Unraveling Strings in Rust: `&str` , `Box<str>` , `OsString` and beyond

Szilárd Parrag (@Axoflow)

Overview

- What's a `String` ?
- Adventures with `String`
- `String` -like types
- FFI (Foreign Function Interface)
- Beyond the basics: `Cow<'a, B>` , `smol_str`

Q: What is a `String` ?

Q: What is a `String` ?

A: Intuitive primitive type

Q: What is a `String` ?

A: Intuitive primitive type

Simple example

```rust
use std::hint::black_box;

fn main() {
  let mut my_string = String::from("Hello ");
  println!("value: {:#?}, len: {}, cap: {}", &my_string, &my_string.len(), &my_string.capacity());

  black_box(my_string.push_str("world"));

  println!("value: {:#?}, len: {}, cap: {}", &my_string, &my_string.len(), &my_string.capacity());

}
```

Q: What is a `String` ?

A: Intuitive primitive type

Simple example

```rust
use std::hint::black_box;

fn main() {
  let mut my_string = String::from("Hello ");
  println!("value: {:#?}, len: {}, cap: {}", &my_string, &my_string.len(), &my_string.capacity());

  black_box(my_string.push_str("world"));

  println!("value: {:#?}, len: {}, cap: {}", &my_string, &my_string.len(), &my_string.capacity());

}
```

```
value: "Hello ", len: 6, cap: 6
value: "Hello world", len: 11, cap: 12
```

█ Q: What is a `String` ?

A: Intuitive primitive type

░░░ Is it?

Q: What is a `String` ?

A: Intuitive primitive type

Is it?

A bit more interesting example

```rust
use std::hint::black_box;

fn main() {
  let mut greeting = String::from("Hello ");
  println!("value: {:#?}, len: {}, cap: {}", &greeting, &greeting.len(), &greeting.capacity());

  black_box(greeting.push_str("⍰"));

  println!("value: {:#?}, len: {}, cap: {}", &greeting, &greeting.len(), &greeting.capacity());
}
```

Q: What is a `String` ?

A: Intuitive primitive type

Is it?

A bit more interesting example

```rust
use std::hint::black_box;

fn main() {
  let mut greeting = String::from("Hello ");
  println!("value: {:#?}, len: {}, cap: {}", &greeting, &greeting.len(), &greeting.capacity());

  black_box(greeting.push_str("⍰"));

  println!("value: {:#?}, len: {}, cap: {}", &greeting, &greeting.len(), &greeting.capacity());
}
```

```
value: "Hello ", len: 6, cap: 6
value: "Hello ⍰", len: 10, cap: 12
```

Q: Common pitfalls?

A: Indexing

How does Python handle it?

```python
some_str = "Hello ⍰"
print(f"value: {some_str}, len: {len(some_str)}")
for c in some_str:
  print(c)
```

Q: Common pitfalls?

A: Indexing

How does Python handle it?

```
some_str = "Hello ⍰"
print(f"value: {some_str}, len: {len(some_str)}")
for c in some_str:
  print(c)
```

```
value: Hello ⍰, len: 7
H
e
l
l
o
 
⍰
```

Q: Common pitfalls?

A: Indexing

How does Python handle it? (1)

```
some_other_str = "y̆es"
print(f"len: {len(some_other_str)}")
for c in some_other_str:
  print(c)

print("---")

for i in range(len(some_other_str)):
  print(f"ind: {i}, value: {some_other_str[i]}")
```

Q: Common pitfalls?

A: Indexing

How does Python handle it? (1)

```python
some_other_str = "y̆es"
print(f"len: {len(some_other_str)}")
for c in some_other_str:
  print(c)

print("---")

for i in range(len(some_other_str)):
  print(f"ind: {i}, value: {some_other_str[i]}")
```

```
len: 4
y
̆
e
s
---
ind: 0, value: y
ind: 1, value: ̆
ind: 2, value: e
ind: 3, value: s
```

Q: Common pitfalls?

