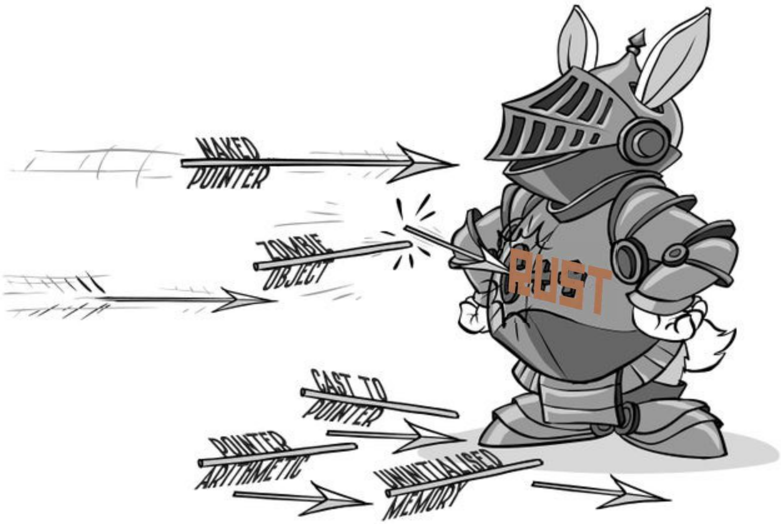


Smart Pointers

What is a Pointer in Rust?

- A *pointer* is a general concept for a **variable that holds a memory address**.
- That memory address refers to or points to some other data in memory.
- The most common pointer in Rust is the *reference*.

What is a Reference in Rust?

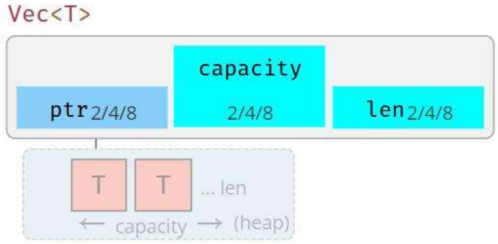
- References simply ***borrow*** **the values they point to**, meaning that they don't have ownership over the values.

- References don't have any special capabilities, which also means they don't have much overhead unlike smart pointers.

What are Smart Pointers?

- Smart pointers are data structures that act like a pointer, but have *metadata* and *extra capabilities* tacked on.
- In many cases, **smart pointers own the data** they point to unlike references which simply borrow the values.
- Smart pointers we already have seen so far include `String` and `Vec<T>`.
- `Box<T>`, `Rc<T>`, `Arc<T>` , `Mutex<T>` , etc.

Observe how `String` differs from `&str` and `&[char]`.



Regular *growable array* vector of single type.

Built-in Fat Pointer Types

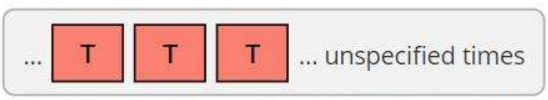
Rust has **two built-in fat pointer types**: types that act as pointers, but which hold additional information about the thing they are pointing to.

- Slice reference (`&[T]`): a reference to a subset of some contiguous collection of values.
- Trait object: a reference to some item that implements a particular trait.

Slice Reference (&[T])

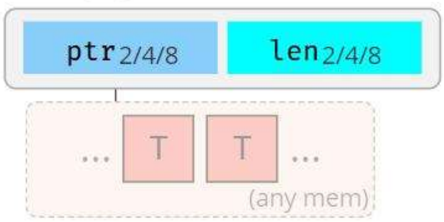
- It's built from a (non-owning) simple pointer, together with a **length** field.
- The type of a slice is written as `&[T]` – a reference to `[T]`, which is the notional type for a contiguous collection of values of type `T`.

`[T]`



Slice type of unknown-many elements. Neither `Sized` (nor carries `len` information), and most often lives behind reference as `&[T]`. ↓

`&'a [T]`

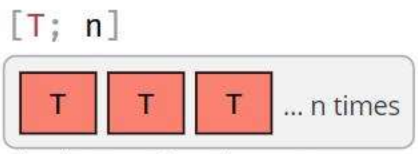


Regular **slice reference** (i.e., the reference type of a slice type `[T]`) ↑ often seen as `&[T]` if `'a` elided.

Slice Reference (cont'd)

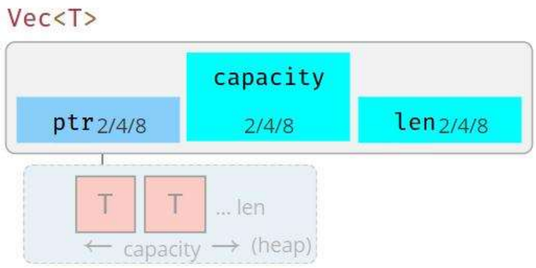
The notional type `[T]` can't be instantiated, but there are three common containers that embody it: Slice Reference, Array and Vector.

