DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 32

1. STRINGS: `String` AND `&str`

2. LIFETIMES

1. STRINGS: `String` AND `&str`

2. LIFETIMES

RUST AND STRINGS

- We have avoided this topic so far
- It's complicated
- Unicode is complicated
- Advantages: internationalization and emojis out of the box

RUST AND STRINGS

- We have avoided this topic so far
- It's complicated
- Unicode is complicated
- Advantages: internationalization and emojis out of the box

- **Rust:** Unicode strings are a first–class citizen
- **Classical programming languages:**
  - ASCII strings are the default
  - Easier to manage
  - Additional libraries needed to deal with Unicode

Reminder: Single Characters (Unicode Scalar Values)

- Type: `char`
- Size: 4 bytes
- Note the single quotes!

REMINDER: SINGLE CHARACTERS (UNICODE SCALAR VALUES)

- Type: `char`
- Size: 4 bytes
- Note the single quotes!

In [2]:
```
let a : char = 'a';
let b = '🦕';
```

Dinosaurs:
🦕 (U+1F995)
🦖 (U+1F996)

REMINDER: SINGLE CHARACTERS (UNICODE SCALAR VALUES)

- Type: `char`
- Size: 4 bytes
- Note the single quotes!

In [2]:
```
let a : char = 'a';
let b = '🦕';
```

Dinosaurs:

🦕 (U+1F995)

🦖 (U+1F996)

In [3]:
```
// Mayan numeral (not all unicode characters are supported everywhere)
let c = '𝋡';
```

REMINDER: SINGLE CHARACTERS (UNICODE SCALAR VALUES)

- Type: `char`
- Size: 4 bytes
- Note the single quotes!

```
In [2]: let a : char = 'a';
        let b = '🦕';
```

Dinosaurs:
🦕 (U+1F995)
🦖 (U+1F996)

```
In [3]: // Mayan numeral (not all unicode characters are supported everywhere)
        let c = '𝋡';
```

```
In [4]: std::mem::size_of_val(&a)

Out[4]: 4
```

```
In [5]: std::mem::size_of_val(&b)

Out[5]: 4
```

STRING LITERALS

- String literal = when you create a string `"like this"`
- Note the double quotes
- What type are they?

STRING LITERALS

- String literal = when you create a string `"like this"`
- Note the double quotes
- What type are they?

In [6]:
```
let sample = "Hello, DS210!";
```

STRING LITERALS

- String literal = when you create a string `"like this"`
- Note the double quotes
- What type are they?

In [6]:
```
let sample = "Hello, DS210!";
```

In [7]:
```
let sample: String = "Hello, DS210!";
```

```
let sample: String = "Hello, DS210!";
                     ^^^^^^^^^^^^^^^^ expected struct `String`, found `&str`
let sample: String = "Hello, DS210!";
            ^^^^^^ expected due to this
mismatched types
help: try using a conversion method

.to_string()
```

STRING LITERALS

- String literal = when you create a string `"like this"`
- Note the double quotes
- What type are they?

In [6]:
```
let sample = "Hello, DS210!";
```

In [7]:
```
let sample: String = "Hello, DS210!";
```

```
let sample: String = "Hello, DS210!";
                     ^^^^^^^^^^^^^^^^ expected struct `String`, found `&str`
let sample: String = "Hello, DS210!";
            ^^^^^^ expected due to this
mismatched types
help: try using a conversion method

.to_string()
```

In [8]:
```
let sample: &str = "Hello, DS210!";
```

STRING LITERALS

- String literal = when you create a string `"like this"`
- Note the double quotes
- What type are they?

