C/CPS 506

Compara **Languages**

Prof. Ale

Topic 10: Ownership and lifetime in Rust

Ryerson University

Notice!

Obligatory copyright notice in the age of digital delivery and online classrooms:

The copyright to this ex Ufkes. Students registered in course CCPS 506 can us al for the purposes of this course but no other use is permitted, and there can be no sale or transfer or use of the work for any other purpose without explicit permission of Alex Ufkes.

Course Administration

- Getting closer! Two res.
- Don't forget about the assignments!

Control Flow

`if / else`

- As with other imperative languages, the else is optional.
- Recall that this is not the case with Haskell!
- a complete if-then-else

Boolean Conditions?

Mandatory.

<u>In C/C++ (and Elixir, with caveats):</u>

- Non-zero values are “
- Only 0/nil considered

```
if (3.141592)
  cout << “Valid!” << endl;
```

<u>In Java (and Haskell, Rust):</u>

tions must be Boolean

```
1592)
.out.println(
“Compile Error”);
```

Converting non-Boolean to Boolean requires implicit conversion, which, as we’ve seen, Rust does not do.

Boolean Conditions?

Mandatory.

Ah! But can we cast?

Nope.

if / else if / else

- As we'd expect.

} even though there's only

ment per branch

- red.

- reats these as blocks whose last line can be an expression.

`if / else if / else`

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

`if` / `else if` / `else`

- `let state = {…};` is a statement
- `{…}` is an expression that will evaluate ring. `if` == expression!
  n", "Liquid", or "Boiling"
  ion is in a scope block { }
  e of a scope block is the last expression
- Leaving the `;` off makes these strings expressions.

if / else if / else

Problem?

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

Might return float, might return int

Remember: Strong, static typing. No implicit conversion!

Looping

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

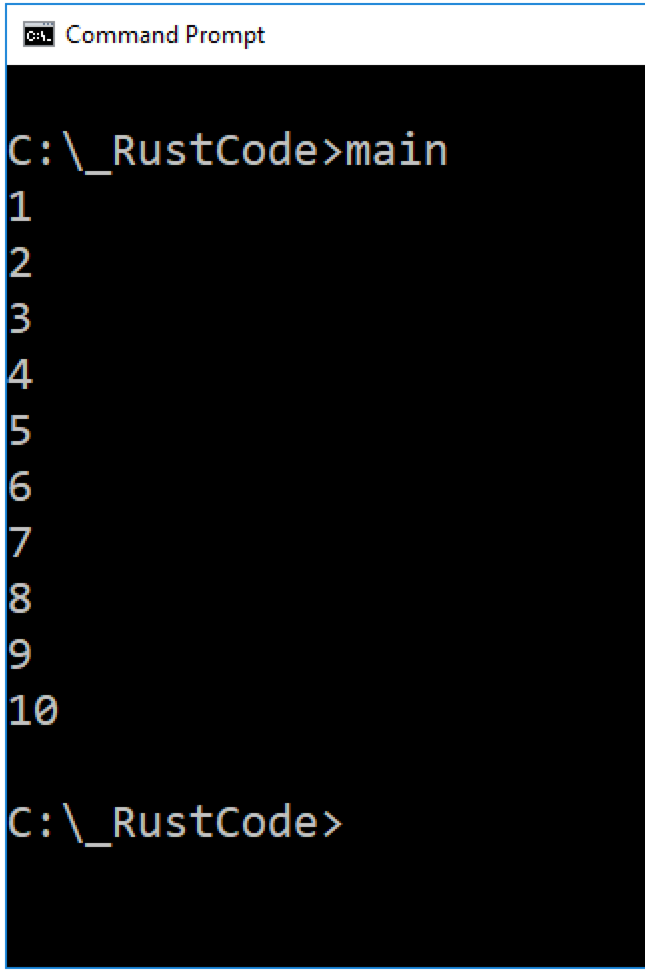
Just like `while(true){}` in Java

Conditional Looping: `while`

other

ages.

- Rust u =



Conditional Looping: `for`

Similar to an enhanced for loop in Java:

- Invoke iter() method of array nums
- **`elem`** takes the value of each ement in the array.