A: Indexing

How does Python handle it? (2)

```
surprise_1 = "é"
print(f"len: {len(surprise_1)}")

for i in range(len(surprise_1)):
  print(f"ind: {i}, value: {surprise_1[i]}")

print("---")

surprise_2 = "é"
print(f"len: {len(surprise_2)}")

for i in range(len(surprise_2)):
  print(f"ind: {i}, value: {surprise_2[i]}")
```

Q: Common pitfalls?

A: Indexing

How does Python handle it? (2)

```
surprise_1 = "é"
print(f"len: {len(surprise_1)}")

for i in range(len(surprise_1)):
  print(f"ind: {i}, value: {surprise_1[i]}")

print("---")

surprise_2 = "é"
print(f"len: {len(surprise_2)}")

for i in range(len(surprise_2)):
  print(f"ind: {i}, value: {surprise_2[i]}")
```

```
len: 1
ind: 0, value: é
---
len: 2
ind: 0, value: e
ind: 1, value: ́
```

Q: Common pitfalls?

A: Indexing

Rust `std::string::String` docs says:

> Due to these ambiguities/restrictions, indexing with a usize is simply forbidden:

```
#![allow(unused)]
fn main() {
let s = "hello";

// The following will not compile!
println!("The first letter of s is {}", s[0]);
}
```

```
error[E0277]: the type `str` cannot be indexed by `{integer}`
 --> src/main.rs:6:43
  |
6 | println!("The first letter of s is {}", s[0]);
  |                                           ^ string indices are ranges of `usize`
  |
  = help: the trait `SliceIndex<str>` is not implemented for `{integer}`
  = note: you can use `.chars().nth()` or `.bytes().nth()`
          for more information, see chapter 8 in The Book: <https://doc.rust-lang.org/book/ch08-02-
strings.html#indexing-into-strings>
  = help: the trait `SliceIndex<[_]>` is implemented for `usize`
  = help: for that trait implementation, expected `[_]`, found `str`
  = note: required for `str` to implement `Index<{integer}>`

For more information about this error, try `rustc --explain E0277`.
```

Q: How does Rust save the day?

A: Type system, which is

- strong
- expressive

Time to read some `std` docs!

`std::string`

> A UTF-8-encoded, growable string.

> The `String` type is the most common string type that has ownership over the contents of the string.

> It has a close relationship with its borrowed counterpart, the primitive `str`.

Time to read some `std` docs!

`std::string`

> `Strings` are always valid UTF-8.
> If you need a non-UTF-8 string, consider `OsString`. It is similar, but without the UTF-8 constrain
> Because UTF-8 is a variable width encoding, `Strings` are typically smaller than an array of the
> same `chars`:

```rust
use std::mem;

fn main() {
    // `s` is ASCII which represents each `char` as one byte
    let s: &str = "hello"; // Note: type annotation by me
    assert_eq!(s.len(), 5);

    // A `char` array with the same contents would be longer because
    // every `char` is four bytes
    let s = ['h', 'e', 'l', 'l', 'o'];
    let size: usize = s.into_iter().map(|c| mem::size_of_val(&c)).sum();
    assert_eq!(size, 20);

    // However, for non-ASCII strings, the difference will be smaller
    // and sometimes they are the same
    let s = "⍰⍰⍰⍰⍰";
    assert_eq!(s.len(), 20);

    let s = ['⍰', '⍰', '⍰', '⍰', '⍰'];
    let size: usize = s.into_iter().map(|c| mem::size_of_val(&c)).sum();
    assert_eq!(size, 20);

    println!("Done")
}
```

String is not just `Vec<char>`? ⍰

Time to read some `std` docs!

```rust
pub struct String { /* private fields */ }
```

Time to read some `std` docs!

```rust
pub struct String { /* private fields */ }
```

Let's take a peek under the hood src https://doc.rust-lang.org/src/alloc/string.rs.html#365 - click on source

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

Unraveling Strings in Rust: `&str` , `Box<str>` , `OsString` and beyond

Overview

- What's a `String` ?
- Adventures with `String`
- `String` -like types
- FFI (Foreign Function Interface)
- Beyond the basics: `Cow<'a, B>` , `smol_str`

Q: But I want my `char`s back, can I have them?