In [6]:
```
let sample = "Hello, DS210!";
```

In [7]:
```
let sample: String = "Hello, DS210!";
```

```
let sample: String = "Hello, DS210!";
                     ^^^^^^^^^^^^^^^^ expected struct `String`, found `&str`
let sample: String = "Hello, DS210!";
            ^^^^^^ expected due to this
mismatched types
help: try using a conversion method

.to_string()
```

In [8]:
```
let sample: &str = "Hello, DS210!";
```

`&str` is a **string slice**, internally behaves like `&[u8]`

ENCODING OF CHARACTERS

`a` and 🦕 were both 4 bytes

```
In [9]: std::mem::size_of_val("a")

Out[9]: 1
```

ENCODING OF CHARACTERS

`a` and `🦕` were both 4 bytes

```
In [9]: std::mem::size_of_val("a")

Out[9]: 1
```

```
In [10]: std::mem::size_of_val("🦕")

Out[10]: 4
```

ENCODING OF CHARACTERS

`a` and `🦕` were both 4 bytes

```
In [9]: std::mem::size_of_val("a")

Out[9]: 1
```

```
In [10]: std::mem::size_of_val("🦕")

Out[10]: 4
```

Characters need 1–4 bytes to be encoded.

ENCODING OF CHARACTERS

`a` and `🦕` were both 4 bytes

```
In [9]: std::mem::size_of_val("a")

Out[9]: 1
```

```
In [10]: std::mem::size_of_val("🦕")

Out[10]: 4
```

Characters need 1–4 bytes to be encoded.

```
In [11]: let dinos = "🦕🦖";
         std::mem::size_of_val(dinos)

Out[11]: 8
```

ENCODING OF CHARACTERS

`a` and `🦕` were both 4 bytes

```
In [9]: std::mem::size_of_val("a")
Out[9]: 1
```

```
In [10]: std::mem::size_of_val("🦕")
Out[10]: 4
```

Characters need 1–4 bytes to be encoded.

```
In [11]: let dinos = "🦕🦖";
         std::mem::size_of_val(dinos)
Out[11]: 8
```

Can select substrings, but they must be aligned with actual characters (or runtime error)

```
In [12]: dinos[0..1]
thread '<unnamed>' panicked at 'byte index 1 is not a c
har boundary; it is inside '🦕' (bytes 0..4) of `🦕🦖`',
src/lib.rs:130:40
stack backtrace:
   0: rust_begin_unwind
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
```

ENCODING OF CHARACTERS

`a` and `🦕` were both 4 bytes

```
In [9]: std::mem::size_of_val("a")
Out[9]: 1
```

```
In [10]: std::mem::size_of_val("🦕")
Out[10]: 4
```

Characters need 1–4 bytes to be encoded.

```
In [11]: let dinos = "🦕🦖";
         std::mem::size_of_val(dinos)
Out[11]: 8
```

Can select substrings, but they must be aligned with actual characters (or runtime error)

```
In [12]: dinos[0..1]
thread '<unnamed>' panicked at 'byte index 1 is not a c
har boundary; it is inside '🦕' (bytes 0..4) of `🦕🦖`',
src/lib.rs:130:40
stack backtrace:
   0: rust_begin_unwind
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
```

```
In [13]: let dinos = "🦕🦖";
         dinos[0..4]
Out[13]: "🦕"
```

ENCODING OF CHARACTERS

`a` and `🦕` were both 4 bytes

```
In [9]: std::mem::size_of_val("a")
Out[9]: 1
```

```
In [10]: std::mem::size_of_val("🦕")
Out[10]: 4
```

Characters need 1–4 bytes to be encoded.

```
In [11]: let dinos = "🦕🦖";
         std::mem::size_of_val(dinos)
Out[11]: 8
```

Can select substrings, but they must be aligned with actual characters (or runtime error)

```
In [12]: dinos[0..1]
```

```
thread '<unnamed>' panicked at 'byte index 1 is not a c
har boundary; it is inside '🦕' (bytes 0..4) of `🦕🦖`',
src/lib.rs:130:40
stack backtrace:
   0: rust_begin_unwind
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
5102f57a/library/std/src/panicking.rs:498:5
   1: core::panicking::panic_fmt
             at /rustc/9d1b2106e23b1abd32fce1f17267604a
```

```
In [13]: let dinos = "🦕🦖";
         dinos[0..4]
Out[13]: "🦕"
```

```
In [14]: let sample = "Hello, world!";
         sample[7..]
Out[14]: "world!"
```