  ! Never go out of bounds.

Conditional Looping: for

Use .. to create a range

- Create a *Range* containing 0 to ***9***
- Top of range not included!
- Just like range() in Python

Conditional Looping: for

Not as safe!

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

- Here we must be careful
- Higher chance of accidentally overrunning array bounds

A loop is a loop is a loop

Wait, *what*?

Wait, what?

- We didn't specify the type of **`i`**, but shouldn't it default to `i32`?
  pe, `i32` should be default.

- Rus ow signed integers to be indexes!
- It inferred the type as unsigned! Thus checking less than zero is pointless.

```
C:\_RustCode>rustc main.rs
warning: comparison is useless due to type limits
 --> main.rs:8:12
  |
8 |         if i < 0 { break; }
  |            ^^^^^
  |
```

Rust doesn't allow signed integers to be used as array indexes!

Need to adjust our logic a bit…

M n....

Ownership

Ownership

Arguably Rust's most unique feature:

- In C, the programmer is responsible for allocating and freein leaks common!

- In Java, ga y looks for unused memory and frees

- Rust takes a third approach: A system of ownership with rules checked at compile time.
  - Thus, the program is not slowed at run-time

Reminder: Stack VS Heap

Stack:
- Last in, first out
- Push/pop stack frames is fast
- Data has known, f

Heap:
- Less organized
- Slower access, follow pointers
- ze can be unknown

- If we dynamically allocate memor pointer goes on the stack, the memory itself is in the heap.
- Heap memory is allocated by the OS at the request of the program.
- Stack memory (some fixed amount) belongs to the program, no need to invoke the OS.

Ownership

Three rules:

1. Each value in Rust has a variable that's called its *owner*.
2. There can only

3. When the owner goes out of value is dropped.

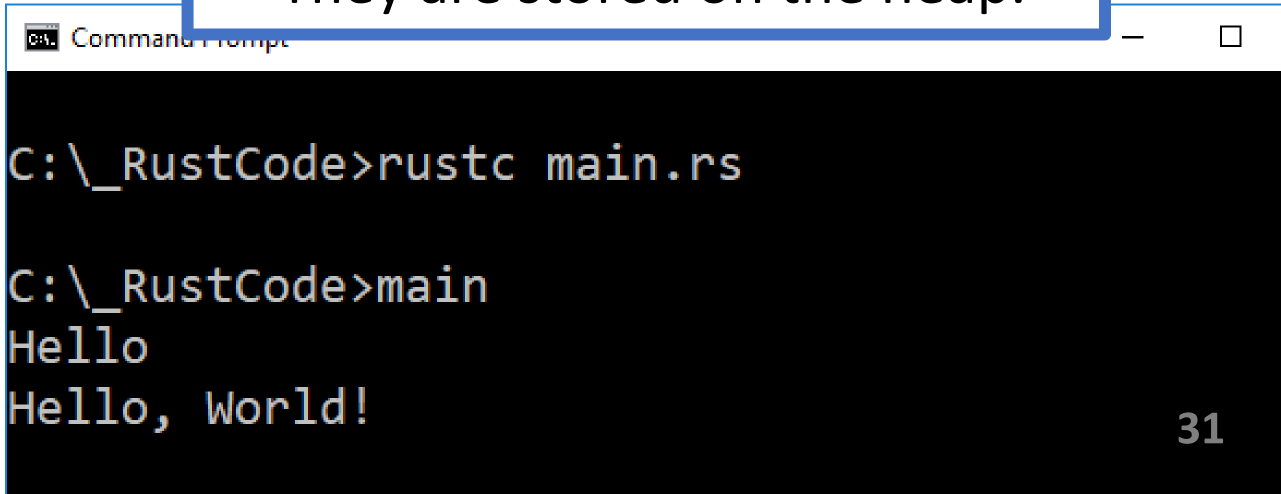
Scope in Rust

This is normal, nothing new.

- Primitives stored on the stack behave as per usual.
- How does Rust clean up data stored on the heap?
- Consider Strings – A complex type stored on the heap.

Strings

- String literals are different from regular strings.
  - heir size is fixed, encoded irectly into the executable.
  - ngs not defined as a literal ht have unknown size
- They are stored on the heap.



