C/CPS 506

Comparative Programming Languages

Prof. Alex Ufkes

Topic 10: Ownership and lifetime in Rust

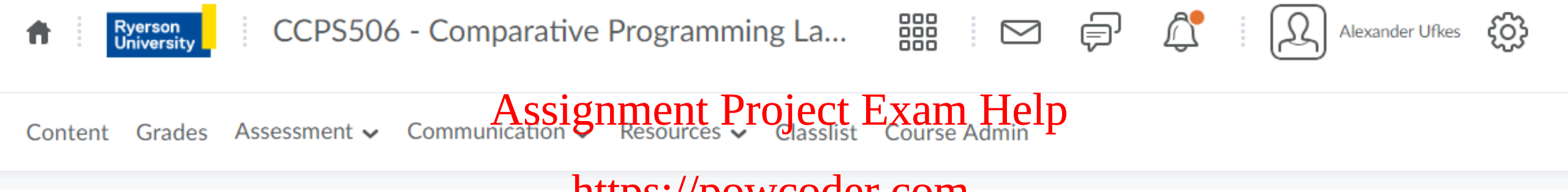
Ryerson University

Notice!

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

The copyright to this original work is held by Alex Ufkes. Students registered in course CPS 506 can use this material 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 more lectures.
- Don’t forget about the assignments!

Control Flow

if / else

main.rs

```rust
fn main()
{
    let num = 3;

    if num > 5 {
        println!("Greater than 5");
    }
    else {
        println!("Not that thing I just said");
    }

}
```

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

Command Prompt

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

C:\_RustCode>main
Not that thing I just said

C:\_RustCode>
```

Boolean Conditions?

Mandatory.

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

- Non-zero values are "truthy".
- Only 0/nil considered false.


```
if (3.141592)
  cout << "Valid!" << endl;
```

In Java (and Haskell, Rust):

- Conditions must be Boolean

```
if (3.141592)
  System.out.println(
    "Compile Error");
```

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

Boolean Conditions?

Mandatory.

main.rs

```rust
fn main()
{
    let num = 3;

    if 1 == true {
        println!("Greater than
    }
    else {
        println!("Not that thin
    }

}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0308]: mismatched types
 --> main.rs:5:13
  |
5 |     if 1 == true {
  |             ^^^^ expected integral variable,
 found bool
  |
  = note: expected type `{integer}`
             found type `bool`

error[E0277]: the trait bound `{integer}: std::c
```

Ah! But can we cast?

Nope.

main.rs

```rust
fn main()
{
    let num = 3;

    if 1 as bool {
        println!("Greater than
    }
    else {
        println!("Not that thi
    }

}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0054]: cannot cast as `bool`
 --> main.rs:5:8
  |
5 |     if 1 as bool {
  |        ^^^^^^^^^ unsupported cast
  |
  = help: compare with zero instead

error: aborting due to previous error

For more information about this error, try `rust
```

if / else if / else

main.rs

```rust
fn main()
{
    let temp = 33;

    if temp < 0 {
        println!("Frozen");
    }
    else if temp < 100 {
        println!("Liquid");
    }
    else {
        println!("Boiling");
    }
}
```

- As we'd expect.
- We use { } even though there's only one statement per branch
- This is required.
- Why? Rust treats these as blocks whose last line can be an expression.

if / else if / else

main.rs

```rust
fn main()
{
    let temp = 33;

    let state = if temp < 0 { "Frozen" }
                else if temp < 100 { "Liquid" }
                else { "Boiling" };

    println!("Water is {}!", state);
}
```

if / else if / else

```rust
fn main()
{
    let temp = 33;

    let state = if temp < 0 { "Frozen" }
                else if temp < 100 { "Liquid" }
                else { "Boiling" };

    println!("Water is {}!", state);
}
```

- `let state = {…};` is a statement
- `{…}` is an expression that will evaluate to a string. `if` == expression!
- "Frozen", "Liquid", or "Boiling"
- Each option is in a scope block { }
- The value of a scope block is the last expression
- Leaving the `;` off makes these strings expressions.

if / else if / else