STRINGS

- String type is dynamic: `Vec<u8>` internally
- Can add characters and strings to the end

STRINGS

- String type is dynamic: `Vec<u8>` internally
- Can add characters and strings to the end

In [15]:
```
let mut sample = String::new();

//append string
sample.push_str("abc");
sample
```

Out[15]: "abc"

STRINGS

- String type is dynamic: `Vec<u8>` internally
- Can add characters and strings to the end

In [15]:
```
let mut sample = String::new();

//append string
sample.push_str("abc");
sample
```

Out[15]: "abc"

In [16]:
```
// append character
sample.push('d');
sample
```

Out[16]: "abcd"

CONVERTING LITERALS TO TYPE `String`

Use `.to_string()` or `String::from(...)`

```
In [17]: let string_1 = "This is a test".to_string();
         let string_2 = String::from("This is a test");
         string_1 == string_2

Out[17]: true
```

CONVERTING LITERALS TO TYPE `String`

Use `.to_string()` or `String::from(...)`

```
In [17]: let string_1 = "This is a test".to_string();
         let string_2 = String::from("This is a test");
         string_1 == string_2

Out[17]: true
```

Can also use macro `format!(...)`:

- same syntax as `println!(...)`
- produces an object of type `String`

```
In [18]: let sample: String = format!("{} == {}",string_1,string_2);
         sample

Out[18]: "This is a test == This is a test"
```

STRING CONCATENATION VIA `+`

- Takes ownership of the first parameter
- Second parameter: `&str`

```
In [19]: let string_1 = "abc".to_string();
         let string_2 = "def".to_string();
```

```
In [20]: string_1 + &string_2

Out[20]: "abcdef"
```

STRING CONCATENATION VIA `+`

- Takes ownership of the first parameter
- Second parameter: `&str`

```
In [19]: let string_1 = "abc".to_string();
         let string_2 = "def".to_string();
```

```
In [20]: string_1 + &string_2

Out[20]: "abcdef"
```

Why `+` takes ownership of `string_1`:

- reason: efficiency
- no need to copy the content of the first string (unless the container size has to be increased)

WRITING GENERIC CODE

- Use string slices &str if possible
- This will work with `String` and `&str`

In [21]:
```
fn show(message: &str) {
    println!("{}",message);
}
```

WRITING GENERIC CODE

- Use string slices &str if possible
- This will work with `String` and `&str`

In [21]:
```
fn show(message: &str) {
    println!("{}",message);
}
```

In [22]:
```
// automatic conversion to &str from &String
let mut my_string = String::from("ds210");
show(&my_string);
show("ds210");
```

```
ds210
ds210
```

1. STRINGS: `String` AND `&str`

2. LIFETIMES

LIFETIMES

- How long your reference is valid
- Important when sharing references
    - Example: via function output

LIFETIMES

- How long your reference is valid
- Important when sharing references
    - Example: via function output

Challenge: return the reference to the greater of two integers

LIFETIMES

- How long your reference is valid
- Important when sharing references
    - Example: via function output

Challenge: return the reference to the greater of two integers

In [23]:
```
fn ref_to_max(x:&mut i32, y:&mut i32) -> &mut i32 {
    if *x >= *y {
        x
    } else {
        y
    }
}
```

```
fn ref_to_max(x:&mut i32, y:&mut i32) -> &mut i32 {
              ^^^^^^^^

fn ref_to_max(x:&mut i32, y:&mut i32) -> &mut i32 {
                          ^^^^^^^^

fn ref_to_max(x:&mut i32, y:&mut i32) -> &mut i32 {
                                        ^ expected named lifetime parameter

missing lifetime specifier
help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `x` or
`y`
help: consider introducing a named lifetime parameter

<'a>
```

SPECIFYING LIFETIMES

`'t` specifies how long a reference lives (`t` is some string)

- immutable example: `&'t i32`
- mutable example: `&'t mut i32`

SPECIFYING LIFETIMES

`'t` specifies how long a reference lives (`t` is some string)