- Array



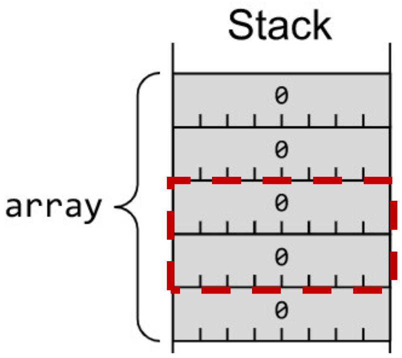
Fixed array of n elements.

- Vector

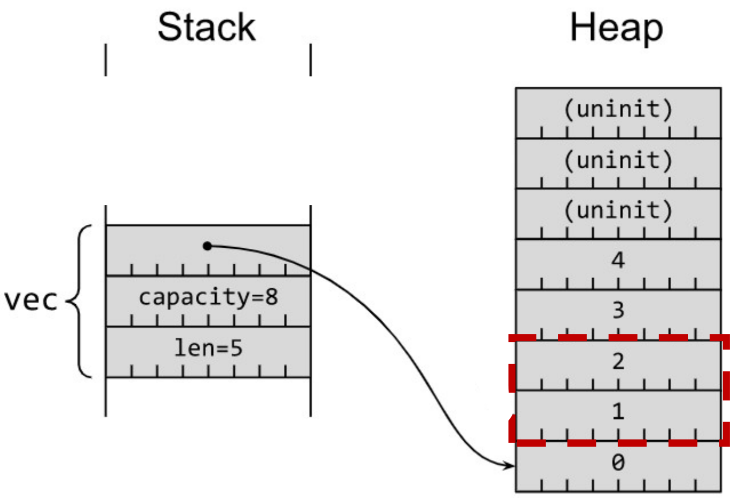


Regular *growable array* vector of single type.


```rust
let array: [u64; 5] = [0u64; 5];
let slice: &[u64] = &array[1..3];
```




```rust
let mut vec = Vec::<u64>::with_capacity(8);
for i in 0..5 { vec.push(i); }
let slice: &[u64] = &vec[1..3];
```



Trait Object

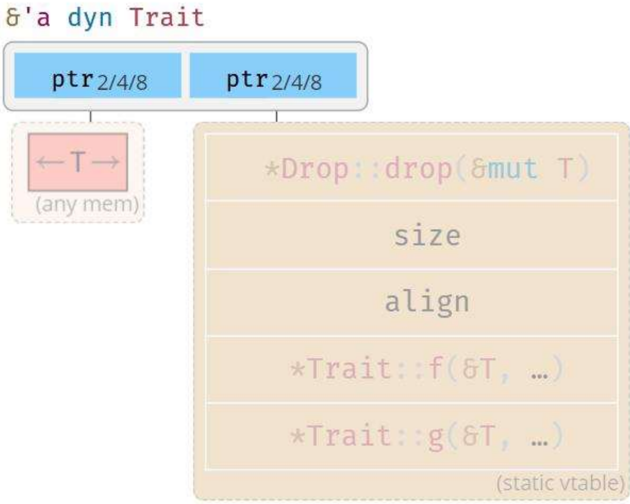
A trait object is a reference to some item that implements a particular trait, represented as `&dyn SomeTrait`.

```
trait Animal {
  fn play(&self);
}

struct Dog; impl Animal for Dog {…}
struct Cat; impl Animal for Cat {…}

fn play(animal: &dyn Animal) {
    animal.play();
}

play(&Dog); play(&Cat);
```



Meta points to vtable, where `*Drop::drop()`, `*Trait::f()`, ... are pointers to their respective `impl` for `T`.