Q: But I want my `char`s back, can I have them?

A: Yes, see the `chars` and `char_indices` functions and the primitive `char` type

Time to read some `std` docs (again)!

`chars()`

> Returns an iterator over the [`char`]s of a string slice.
>
> As a string slice consists of valid UTF-8, we can iterate through a
> string slice by [`char`]. This method returns such an iterator.
>
> It's important to remember that [`char`] represents a Unicode Scalar
> Value, and might not match your idea of what a 'character' is.
>
> Iteration over grapheme clusters may be what you actually want.
> This functionality is not provided by Rust's standard library, check crates.io instead.

Notes

- grapheme cluster ~= what a toddler would consider a single unit
- Example: ⍰ (U+1F46A, "Family") is a single `char`

Example

```rust
fn main() {
  let word = "goodbye";

  let count = word.chars().count();
  assert_eq!(7, count);

  let mut chars = word.chars();

  assert_eq!(Some('g'), chars.next());
  assert_eq!(Some('o'), chars.next());
  assert_eq!(Some('o'), chars.next());
  assert_eq!(Some('d'), chars.next());
  assert_eq!(Some('b'), chars.next());
  assert_eq!(Some('y'), chars.next());
  assert_eq!(Some('e'), chars.next());

  assert_eq!(None, chars.next());
}
```

char_indices

Remember, `char`s might not match your intuition about characters:

```rust
fn main() {
    let yes = "y̆es";

    let chars_count = yes.chars().count();
    assert_eq!(4, chars_count);

    let chars_size = yes.len();
    assert_eq!(5, chars_size);


    let mut char_indices = yes.char_indices();

    assert_eq!(Some((0, 'y')), char_indices.next()); // not (0, 'y̆')
    assert_eq!(Some((1, '\u{0306}')), char_indices.next());

    // note the 3 here - the previous character took up two bytes
    assert_eq!(Some((3, 'e')), char_indices.next());
    assert_eq!(Some((4, 's')), char_indices.next());

    assert_eq!(None, char_indices.next());
}
```

Cases

- `pub fn to_uppercase(self) -> ToUppercase` and
- `pub fn to_lowercase(self) -> ToLowercase`

> Returns an iterator that yields the uppercase/lowercase mapping of this char as one or more chars.

Strong and expressive type system

Story time

https://dev.to/jagracey/hacking-github-s-auth-with-unicode-s-turkish-dotless-i-460n std docs note

> Note on locale

> In Turkish, the equivalent of ‘i’ in Latin has five forms instead of two:

> ‘Dotless’: I / ı, sometimes written ï
> ‘Dotted’: İ / i

> Note that the lowercase dotted ‘i’ is the same as the Latin.

> [...]

> The value of upper_i here relies on the language of the text:
> if we’re in en-US, it should be "I", but if we’re in tr_TR, it should be "İ".

> to_uppercase() does not take this into account, and so:

```
fn main() {
  let upper_i = 'i'.to_uppercase().to_string();
  assert_eq!(upper_i, "I");
}
```

Q: What is the difference between `String` and `Box<str>` ?

Q: What is the difference between `String` and `Box<str>` ?

A: One `usize` smaller, but no resizing

One potential use case: lots of immutable* text, but measure first before optimizing

See the `rust-analyzer` src

```
use std::mem;

fn main() {
  assert_eq!(16, mem::size_of::<Box<str>>());
  assert_eq!(24, mem::size_of::<String>());
  assert_eq!(8, mem::size_of::<usize>());
}
```

🟩 **Q: What is the difference between `Box<str>` and `Box<&str>` ?**

🟩 **A: `Box<str>` has ownership, meanwhile `Box<&str>` is just a (fancy) borrow that lives on the heap**

```rust
use std::mem;
fn main() {
    // Using Box<&str> to store a heap-allocated reference to a string slice
    let borrowed_string: &str = "Hello, borrowed string!";
    let boxed_borrowed_string: Box<&str> = Box::new(borrowed_string);

    // Using Box<str> to store an owned, heap-allocated string slice
    let owned_string: Box<str> = Box::from("Hello, owned string!");

    // Printing the sizes
    println!("Size of Box<&str>: {} bytes", mem::size_of_val(&boxed_borrowed_string));
    println!("Size of Box<str>: {} bytes", mem::size_of_val(&owned_string));
}
```

█ Q: What is the difference between `Box<str>` and `Box<&str>` ?

