Rust Fundamentals

Basics of Rust
Part VI

AZZAM S.A

RustCourse
Nov. 3th, 2023

Azzam S.A

OSS devotee, speaker, and teacher.

Open sourceror. Namely Rust, Python, and Emacs.

azzamsa azzamsyawqi azzamsa.com

Follow along!

- Rust Playground
- Exercises

▶ Show hints

Standard Library

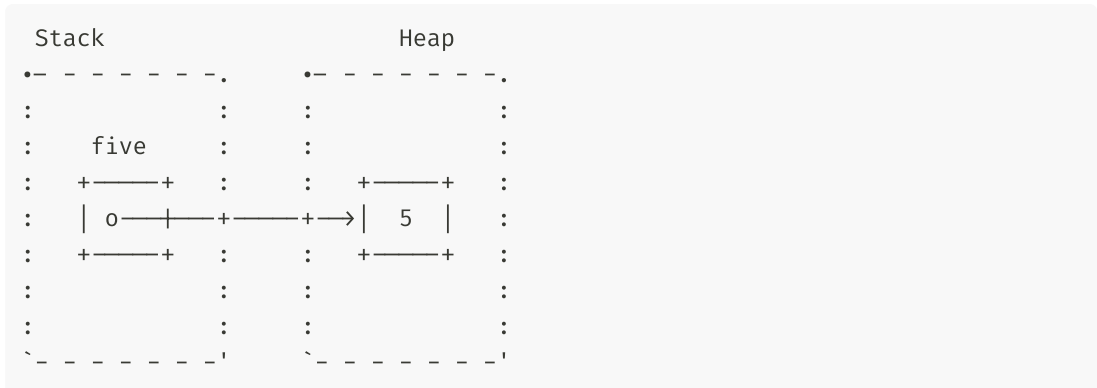
The common vocabulary types include:

- Box<T>: for allocating values on the heap.
- Rc<T>: a reference counting type that enables multiple ownership.

Box

- `Boxes` allow you to store data on the heap rather than the stack.

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



When to use `Box`?

- When the size of a type is unknown at compile time, but it's needed in a context that requires a fixed size.
- When transferring ownership of a large data chunk without copying it.
- When you want to own a value based on a specific trait, not a particular type.

```
fn main() {
    let five = Box::new(5);
    println!("five: {}", *five);
}
```

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

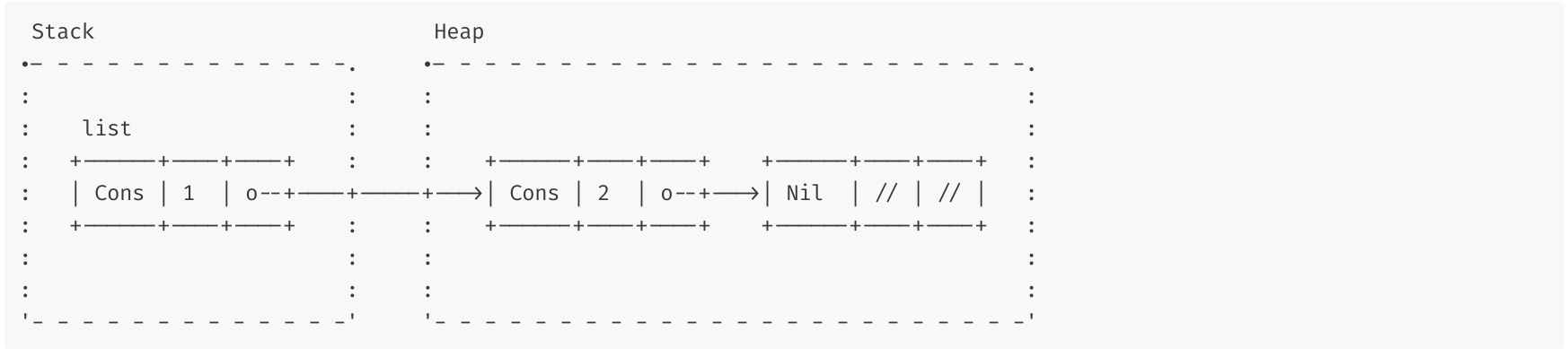
Box with Recursive Data Structures

Recursive data types or data types with dynamic sizes need to use a `Box`:

```
#[derive(Debug)]
enum List<T> {
    Cons(T, Box<List<T>>),
    Nil,
}

fn main() {
    let list: List<i32> = List::Cons(1, Box::new(List::Cons(2, Box::new(List::Nil))));
    println!("{list:?}");
}
```

- Recursive types pose an issue because, at compile time, Rust needs to know how much space a type takes up.
- The cons list isn't a commonly used data structure in Rust. Most of the time, `Vec\<T\>` is a better choice to us.