Heap Strings

- Memory for string requested at run time.
- Memory must be returned to the OS when we're done with the string.

- Calli a memory request.
- Onc avior. In Java we would say: String s = new String accomplish the same.

What happens when we no longer need that string?

What happens when we no longer need that string?

- Without garbage collection, we must identify when memory is no longer being used and free it explicitly.
- This has historically been a difficult programming problem.
- Too early, vari late, waste memory. Do it problem.
- We need to pair one `allocat` `free()`.

In Rust, memory is automatically returned when the variable that ***owns*** it leaves scope.

In Rust, memory is automatically returned when the variable that owns it leaves scope.

What about having multiple references to a single object?
Freeing after one leaves scope invalidates the others. In Java:

Three references, one object!

But Remember!

Ownership - Three Rules:

1. Each value in Rust has a variable that's called its *owner*.
2. There can only e.
3. When the owner goes out of e value is dropped.

There can only be one!

In Rust, memory is automatically returned when the variable that owns it leaves scope.

- When a variable goes out of scope, Rust calls a special function automatically called **`drop()`**
- This function is called at the closing **}**
- What ha variables interacti


```rust
fn main()
{
    let x = 5;
    let y = x;
}
```

- With primitives, we get two separate variables stored in memory (stack)
- **x** and **y** are separate – changing one does not affect the other
- This is typical, and efficient

```
fn main()
{
    let s1 = String::from("Hello");
    let s2 = s1;
}
```


On the stack

On the heap

```rust
fn main()
{
    let s1 = String::from("Hello");
    let s2 = s1;
}
```


opied; heap data is not.

- C data is more expensive.
- T in most imperative languages.
- We can still potentially free data twice
- We can still potentially invalidate other references

1. Each value in Rust has a variable that’s called its *owner*.
2. **There can only be one owner at a time.**
3. When the owner goes out of scope, the value is dropped.

1. Each value in Rust has a variable that's called its *owner*.
2. **There can only be one owner at a time.**
3. When the owner goes out of scope, the value is dropped.

When we say `let s2=s1`,
1 becomes invalid.
hus, when it leaves scope,
memory is not freed.
- We can no longer use s1!

```rust
fn main()
{
    let s1 = String::from("Hello");
    let s2 = s1;
}
```


We say s1 gets *moved* to s2

In Rust, we say s1 gets ***moved*** to s2

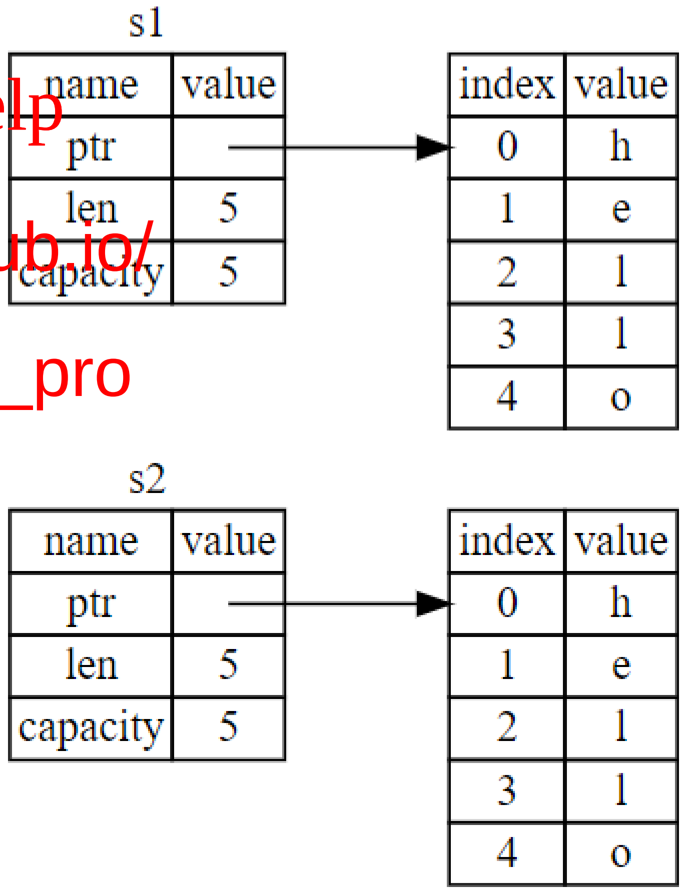
Different fr py, since the

old re idated.

