dropping ... `d instance`!
Dropping ... `c instance`!
```

# Early Memory Drop

Rust doesn't let us call the Drop trait's drop method manually; instead we have to call the std::mem::drop function provided by the standard library if we want to force a value to be dropped before the end of its scope.

```
fn code10() {
    let c = CustomSmartPointer {
        data: String::from("c instance"),
    };
    println!("instance created.");
    //c.drop();
    drop(c);
    println!("instance dropped early.");
}
```

Rust would not let us call drop explicitly because Rust would also automatically call drop on the value at the end of main.

- This would lead to double free error.

```
$ cargo run
   Compiling drop-example v0.1.0 (file:///projects/drop-example)
    Finished dev [unoptimized + debuginfo] target(s) in 0.73s
     Running `target/debug/drop-example`
instance created.
Dropping ... `c instance`!
instance dropped early.
```

# Rc<T>:
# Reference Counter

# Reference Counted Smart Pointer

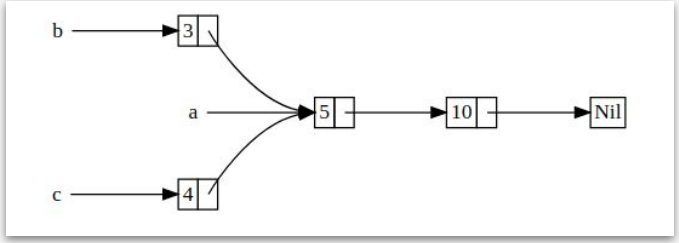
There are cases when a single value may have multiple owners.

- For example, in graph data structures, multiple edges might point to the same node, and that node is conceptually owned by all of the edges that point to it. A node shouldn't be cleaned up unless it doesn't have any edges pointing to it.

To enable multiple ownership, Rust has a type called Rc<T>, which is an abbreviation for reference counting.

- The Rc<T> type keeps track of the number of references to a value which determines whether or not a value is still in use.
  - If there are zero references to a value, the value can be cleaned up without any references becoming invalid.

Note that Rc<T> is only for use in single-threaded scenarios.



# Rc<T> to Share Data

```
enum List {
    Cons(i32, Box<List2>),
    Nil,
}
```

```
fn code11() {
    let a = List::Cons(5, Box::new(List::Cons(10, Box::new(List::Nil))));
    let b = List::Cons(3, Box::new(a));
    let c = List::Cons(4, Box::new(a));
}
```

```
use std::rc::Rc;

enum List {
    Cons(i32, Rc<List3>),
    Nil,
}
```

```
fn code11() {
    let a = Rc::new(List::Cons(
        5,
        Rc::new(List::Cons(10, Rc::new(List::Nil))),
    ));
    let b = List::Cons(3, Rc::clone(&a));
    let c = List::Cons(4, Rc::clone(&a));
}
```

# < Smart Pointers />