Stack
Heap
list
Cons | 1 | o
Cons | 2 | o
Nil | // | // |

The compiler is smart enough

Even if you missed the `Box`, the compiler is here for you!

```
error[E0072]: recursive type `List` has infinite size
 --> src/main.rs:2:1
  |
2 | enum List<T> {
  | ^^^^^^^^^^^^
3 |     Cons(T, List<T>),
  |             ------- recursive without indirection
  |
help: insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle
  |
3 |     Cons(T, Box<List<T>>),
  |             ++++       +
```

- "Indirection" means that instead of storing a value directly, we should change the data structure to store the value indirectly by storing a pointer to the value instead.

Rc (Reference Counted)

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

fn main() {
    let a = Rc::new(10);
    let b = Rc::clone(&a);

    println!("a: {a}");
    println!("b: {b}");
}
```

Exersices

- `box1.rs`
- `rc1.rs`

`box1.rs`

```
#[derive(PartialEq, Debug)]
pub enum List {
-   Cons(i32, List),
+   Cons(i32, Box<List>),
    Nil,
}

pub fn create_empty_list() -> List {
+    List::Nil
}

pub fn create_non_empty_list() -> List {
+    List::Cons(10, Box::new(List::Nil))
}
```

`rc1.rs`

`Rc::clone()` https://doc.rust-lang.org/std/rc/struct.Rc.html#method.clone

> This creates another pointer to the same allocation, increasing the strong reference count.

```
+    let saturn = Planet::Saturn(Rc::clone(&sun));
     println!("reference count = {}", Rc::strong_count(&sun)); // 7 references
     saturn.details();

+    let uranus = Planet::Uranus(Rc::clone(&sun));
     println!("reference count = {}", Rc::strong_count(&sun)); // 8 references
     uranus.details();

+    let neptune = Planet::Neptune(Rc::clone(&sun));

+    drop(earth);
     println!("reference count = {}", Rc::strong_count(&sun)); // 3 references

+    drop(venus);
     println!("reference count = {}", Rc::strong_count(&sun)); // 2 references

+    drop(mercury);
     println!("reference count = {}", Rc::strong_count(&sun)); // 1 reference
```

Generics

Generic Data Types

You can use generics to abstract over the concrete field type:

```
#[derive(Debug)]
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 };
    println!("{integer:?} and {float:?}");
}
```

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

fn main() {
    let p = Point { x: 5, y: 10.0 };
}
```

Gives a compiler error:

```
error[E0308]: mismatched types
 --> src/main.rs:8:30
  |
8 |     let p = Point { x: 5, y: 10.0 };
  |                              ^^^^ expected integer, found floating-point number

For more information about this error, try `rustc --explain E0308`.
error: could not compile `playground` (bin "playground") due to previous error
```

Why? 🤔

Generic Methods

You can declare a generic type on your `impl` block:

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

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.0  // + 10
    }
}

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

Monomorphization

Generic code is turned into non-generic code based on the call sites:

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

Behaves as if you wrote

```
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);
}
```

Exercises

`generics1.rs`

```
fn main() {
-   let mut shopping_list: Vec<?> = Vec::new();
+   let mut shopping_list: Vec<&str> = Vec::new();
    shopping_list.push("milk");
}
```

`generics2.rs`

```
- struct Wrapper {
+ struct Wrapper<T> {
-     value: u32,
+     value: T,
}

- impl Wrapper {
+ impl<T> Wrapper<T> {
-     pub fn new(value: u32) -> Self {
+     pub fn new(value: T) -> Self {
        Wrapper { value }
    }
}
```

Credits 🌟

- Mo's (mo8it) Comprehensive Rust 🦀
- rustlings 🦀