Only one reference can free the heap memory.

`clone()`

Like most languages, Rust can clone:



`clone()`

Like most languages, Rust can clone:

Ownership and Functions

Passing an argument moves or copies, just like assignment:

Ownership and Functions

Passing an argument moves or copies, just like assignment:

- Ownership moved from **s** to **word**!
  nvalid!
  y different from any other
  e're used to.
- happen with primitives because they will simply be copied.
- We get a hint:

```
= note: move occurs because `s` has type `std::string::String`, which does not implement the `Copy` trait
```

Returning Ownership

Returning Ownership

- Ownership moved from **s** to **word** and back to **s**

  alid when we move to **word**

  invalid when moved to **s**

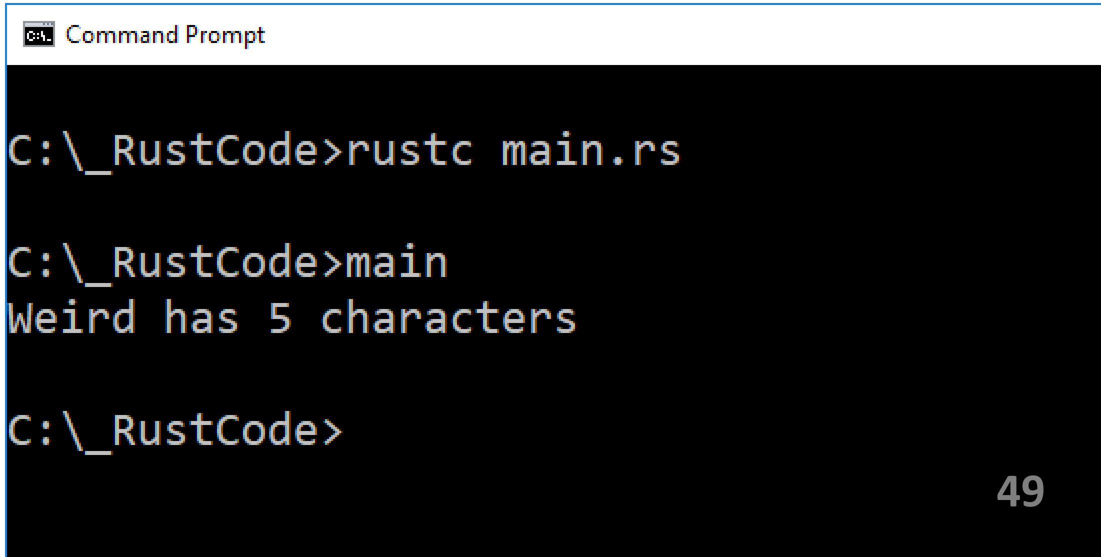
  because **s** is mutable.

  ing_pass reaches }, **word** has already been moved to **s**
- Thus **word** is invalid and the string on the heap isn't freed.

Returning Ownership

Limiting. Forced to use return value for ownership.

- **s1** moves to **word**, **word** moves to **s2**
  tuple consisting of the
  f word, and word itself.
- ion returns length of array.



```
Command Prompt

C:\_RustCode>rustc main.rs

C:\_RustCode>main
Weird has 5 characters

C:\_RustCode>
```

Ownership: Moving VS *Borrowing*

Instead of returning a tuple, pass a reference:

- This looks like C++
- **`word`** is now a *reference* to **`s1`**
- What about ownership?
- What's happening in memory?

Ownership: Moving VS *Borrowing*

word s1

- word is a reference to s1, it does NOT point to the string in the heap.
- word has no ownership over s1.
- We call this ***borrowing***.

Ownership: Moving VS *Borrowing*

Unlike C++, we can't modify something we're borrowing:

?

```
&String) -> usize
-------- use `&mut String` here to make mutable
```


`word` is a mutable reference, borrowed from `s1`

Borrowing Rules

Can only have <u>one</u> mutable borrow at a time:

When the first mutable borrow goes out of scope, we can borrow again

Borrowing Rules

Can only have one mutable borrow at a time:

- `push_str` must make utable borrow of s1
  ot allowed!