Trait Object

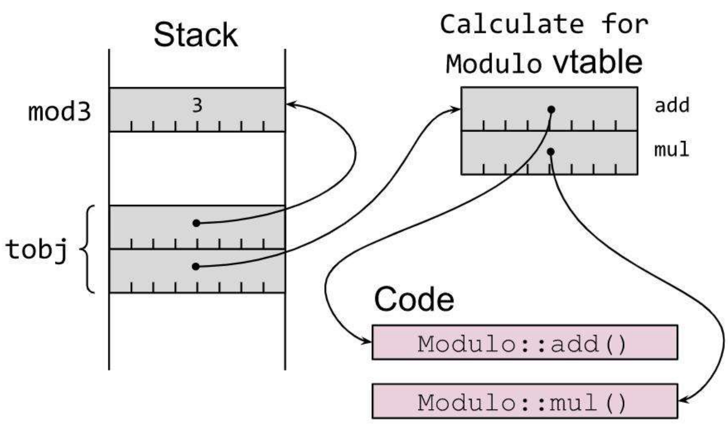
```
trait Calculate {
  fn add(&self, l: u64, r: u64) -> u64;
  fn mul(&self, l: u64, r: u64) -> u64;
}

struct Modulo(pub u64);

impl Calculate for Modulo {
  fn add(&self, l: u64, r: u64) -> u64 {
    (l + r) % self.0
  }
  fn mul(&self, l: u64, r: u64) -> u64 {
    (l * r) % self.0
  }
}

let mod3 = Modulo(3);
```

```
let tobj: &dyn Calculate = &mod3;
```



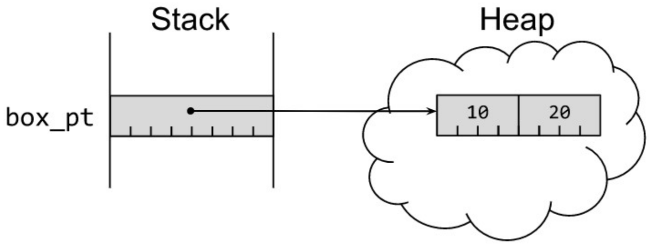
Box<T>

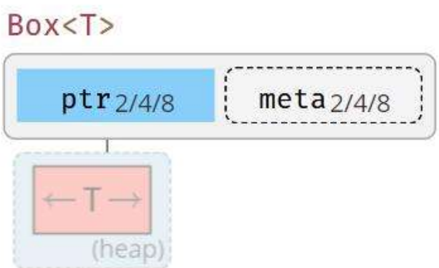
The std::boxed::Box

- The `Box<T>` is a smart pointer that allows you to **allocate values on the heap**. What remains on the stack is the pointer to the heap data.

```
struct Point { x: i32, y: i32 }

let box_pt: Box<Point> = Box::new(
    Point { x: 10, y: 20 }
);
```





For some T stack proxy may carry meta ↑ (e.g., Box<[T]>).

Using Box<T> like a Reference

- Following the pointer to the value with * (dereference operator).

```
let x = 5;
let y = &x;
assert_eq!(*y, 5);

let box_t = Box::new(x);
assert_eq!(*box_t, 5);

println!("x = {x}, y = {y}, box_t = {box_t}");
```


When to Use Box<T>

- When you want to use a value of a type **whose size can't be known at compile time**. For example, recursive types such as *cons lists*.

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

- When you want to **transfer (i.e., move) a large amount of data** without excessive copying.

- When you want to own a *trait object*, i.e., value of a type that implements a particular trait rather than being of a specific type.

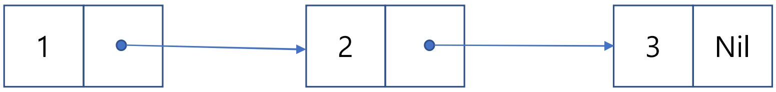
```
let animal: Box<dyn Animal> = random_animal(0.4);
```

Cons List: A Recursive Data Type

- A cons list consists of a pair **(head, tail)**, where a tail is itself a cons list.

head　　　tail

```
(1, (2, (3, Nil)))
```



- How to define cons list in Rust?

```
// Java
class Node {
    int value;
    Node node;
};
```