- immutable example: `&'t i32`
- mutable example: `&'t mut i32`

In [24]:
```
fn ref_to_max<'a>(x:&'a mut i32, y:&'a mut i32) -> &'a mut i32 {
    if *x >= *y {
        x
    } else {
        y
    }
}
```

SPECIFYING LIFETIMES

`'t` specifies how long a reference lives (`t` is some string)

- immutable example: `&'t i32`
- mutable example: `&'t mut i32`

In [24]:
```
fn ref_to_max<'a>(x:&'a mut i32, y:&'a mut i32) -> &'a mut i32 {
    if *x >= *y {
        x
    } else {
        y
    }
}
```

In [25]:
```
let mut x = 13;
let mut y = 3;
{
    println!("{} {}",x,y);
    *ref_to_max(&mut x, & mut y) = 5;
    println!("{} {}",x,y);
    *ref_to_max(&mut x, & mut y) = 1;
    println!("{} {}",x,y);
    *ref_to_max(&mut x, & mut y) = 0;
    println!("{} {}",x,y);
};
```

```
13 3
5 3
1 3
1 0
```

APPLYING THIS FUNCTION

- Different references may have different lifetimes
- Rust will automatically select the shortest

APPLYING THIS FUNCTION

- Different references may have different lifetimes
- Rust will automatically select the shortest

In [26]:
```
let mut x = 1;
let mut y = 10;
{
    let ref1 = &mut y;
    {
        let ref2 = &mut x;
        *ref_to_max(ref1, ref2) = 3;
    }
    *ref1 *= -1;
};
(x,y)
```

Out[26]: (1, -3)

APPLYING THIS FUNCTION

- Different references may have different lifetimes
- Rust will automatically select the shortest

In [26]:
```
let mut x = 1;
let mut y = 10;
{
    let ref1 = &mut y;
    {
        let ref2 = &mut x;
        *ref_to_max(ref1, ref2) = 3;
    }
    *ref1 *= -1;
};
(x,y)
```

Out[26]: (1, -3)

MULTIPLE LIFETIMES POSSIBLE

In [27]:
```
fn multiple<'a, 'b>(x:&'a str, y:&'b str) -> (&'a str,&'b str) {
    (x,y)
}
multiple("abc","def")
```

Out[27]: ("abc", "def")

STRING LITERALS ARE FOREVER

- Memory for them assigned in the code
- Their references do not expire
- Can be specified by `'static`

```
In [28]: let example: &'static str = "abc";
```

IN SOME CASES RUST AUTOMATICALLY GUESSES REQUIRED LIFETIMES

Example 1: exactly one input lifetime parameter => used as the lifetime of output

All functions `get_shorter` below equivalent

In [29]:
```
struct TwoStrings{
    a: String,
    b: String,
}
```

In [30]:
```
fn get_shorter_1(ts:&TwoStrings) -> &str {
    if ts.a.len() < ts.b.len() {
        &ts.a
    } else {
        &ts.b
    }
}
```

In [31]:
```
fn get_shorter_2<'a>(ts:&'a TwoStrings) -> &'a str {
    if ts.a.len() < ts.b.len() {
        &ts.a
    } else {
        &ts.b
    }
}
```

In [32]:
```
fn get_shorter_3<'a>(ts:&'a TwoStrings) -> &str {
    if ts.a.len() < ts.b.len() {
        &ts.a
    } else {
        &ts.b
    }
}
```

IN SOME CASES RUST AUTOMATICALLY GUESSES REQUIRED LIFETIMES

Example 2: one of the lifetime parameters is `&self` or `&mut self` => its lifetime used as the lifetime of output

Methods `get_longer` below equivalent

In [33]:
```
impl TwoStrings {
    fn get_longer_1(&self, unused:&TwoStrings) -> &str {
        if self.a.len() < self.b.len() {
            &self.a
        } else {
            &self.b
        }
    }

    fn get_longer_2<'a, 'b>(&'a self, unused:&'b TwoStrings) -> &'a str {
        if self.a.len() < self.b.len() {
            &self.a
        } else {
            &self.b
        }
    }
}
```