When the first mutable borrow goes out of scope, we can borrow again

pe of r1

Scope of r2

When the first mutable borrow goes out of scope, we can borrow again

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

Here, **r3** is already a reference.
We're not borrowing again.

Borrowing Rules

Using an immutably borrowed value prevents mutable borrow:

```
fn main()
{
    let mut word = String::from("
    let r1 = &word;
    word.push_str(", or what?");
    println!("{}", r1);
}
```

Borrowing Rules: In Short

In any given scope, only ONE of the following can be true:

1. We can have a single mutable borrow
2. We can have able borrows

These restrictions keep mutation under control

Slices

Slices

Reference to a subset of an array

- We've seen this notation before!
- Remember that the second index is *not* included

Slices, Arguments, Functions

- **Reminder:** indexes must be **usize**
- Pass in reference to array

slice (reference to subarray)

nly e memory

nd **nums** point to different parts of the same memory.

String Slices

… are a little bit different.

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
Hello
World!

C:\_RustCode>
```

Normal so far

String Slice Type

- &str is a reference to a string slice
- &String is a reference to a String
- String VS string slice: different types

han that, the function works

e as with numeric arrays.

lice is effectively a ***read-only*** view of a String.

String Slice Type

Better to d

```
fn get_slice (w: &str, s: usize, e: usize) -> &str
{
    &w[s..e]
}
```

Works for both Strings and string slices

String Literals

Recall:

- String literals are different from regular strings.
- Their size is fixed, ***encoded directly into the executable***.
- They are immutable.

In es:

type of **`msg`** is **`&str`**

- It's a slice pointing to a specific point of the binary file.
- This is why string literals are immutable!

Lifetime

Rust Features

Memory Safety:

o be memory safe

ointers are not

permitte

Dangling References

Rust prevents them:

e()

n a reference to it

- s goes out of scope when dangle function ends.
- What happens to the reference that was returned?

Dangling References

Rust prevents them:

Lifetime?

Lifetime is a very distinct feature of Rust:

Every reference in Rust has *lifetime*

The lifetime of a reference is the scope for whic[illegible] valid.

Lifetimes are typically i[illegible]d inferred, but can be defined explicitly

Just like variable types!

Example

- **r** is a reference to **x**
- **x** goes out of scope while **r** is still referring to it!

The Borrow Checker

- The Rust compiler has a “Borrow Checker” that compares scope to determine if borrows are valid

- If one variable bo le being borrowed must have a lifetime at

le doing the borrowing.

What happens if the borrow checker gets confused?

Generic Lifetimes

Consider:

gram:

n accepts two string slices,

e slice that is longer.

- t slices are just references
- There's no ownership changing here
- No moves

Generic Lifetimes

Consider:

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

Generic Lifetimes

The Borrow Checker can't determine lifeti turn value, because it's not clear which input argument the return value will borrow from.

More generally: The borrow checker follows certain patterns when determining lifetime. If none of its patterns apply, we get a lifetime error.

Generic Lifetimes

- We as programmers know that this function is perfectly safe.
- er to string literals which live ire d e program.

  R
- What's obvious to us is not necessarily obvious to the compiler.
- Thus, we get compile errors.

Generic Lifetimes

It even happens when the return reference is fixed:

Lifetime Annotation Syntax

When the borrow checker is confused (for whatever reason), we must be specific:

ric lifetime

generic type: <T>

- es a generic lifetime, **a**
- is reference has lifetime **a**

Command Prompt