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



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

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

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:5:5
  |
5 |     enum List {
  |     ^^^^^^^^^
6 |         Nil,
7 |         Cons(i32, List),
  |                   ---- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
7 |         Cons(i32, Box<List>),
  |                   ++++    +
```

store the value indirectly by storing a pointer to the value instead

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

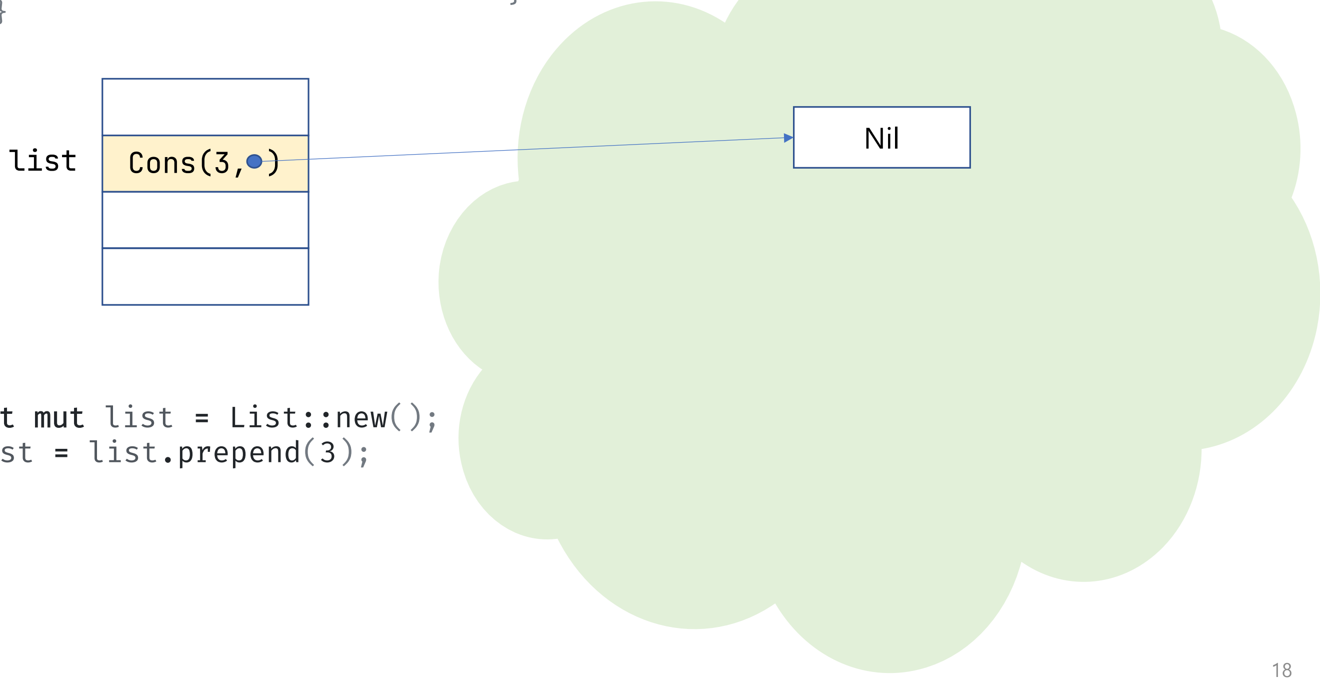
```rust
fn prepend(self, elem: T) -> List<T> {
    Cons(elem, Box::new(self))
}
```


```rust
let mut list = List::new();
```



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

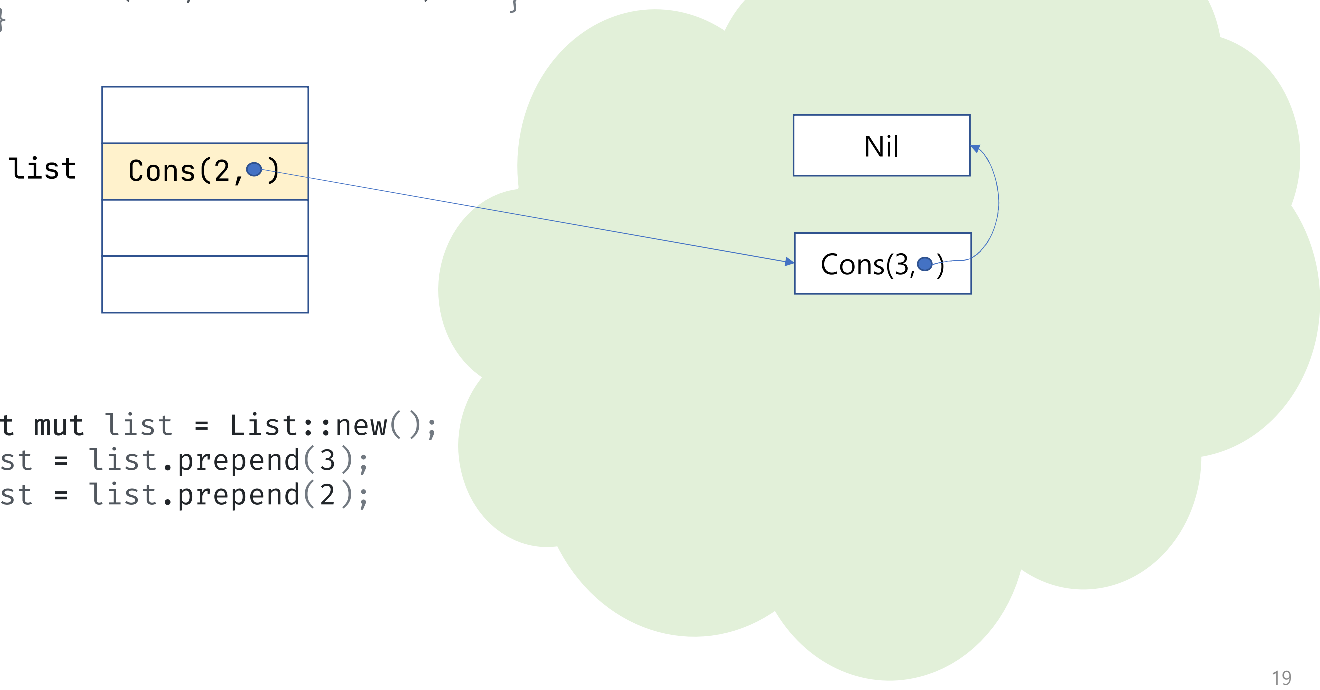
```rust
fn prepend(self, elem: T) → List<T> {
    Cons(elem, Box::new(self))
}
```