█ A: `Box<str>` has ownership, meanwhile `Box<&str>` is just a (fancy) borrow that lives on the heap

```rust
use std::mem;
fn main() {
    // Using Box<&str> to store a heap-allocated reference to a string slice
    let borrowed_string: &str = "Hello, borrowed string!";
    let boxed_borrowed_string: Box<&str> = Box::new(borrowed_string);

    // Using Box<str> to store an owned, heap-allocated string slice
    let owned_string: Box<str> = Box::from("Hello, owned string!");

    // Printing the sizes
    println!("Size of Box<&str>: {} bytes", mem::size_of_val(&boxed_borrowed_string));
    println!("Size of Box<str>: {} bytes", mem::size_of_val(&owned_string));
}
```

```
Size of Box<&str>: 8 bytes
Size of Box<str>: 16 bytes
```

FFI (Foreign Function Interface)

Caution, potential `unsafe` operations ahead.

Q: What is `unsafe`? Why is it `unsafe`?

A: See the `std` docs / Nomicon, but in short:

> No matter what, Safe Rust can’t cause Undefined Behavior. This is referred to as soundness: a well-typed program actually has the desired properties.

> The Nomicon has a more detailed explanation on the subject.

> To ensure soundness, Safe Rust is restricted enough that it can be automatically checked.
> Sometimes, however, it is necessary to write code that is correct for reasons which are too clever for the compiler to understand.
> In those cases, you need to use Unsafe Rust.

> Here are the abilities Unsafe Rust has in addition to Safe Rust:

- Dereference raw pointers
- Implement unsafe traits
- **Call unsafe functions**
- Mutate statics (including external ones)
- Access fields of unions

> However, this extra power comes with extra responsibilities: it is now up to you to ensure soundness.
> The unsafe keyword helps by clearly marking the pieces of code that need to worry about this.

FFI (Foreign Function Interface)

OsString

Q: Have I told you how fun `std` docs are?

> A type that can represent owned, mutable platform-native strings, but is cheaply inter-convertible with Rust strings.
> The need for this type arises from the fact that:
> On Unix systems, strings are often arbitrary sequences of non-zero bytes, in many cases interpreted as UTF-8.

> On Windows, strings are often arbitrary sequences of non-zero 16-bit values, interpreted as UTF-16 when it is valid to do so.

> In Rust, strings are always valid UTF-8, which may contain zeros.

> OsString and OsStr bridge this gap by simultaneously representing Rust and platform-native string values,
> and in particular allowing a Rust string to be converted into an “OS” string with no cost if possible.

> A consequence of this is that OsString instances are not NUL terminated; in order to pass to e.g., Unix system call, you should create a CStr. [...]

Q: Okay, but what are they good for?

A:

> OsString and OsStr are useful when you need to transfer strings to and from the operating system itself,
> or when capturing the output of external commands.

> Conversions between OsString, OsStr and Rust strings work similarly to those for CString and CStr.

OsString

Q: What does it mean in practice?

A: Behold

```rust
use std::ffi::OsString;
use std::fs::File;
use std::io::{self, Write};
use std::os::unix::ffi::OsStringExt;

fn main() -> io::Result<()> {
    // Create an OsString with invalid UTF-8
    // Source: https://www.cl.cam.ac.uk/~mgk25/ucs/examples/UTF-8-test.txt
    let invalid_utf8_osstring = OsString::from_vec(vec![0x80, 0x80, 0x80, 0xbf, 0xbf, 0xe0,
0x80]);

    // Write the OsString to a file
    let mut path = std::path::PathBuf::new();
    path.push("/home/orion/tmp/rust_talk/");
    path.push(&invalid_utf8_osstring);
    let mut file = File::create(&path)?;

    file.write_all(&invalid_utf8_osstring.into_vec())?;

    println!("File created with invalid UTF-8 content");
    println!("Don't forget to show it");

    Ok(())
}
```

