Lecture 4: Rewind

Barinov Denis

March 7, 2023

barinov.diu@gmail.com

`String` and `&str`

`String`[1]

A Rust way to store a string.

[1] `String` documentation

`String`[1]

A Rust way to store a string.

- It's **UTF-8–encoded**.

[1] `String` documentation

`String`[1]

A Rust way to store a string.

- It's **UTF-8–encoded**.
- Growable like a `Vec`. It also made up of three components: a pointer to some bytes, a length, and a capacity. This even gives us many functions same to `Vec`.

[1] `String` documentation

`String`[1]

A Rust way to store a string.

- It's **UTF-8–encoded**.
- Growable like a `Vec`. It also made up of three components: a pointer to some bytes, a length, and a capacity. This even gives us many functions same to `Vec`.
- UTF-8 is a variable-width character encoding, so you cannot index it since it's UTF-8. To find N-th symbol, you should iterate over string, parsing code points.

[1] `String` documentation

String API

```
struct String {
    vec: Vec<u8>,
}

impl String {
    fn new() -> String;
    fn with_capacity(capacity: usize) -> String;
    fn from_utf8(vec: Vec<u8>) -> Result<String, FromUtf8Error>;
    fn from_utf16(v: &[u16]) -> Result<String, FromUtf16Error>;
    fn into_bytes(self) -> Vec<u8>;
    fn as_bytes(&self) -> &[u8];
}
```

`String` in depth

What will this code print?

```
let s = String::from(""); // hello
println!("{}", s.len());
```

`String` in depth

What will this code print?

```
let s = String::from(""); // hello
println!("{}", s.len());
```

This outputs 12, since `.len()` gives count of *bytes* in string.

char type

```
let mut chars = "é".chars();
// U+00e9: 'latin small letter e with acute'
assert_eq!(Some('\u{00e9}'), chars.next());
assert_eq!(None, chars.next());

let mut chars = "é".chars();
// U+0065: 'latin small letter e'
assert_eq!(Some('\u{0065}'), chars.next());
// U+0301: 'combining acute accent'
assert_eq!(Some('\u{0301}'), chars.next());
assert_eq!(None, chars.next());
```

`char` type

The size of `char` is always 4 bytes:

```
assert_eq!(std::mem::size_of::<char>(), 4);
```

&str

`&str` is a slice type of `String`, similar to `std::string_view`. Just like:

```
let vec = vec![1, 2, 3, 4];
let vec_slice = &vec[1..3]; // &[2, 3]
let s = String::from("hello");
let s_slice = &s[1..3]; // "el"
```

&str

Ok, let's take a UTF-8 slice!

```
let s = String::from(""); // hello
let s_slice = &s[1..3];
```

&str

Ok, let's take a UTF-8 slice!

```
let s = String::from(""); // hello
let s_slice = &s[1..3];
// thread 'main' panicked at 'byte index 1 is
// not a char boundary; it is inside '' // h
// (bytes 0..2) of ``' // hello
```

&str

Ok, let's take a UTF-8 slice!

```
let s = String::from(""); // hello
let s_slice = &s[1..3];
// thread 'main' panicked at 'byte index 1 is
// not a char boundary; it is inside '' // h
// (bytes 0..2) of ``' // hello
```

That means &str also have a UTF-8 invariant checked at runtime.

&str

As a string slice, `&str` have most functions `String` have:

```
fn as_bytes(&self) -> &[u8];
fn chars(&self) -> Chars<'_>;
fn trim(&self) -> &str;
// And so on
```

&str

All string constants are `&str`.

```
let s: &str = "Hello world!";
let t1 = s.to_string();
let t2 = s.to_owned(); // The same as t1
let t3 = String::from(s); // The same as t1
```

`Box` and `Rc`

Box

We are already familiar with `Box` type. Let's check one advanced function:

```
fn leak<'a>(b: Box<T, A>) -> &'a mut T;
fn into_raw(b: Box<T, A>) -> *mut T;
```

Example:

```
let x = Box::new(41);
let static_ref: &'static mut usize = Box::leak(x);
*static_ref += 1;
assert_eq!(*static_ref, 42);
```

But stop! Rust is the safe language, no memory unsafety, no undefined behavior, what's wrong!?

But stop! Rust is the safe language, no memory unsafety, no undefined behavior, what's wrong!?

In reality, when you're creating global objects or interacting with other languages, you ***have to*** *leak objects. Moreover, it's* ***safe*** *to leak memory, just not good!*

Rc

`Rc` is single-threaded reference-counting pointer. "`Rc`" stands for "Reference Counted".

```
let rc = Rc::new(());
let rc2 = rc.clone(); // Clones Rc, not what inside!
let rc3 = Rc::clone(&rc); // The same
```

`Rc` is dropped when all instances of `Rc` are dropped.

Primary functions:

```
fn get_mut(this: &mut Rc<T>) -> Option<&mut T>;
fn downgrade(this: &Rc<T>) -> Weak<T>;
fn weak_count(this: &Rc<T>) -> usize;
fn strong_count(this: &Rc<T>) -> usize;
```

References to the variable inside `Rc` are controlled at runtime:

```
let mut rc = Rc::new(42);
println!("{}", *rc);

*Rc::get_mut(&mut rc).unwrap() -= 41;
println!("{}", *rc);

let mut rc1 = rc.clone();
println!("{}", *rc1);
// thread 'main' panicked at 'called `Option::unwrap()`
// on a `None` value'
// *Rc::get_mut(&mut rc1).unwrap() -= 1;
```

`get_mut` guarantees that it will return mutable reference only if there's only one pointer. If there are more, you won't have a chance to modify `Rc`.

Weak

`Rc` is a **strong** pointer, while `Weak` is a **weak** pointer. Both of them have *ownership over allocation*, but only `Rc` have *ownership over the value inside*:

You can upgrade `Weak` to `Rc`:

```
fn upgrade(&self) -> Option<Rc<T>>;
```

Weak

```
let rc1 = Rc::new(String::from("string"));
let rc2 = rc1.clone();
let weak1 = Rc::downgrade(&rc1);
let weak2 = Rc::downgrade(&rc1);
drop(rc1); // The string is not deallocated
assert!(weak1.upgrade().is_some());
drop(weak1); // Nothing happens
drop(rc2); // The string is deallocated
assert_eq!(weak2.strong_count(), 0);
// If no strong pointers remain, this will return zero.
assert_eq!(weak2.weak_count(), 0);
assert!(weak2.upgrade().is_none());
drop(weak2); // The Rc is deallocated
```

Questions?