```rust
let mut list = List::new();
list = list.prepend(3);
```

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

```
fn prepend(self, elem: T) -> List<T> {
    Cons(elem, Box::new(self))
}
```

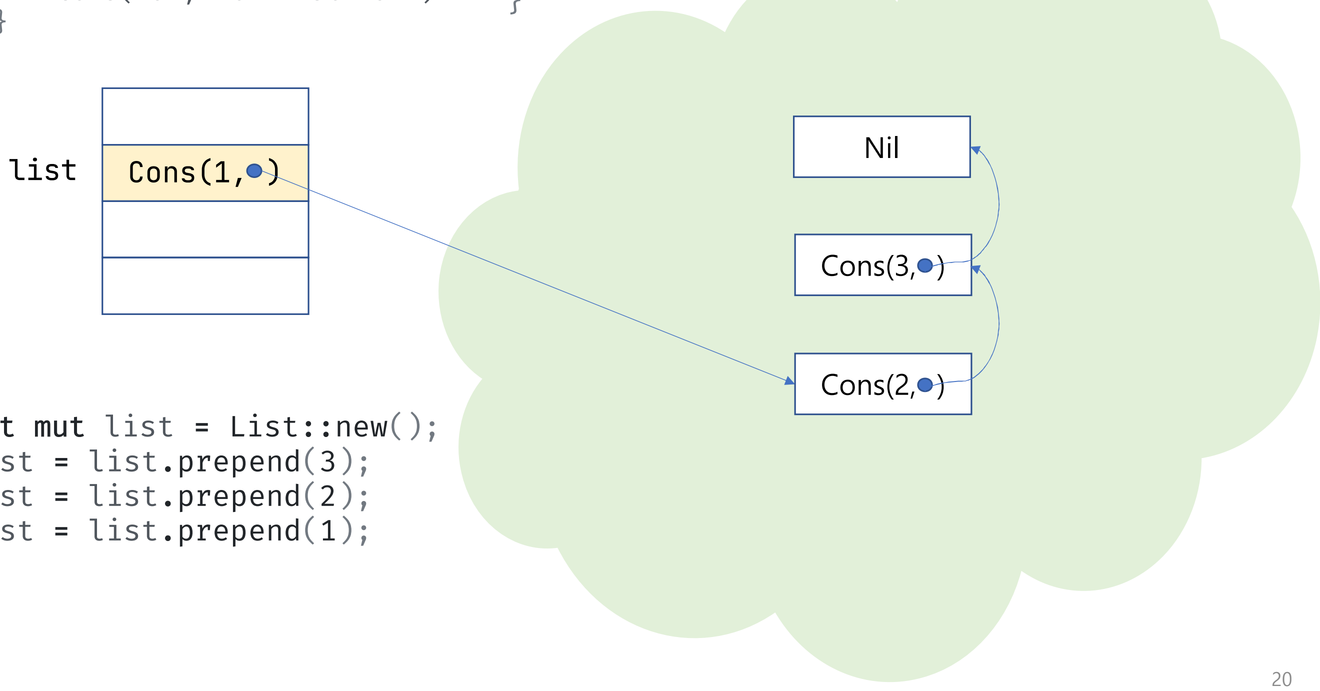



```
let mut list = List::new();
list = list.prepend(3);
list = list.prepend(2);
```



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

```
fn prepend(self, elem: T) -> List<T> {
    Cons(elem, Box::new(self))
}
```




```
let mut list = List::new();
list = list.prepend(3);
list = list.prepend(2);
list = list.prepend(1);
```

Custom Smart Pointers

Defining Our Own Smart Pointer

```
#[derive(Debug)]
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) → MyBox<T> {
        MyBox(x)
    }
}

let x = MyBox::new(String::from("hello"));

let y = &*x; // error: type `MyBox<{String}>`
             // cannot be dereferenced
```

We need a way to access
the value inside smart pointer!

How to solve?

Defining Our Own Smart Pointer (Cont'd)

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

    fn as_ref(&self) -> &T {
        &self.0
    }
};
```

```
let x = Box::new(String::from("hello"));
let y = &*x; // OK
let len = x.len(); // OK

let x = &String::from("hello");
let y = &*x; // OK
let len = x.len(); // OK
```

```
let x = MyBox::new(String::from("hello"));
// let y = &*x; // Error!
let y = x.as_ref();
// let len = x.len(); // Error!
let len = x.as_ref().len();
```

Oops!