`std::ffi::CStr`

```rust
use std::ffi::{CString, CStr};
use std::os::raw::c_char;


fn my_string_safe() -> String {
    // See the docs for safety
    let cstr = unsafe {
        // Convert the raw C-style string to a CStr
        CStr::from_ptr(my_string())
    };

    // Get copy-on-write Cow<'_, str>, then guarantee a freshly-owned String allocation
    String::from_utf8_lossy(cstr.to_bytes()).to_string()
}

fn main() {
    println!("my_string_safe: {:#?}", my_string_safe());
}

#[no_mangle]
pub extern "C" fn my_string() -> *const c_char {
    // Create a static C-style string
    let my_static_string = "Hello from the C world";

    let c_str = CString::new(my_static_string).expect("CString::new failed");

    // Return the raw pointer
    c_str.into_raw()
}
```

`std::ffi::CStr`

```rust
use std::ffi::{CString, CStr};
use std::os::raw::c_char;


fn my_string_safe() -> String {
    // See the docs for safety
    let cstr = unsafe {
        // Convert the raw C-style string to a CStr
        CStr::from_ptr(my_string())
    };

    // Get copy-on-write Cow<'_, str>, then guarantee a freshly-owned String allocation
    String::from_utf8_lossy(cstr.to_bytes()).to_string()
}

fn main() {
    println!("my_string_safe: {:#?}", my_string_safe());
}

#[no_mangle]
pub extern "C" fn my_string() -> *const c_char {
    // Create a static C-style string
    let my_static_string = "Hello from the C world";

    let c_str = CString::new(my_static_string).expect("CString::new failed");

    // Return the raw pointer
    c_str.into_raw()
}
```

```
my_string_safe: "Hello from the C world"
```

`std::ffi::CStr`

`pub unsafe fn from_ptr<'a>(ptr: *const i8) -> &'a CStr` says:

Safety

- The memory pointed to by `ptr` must contain a valid nul terminator at the end of the string.
- `ptr` must be valid for reads of bytes up to and including the null terminator. This means in particular:
  - The entire memory range of this `CStr` must be contained within a single allocated object!
  - `ptr` must be non-null even for a zero-length cstr.
- The memory referenced by the returned `CStr` must not be mutated for the duration of lifetime `'a`
- .
- The nul terminator must be within `isize::MAX` from `ptr``

`std::Cow` (⍰)

```
pub enum Cow<'a, B>where
    B: 'a + ToOwned + ?Sized,{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Q: What is a `Cow<>`?

A: A neat encapsulation for "give me owned type when needed"

What's `ToOwned`?

```
pub trait ToOwned {
    type Owned: Borrow<Self>;

    // Required method
    fn to_owned(&self) -> Self::Owned;

    // Provided method
    fn clone_into(&self, target: &mut Self::Owned) { ... }
}
```

▒▒▒▒ What's `Borrow` ?

> [..] These types provide access to the underlying data through references to the type of that data.
> They are said to be ‘borrowed as’ that type.
> For instance, a Box can be borrowed as T while a String can be borrowed as str.

```rust
pub trait Borrow<Borrowed>where
    Borrowed: ?Sized,{
    // Required method
    fn borrow(&self) -> &Borrowed;
}
```

▒▒▒▒ What's `?Sized` ?

```rust
pub trait Sized {}
```

> Types with a constant size known at compile time.

> All type parameters have an implicit bound of Sized.
> The special syntax ?Sized can be used to remove this bound if it’s not appropriate.

`std::Cow` (⍰)

```
pub enum Cow<'a, B>where
    B: 'a + ToOwned + ?Sized,{
    Borrowed(&'a B),
    Owned(<B as ToOwned>::Owned),
}
```

Q: What is a `Cow<>?`

A: A neat encapsulation for "Convert the `Borrowed<'a B>` to `Owned B` (-ish) when needed"

`Cow`

- **Q: Why is it really useful?**
- **A: When something might need modifications (see `into_owned()`)**