```
C:\_RustCode>main
abcde

C:\_RustCode>
```

Generic Lifetimes

<u>What does mean precisely?</u>

- The function accepts two arguments
- Both live at least as long as lifetime **a**
  tring slice returned will live
  long as lifetime **a**
- ow what **a** is, just that both
  nd return value have the
  same lifetime.

Generic Lifetimes

<u>However!</u>

- We're ***NOT*** actually changing any lifetimes!
- We're just explicitly indicating them to help orrow Checker.
- ecker will reject any values that to these constraints.

So how can we break this?

Consider

- Lifetime of **s1** is different from **s2** and **s3**.
- Lifetime **'a** is the scope in which **x** and **y** are when **s1** and **s2** are valid.
  se **s3**, **s1** and **s2** are valid. Thus, checker accepts this code.
- **s3** re mething that is valid until after the last time **s3** is used.

Now This:

- Here, lifetime **a** excludes a reference made by **s3**
- **s3** references something that *might* be out of scope (**s2** will be, **s1** won't be)
- When we last use **s3**, **s2** is no longer valid.

e it doesn't matter, because
h s1 and s2 as string slices.

- Slices are ap, and thus references to them will always be valid.

Command Prompt

```
C:\_RustCode>rustc main.rs

C:\_RustCode>main
abcde
```

Oops. Let's try again with Strings instead…

Assignment Project Exam Help

https://eduassistpro.github.io/

Add WeChat edu_assist_pro

Lifetime Considerations

In general, we need some sort of lifetime indication any time we're passing in more than one reference and returning a reference.

eit pointless


```
fn sum_len (x: &str, y: &str) -> usize
{
    x.len() + y.len()
}
```

As is this

Lifetime Considerations

Originally, every reference required a lifetime specifier.

The Rust developers noticed some cases of reference passing were always the same, and thus ad [illegible] he compiler to recognize without r [illegible] nnotations.


```
fn sum_len (x: &str, y: &str) -> usize
{
    x.len() + y.len()
}
```

```
fn first (x: &str) -> &str
{
    x
}
```

Lifetime Considerations

The compiler first checks its list of known patterns

If none are found, we ch as we've been seeing

What are thes ?

Lifetime Inference Rules

1. The compiler first assigns a *different* lifetime to each reference input parameter.

Is

```
fn sum_len<'a,'b>(x: &'a str, y: &'b str) -> usize
{
    x.len() + y.len()
}
```

Lifetime Inference Rules

1. The compiler first assigns a *different* lifetime to each reference input parameter.
2. If there is ***one*** input reference parameter, it is assigned the same lifetime as any


```
fn first (x: &str) -> &str
{
    x
}
```

Is seen as:

Lifetime Inference Rules

1. The compiler first assigns a *different* lifetime to each reference input parameter.
2. If there is ***one*** input reference parameter, it is assigned the same lifetime as any
3. If there are mul t one of them is **`&self`**, then the output references hav lifetime as **`&self`**.

If, after applying these rules, there are still references *without* a lifetime specifier, we get a compile error.

If, after applying these rules, there are still references without a lifetime specifier, we get a compile error.

We don’t get errors here, because applying rules 1 and 2 results in all references having annotated lifetimes

We get an error here, because even after applying all three rules, we still don't have a lifetime annotation for the output:

```
fn first<'a,'b> (x: &'a str, y: &'b str) -> &str
{
    x
}
```

1. The compiler first assigns a *different* lifetime to each reference input parameter.

ere is ***one*** input reference
meter, it is assigned the same
e as any output references.
are multiple input references,
but one of them is **`&self`**, then the output references have the same lifetime as **`&self`**.

Rule 1 applies, Rules 2 and 3 do not

We get an error here, because even after applying all three rules, we still don't have a lifetime annotation for the output:

- No lifetime annotation after applying rules.
- mpile error.


```
fn first<'a,'b> (x: &'a str, y: &'b str) -> &str
{
    x
}
```

```
Command Prompt
C:\_RustCode>rustc main.rs
error[E0106]: missing lifetime specifier
  --> main.rs:17:45
   |
17 |  fn first<'a,'b> (x: &'a str, y: &'b str) -> &str
   |                                              ^ expec
ted lifetime parameter
   |
```

Static Lifetime

- A special lifetime that is simply the duration of the program.
- String literals have a static lifetime.
- Makes sense, they're not on the heap but embedded in the executable

Static Lifetime

- You might get error messages suggesting you use static lifetime.
- Be careful doing so. Does your reference really need to live for the duration of t
- It's a lazy solutio s of global variables to avoid using pointers or refere

Fantastic Rust Reference:

https://doc.rust/second-edition/