Need to handle our smart pointers differently from normal references?

Smart Pointer Implementation

- Smart pointers are usually implemented as **structs**.
- But unlike regular structs, they implement `Deref` and `Drop` traits.
- Smart pointers are general design pattern used frequently in Rust.
  And also many libraries implement their own smart pointers.

std::ops::Deref

- The `Deref` trait is used for effective dereferencing, enabling easy access to the data stored behind the smart pointers.

- The `Deref` trait *allows smart pointer to be treated like references*, so you can write codes which works with either references or smart pointers.

std::ops::Drop

- The `Drop` trait allows you to customize the code that is run when the smart pointer goes out of scope.

- The `Drop` trait implementation usually cleans up resources that are no longer being used by a program.

Treating a Type Like a Reference by Implementing the `std::ops::Deref` Trait

- The `Deref` trait has one method `deref` that borrows self and returns a reference to the inner data.

```
pub trait Deref {
  type Target: ?Sized;

  fn deref(&self) -> &Self::Target;
}
```

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

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

  fn deref(&self) -> &Self::Target {
      &self.0
  }
}
```



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

  fn deref(&self) -> &Self::Target {
      &self.0
  }
}

let x = MyBox::new(String::from("hello"));
let y = &*x; // OK
let len = x.len(); // OK
```

\(◦'‿'◦)/

```
let x = Box::new(String::from("hello"));
let y = &*x; // OK
let len = x.len(); // OK

let x = &String::from("hello");
let y = &*x; // OK
let len = x.len(); // OK
```

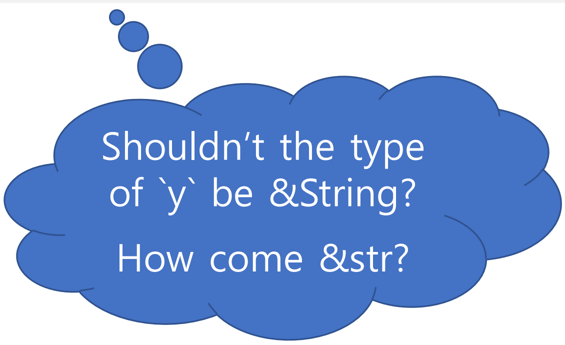
How dereferencing works?

- Without the `Deref` trait, the compiler can only dereference `&` references.
- Note that the dereference operator `*` is replaced with a call to the `deref` method and then a call to the `*` operator just once, each time we use a `*` in our code.

```
*y === *(y.deref())
```


What is the type of variable y?

```rust
let x = &String::from("hello");
let y: &str = &*x; // type of y???
```



Deref Coercion

If you have a type `U`, and it implements `Deref<Target=T>`, values of `&U` will automatically coerce to a `&T`, *if needed*.

impl Deref<Target = str> for String { … }

`&String ⇒ &str`

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}
let owned = String::from("Rust");
hello(&owned); // &String => &str deref coercion
hello(&(*owned)[..]); // in case we don't have the Deref coercion
```


Implicit "Deref Coercions" with Functions and Methods

- **Deref coercion** converts a reference to `Deref` type† into a reference to another type. For example, `&String` => `&str`.
- **Deref coercion** happens automatically when the actual parameters doesn't match with the formal parameters in a function or method.
- A *sequence* of calls to the `deref` method converts the actual argument's type into the formal parameter's type.
- Compiler uses `Deref::deref` as many times as necessary to get a reference to match the parameter's type.

† By Deref type, I mean a type that implements the Deref trait.

Deref Coercion in Action

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

let m = MyBox::new(String::from("Rust"));
hello(&m); // &Box<String> => &String => &str in sequence

hello(&(*m.as_ref())[..]);
// in case we don't have the Deref coercion
```


How Deref Coercion Interacts with Mutability

- Use the `Deref`/`DerefMut` trait to override the `*` operator on immutable/mutable references, respectively.

- *Deref coercion* occurs in three cases:
  - From `&U` to `&T` when `U: Deref<Target=T>`
  - From `&mut U` to `&mut T` when `U: DerefMut<Target=T>`
  - From `&mut U` to `&T` when `U: Deref<Target=T>`

  Note that immutable references will never coerce to mutable references.