```rust
use std::borrow::Cow;
use std::time::{SystemTime, UNIX_EPOCH};

fn apply_fixup_on_demand(msg: &mut Cow<str>) {
    if !msg.contains("Timestamp:") {
        // If the log message is missing a timestamp, add a timestamp
        let timestamp = SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs();

        match msg {
            Cow::Borrowed(b) => {
                // If it's borrowed, convert to owned and modify
                *msg = Cow::Owned(format!("Timestamp: {} {}", timestamp, b));
            }
            Cow::Owned(m) => {
                // If it's already owned, modify in place
                m.insert_str(0, &format!("Timestamp: {} ", timestamp));
            }
        }
    }
}


fn main() {
    let log_message1: &str = "ERROR: Something went wrong";
    let mut log_message1 = Cow::Borrowed(log_message1);
    apply_fixup_on_demand(&mut log_message1);
    println!("Result 1: {:#?}", log_message1);

    let log_message2: String = "ERROR: Another issue".to_string();
    let mut log_message2 = Cow::Owned(log_message2);
    apply_fixup_on_demand(&mut log_message2);
    println!("Result 2: {:#?}", log_message2);

}
```

`Cow`

Q: Why is it really useful?

A: When something might need modifications (see `into_owned()`)

```rust
use std::borrow::Cow;
use std::time::{SystemTime, UNIX_EPOCH};

fn apply_fixup_on_demand(msg: &mut Cow<str>) {
    if !msg.contains("Timestamp:") {
        // If the log message is missing a timestamp, add a timestamp
        let timestamp = SystemTime::now().duration_since(UNIX_EPOCH).unwrap().as_secs();

        match msg {
            Cow::Borrowed(b) => {
                // If it's borrowed, convert to owned and modify
                *msg = Cow::Owned(format!("Timestamp: {} {}", timestamp, b));
            }
            Cow::Owned(m) => {
                // If it's already owned, modify in place
                m.insert_str(0, &format!("Timestamp: {} ", timestamp));
            }
        }
    }
}


fn main() {
    let log_message1: &str = "ERROR: Something went wrong";
    let mut log_message1 = Cow::Borrowed(log_message1);
    apply_fixup_on_demand(&mut log_message1);
    println!("Result 1: {:#?}", log_message1);

    let log_message2: String = "ERROR: Another issue".to_string();
    let mut log_message2 = Cow::Owned(log_message2);
    apply_fixup_on_demand(&mut log_message2);
    println!("Result 2: {:#?}", log_message2);

}
```

```
Result 1: "Timestamp: 1707043908 ERROR: Something went wrong"
Result 2: "Timestamp: 1707043908 ERROR: Another issue"
```

`smol_str` crate

From their README:

A SmolStr is a string type that has the following properties:

- [..]
- Strings are stack-allocated if they are:
  - Up to 23 bytes long
  - Longer than 23 bytes, but substrings of WS (see src/lib.rs). Such strings consist solely of consecutive newlines, followed by consecutive spaces
- If a string does not satisfy the aforementioned conditions, it is heap-allocated [..]

Fasterthanlime article: https://fasterthanli.me/articles/small-strings-in-rust#the-smol-str-crat

TL;DR:

> Our task for today is going to be: parsing a list of the 1000 largest US cities from a JSON file.

```rust
#[derive(Deserialize)]
struct Record {
    #[allow(unused)]
    city: String,
    #[allow(unused)]
    state: String,
}
```

> We can also see that between those peaks, memory usage increases steadily - each String stores its
> data on the heap,
> which explains the number of allocation events, 2017.

> Memory usage is lower than with String, and the number of allocations fell from 2017 to just 23!

> As we did before, we see peaks when the Vec resizes, but between them, everything is flat.
> It seems that 22 bytes is enough to store most of the names of the top 1000 US cities.

> [..] README says it primary use case is "good enough default storage for tokens of typical
> programming languages".

Recommended resources

- `std` docs
- https://fasterthanli.me/articles/working-with-strings-in-rust
- https://deterministic.space/secret-life-of-cows.html

Questions?