Running Code on Cleanup with the Drop Trait

- The second trait important to the smart pointer pattern is `Drop`, which lets you customize what happens when a value is about to go out of scope.
- It can be **used to release resources** like files or network connections.
- For example, when a `Box<T>` is dropped it will deallocate the space on the heap that the box points to.

```
pub trait Drop {
    fn drop(&mut self);
}
```


Drop in Action

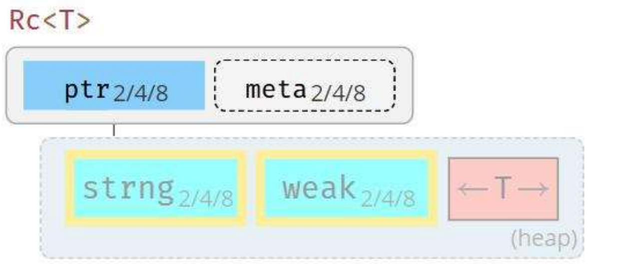
Drop Order:

- For structs, it’s the same order that they’re declared.
- Unlike for structs, local variables are dropped in reverse order.

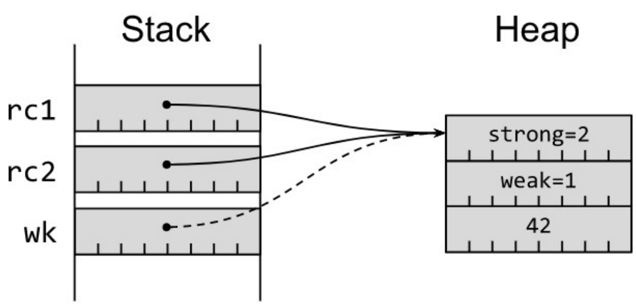
Rc<T>

std::rc::Rc, Reference Counted Smart Pointer

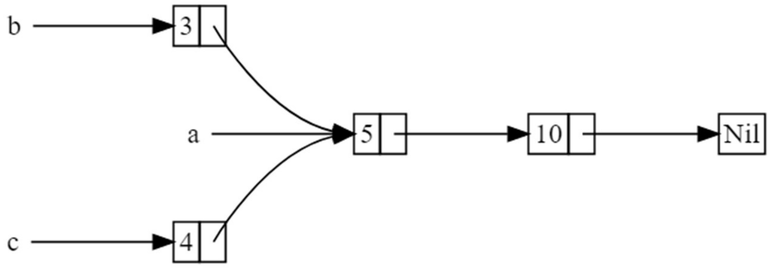
- Single-threaded **reference-counting smart pointers** to support multiple owners.
- Rc<T> allows a piece of data to have multiple owners by keeping track of the owners, and once there are no more owners, it cleans up the data.



Share ownership of T in same thread. Needs nested Cell or RefCell to allow mutation. Is neither Send nor Sync.



Using Rc to Shared Data




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

use crate::List::{Cons, Nil};

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


Compile Error!

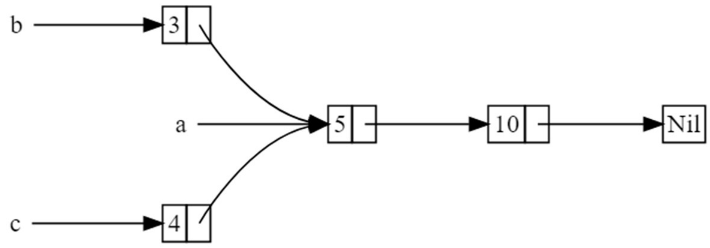
```
error[E0382]: use of moved value: `a`
  --> src/main.rs:14:30
   |
12 |     let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
   |         - move occurs because `a` has type `List`, which does not
implement the `Copy` trait
13 |     let b = Cons(3, Box::new(a));
   |                              - value moved here
14 |     let c = Cons(4, Box::new(a));
   |                              ^ value used here after move
```

Change Box<T> to Rc<T>

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

use List::{Cons, Nil};
use std::rc::Rc;

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



Cloning an Rc<T> Increases the Reference Count

```rust
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a));         // 1
    let b = Cons(3, Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a));         // 2
    {
        let c = Cons(4, Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a));     // 3
    }
    println!("count after c goes out of scope = {}", Rc::strong_count(&a)); // 2
}
```

What if we want to modify the content of Rc?

```
--> src/smart_pointers/bin\rc_refcell.rs:89:18
   |
   |             Cons(ref mut head, _) => {
   |                  ^^^^^^^^^^^^ cannot borrow as mutable
   |
   = help: trait `DerefMut` is required to modify through a dereference,
but it is not implemented for `std::rc::Rc<rc_refcell::rc_type::List>`
```

```
match *a {
    Cons(ref mut head, _) => {
        *head = 42;
    }
    Nil => {}
}

println!("a after = {a:?}");
```


Interior Mutability

Onwership Rule Again

Rust memory safety is based on this rule: Given an object `T`, it is only possible to have one of the following:

- Having several immutable references (`&T`) to the object (also known as **aliasing**), aka *shared references*
- Having one mutable reference (`&mut T`) to the object (also known as **mutability**), aka *unique reference*.

However, sometimes it is required to have multiple references to an object and yet mutate it.

Inherited Mutability vs. Interior Mutabiliy

- **Sharable mutable containers** (i.e. cell types in `std::cell`) come in two flavors: `Cell<T>` and `RefCell<T>`
- Cell types allow mutation through shared references (`&T`).
- `Cell<T>` and `RefCell<T>` provide '***interior mutability***', as compared to typical '***inherited mutability***'.
- `Cell<T>` implements interior mutability by moving values in and out of the `Cell<T>`. To use references instead of values, one must use the `RefCell<T>` type.
- Both cell types are not thread-safe.
  - Consider using `RwLock<T>` or `Mutex<T>` if you need shared mutability in a multi-threaded situation.

std::cell::Cell

- For `Copy` data, the `get` method retrieves the current interior value.
- For `Default` data, the `take` method replaces the current interior value with `Default::default()` and returns the replaced value.
- For all types,
  - the `replace` method replaces the current interior value and returns the replaced value and
  - the `into_inner` method consumes the `Cell<T>` and returns the interior value.
  - the `set` method replaces the interior value, dropping the replaced value.


```
#[derive(Debug)]
enum List {
    Cons(Cell<i32>, Rc<List>),
    Nil,
}

use self::List::{ Cons, Nil };

let mut a = Rc::new(Cons(Cell::new(5),
                    Rc::new(Cons(Cell::new(10), Rc::new(Nil)))));
println!(“a after = {a:?}");

match *a {
    Cons(ref head, _) => {
        head.replace(42);
    }
    Nil => {}
}

println!("a after = {:?}", a);
```

std::cell::RefCell

- It's very common to put a `RefCell<T>` inside shared pointer types to reintroduce mutability – interior mutability.
- The normal **borrow checks are moved from compile-time to run-time**.
- The `RefCell` itself doesn't implement any smart pointer logic. Instead,
  - `borrow()` returns a `Ref<T>` and
  - `borrow_mut()` returns a `RefMut<T>`,

  each of which actually implement the smart pointer logic.


```
use std::cell::{ RefCell, RefMut };
use std::collections::HashMap;
use std::rc::Rc;

let shared_map: Rc<RefCell<_>> = Rc::new(RefCell::new(HashMap::new()));
// Create a new block to limit the scope of the dynamic borrow
{
    let mut map: RefMut<_> = shared_map.borrow_mut();
    map.insert("africa", 92388);
    map.insert("kyoto", 11837);
    map.insert("piccadilly", 11826);
    map.insert("marbles", 38);
}

// Note that if we had not let the previous borrow of the cache fall out
// of scope then the subsequent borrow would cause a dynamic thread panic.
// This is the major hazard of using `RefCell`.
let total: i32 = shared_map.borrow().values().sum();
println!("{total}");
```

When to choose interior mutability

(See https://doc.rust-lang.org/std/cell/)

- Introducing mutability 'inside' of something immutable
- Implementation details of logically-immutable methods
  - Logically immutable, but implementation details may exploit mutation "under the hood".
- Mutating implementations of `Clone`
  - This is simply a special case of the previous: hiding mutability for operations that appear to be immutable.

```
pub trait Clone: Sized {
    fn clone(&self) -> Self;
```


Introducing mutability 'inside' of something immutable

```
use std::cell::{ RefCell, RefMut };
use std::collections::HashMap;
use std::rc::Rc;

let shared_map: Rc<RefCell<_>> = Rc::new(RefCell::new(HashMap::new()));
// Create a new block to limit the scope of the dynamic borrow
{
    let mut map: RefMut<_> = shared_map.borrow_mut();
    map.insert("africa", 92388);
    map.insert("kyoto", 11837);
    map.insert("piccadilly", 11826);
    map.insert("marbles", 38);
}

// Note that if we had not let the previous borrow of the cache fall out
// of scope then the subsequent borrow would cause a dynamic thread panic.
// This is the major hazard of using `RefCell`.
let total: i32 = shared_map.borrow().values().sum();
println!("{total}");
```