```rust
fn main()
{
    let temp = 33;

    let state = if temp < 0 { "Frozen" }
                else if temp < 100 { "Liquid" }
                else { "Boiling" };

    println!("Water is {}!", state);
}
```

Command Prompt

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

C:\_RustCode>main
Water is Liquid!

C:\_RustCode>
```

Problem?

```rust
fn main()
{
    let temp = 33;

    let num = if temp > 50 { 33.33 }
              else { 99 };

    println!("num: {}", num);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0308]: if and else have incompatible types
 --> main.rs:5:15
  |
5 |         let num = if temp > 50 { 33.33 }
  |                   ^^^^^^^^^^^^^^^^^^^^^^
6 |                   else { 99 };
  |                   ^^^^^^^^^^^^ expected floating-point variable, found integral variable
  |
  = note: expected type `{float}`
                found type `{integer}`
```


Might return float, might return int

Remember: Strong, static typing. No implicit conversion!

Looping

Looping

main.rs

```rust
fn main()
{
    loop {
        println!("Again!");
    }
}
```

Command Prompt

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

C:\_RustCode>main
```

Command Prompt

```
Again!
Again!
Again!
Again!
Again!
Again!
Again!
Again!
Again!
Again!
^C
C:\_RustCode>
```


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

Conditional Looping: `while`

```rust
fn main()
{
    let mut n = 1;

    while n <= 10
    {
        println!("{}", n);
        n += 1;
    }
}
```


- Similar in form to other imperative languages.
- Rust understands +=

Command Prompt

```
C:\_RustCode>main
1
2
3
4
5
6
7
8
9
10

C:\_RustCode>
```

Conditional Looping: for

Similar to an enhanced for loop in Java:

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8];

    for elem in nums.iter()
    {
        print!("{} ", elem);
    }
    println!();
}
```

- Invoke iter() method of array nums
- **elem** takes the value of each element in the array.
- Safe! Never go out of bounds.

Command Prompt

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

C:\_RustCode>main
1 2 3 4 5 6 7 8

C:\_RustCode>
```

Conditional Looping: for

Use .. to create a range

```
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

    for i in (0..10).rev()
    {
        print!("{} ", nums[i]);
    }
    print!("\nLIFTOFF!\n");
}
```


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

Conditional Looping: for

Not as safe!

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 1

    for i in (0..10).rev()
    {
        print!("{} ", nums[i]);
    }
    print!("\nLIFTOFF!\n");
}
```

Command Prompt

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

C:\ RustCode>main
10 9 8 7 6 5 4 3 2 1
LIFTOFF!
```

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

A loop is a loop is a loop

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut i = 9;

    loop
    {
        if i < 0 { break; }
        print!("{} ", nums[i]);
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs
warning: comparison is useless due to type limits
 --> main.rs:8:12
  |
8 |         if i < 0 { break; }
  |            ^^^^^
  |
  = note: #[warn(unused_comparisons)] on by defaul
t

C:\_RustCode>
```

Wait, *what*?

Wait, what?

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7];
    let mut i = 9;

    loop
    {
        if i < 0 { break; }
        print!("{} ", nums[i]);
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

- We didn't specify the type of **`i`**, but shouldn't it default to **`i32`**?
- Rust infers type, **`i32`** should be default.
- **HOWEVER!**
- Rust doesn't allow signed integers to be used as array 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!

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut i: i32 = 9;

    loop
    {
        if i < 0 { break; }
        print!("{} ", nums[i]);
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

```
C:\_RustCode>rustc main.rs
error[E0277]: the trait bound `i32: std::slice::SliceIndex<[{integer}]>` is not satisfied
 --> main.rs:9:23
  |
9 |         print!("{} ", nums[i]);
  |                       ^^^^^^^ slice indices are of type `usize` or ranges of `usize`
  |
  = help: the trait `std::slice::SliceIndex<[{integer}]>` is not implemented for `i32`
  = note: required because of the requirements
```

Need to adjust our logic a bit…

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let mut i = 9;

    loop
    {
        print!("{} ", nums[i]);
        if i == 0 { break; }
        i -= 1;
    }
    print!("\nLIFTOFF!\n");
}
```

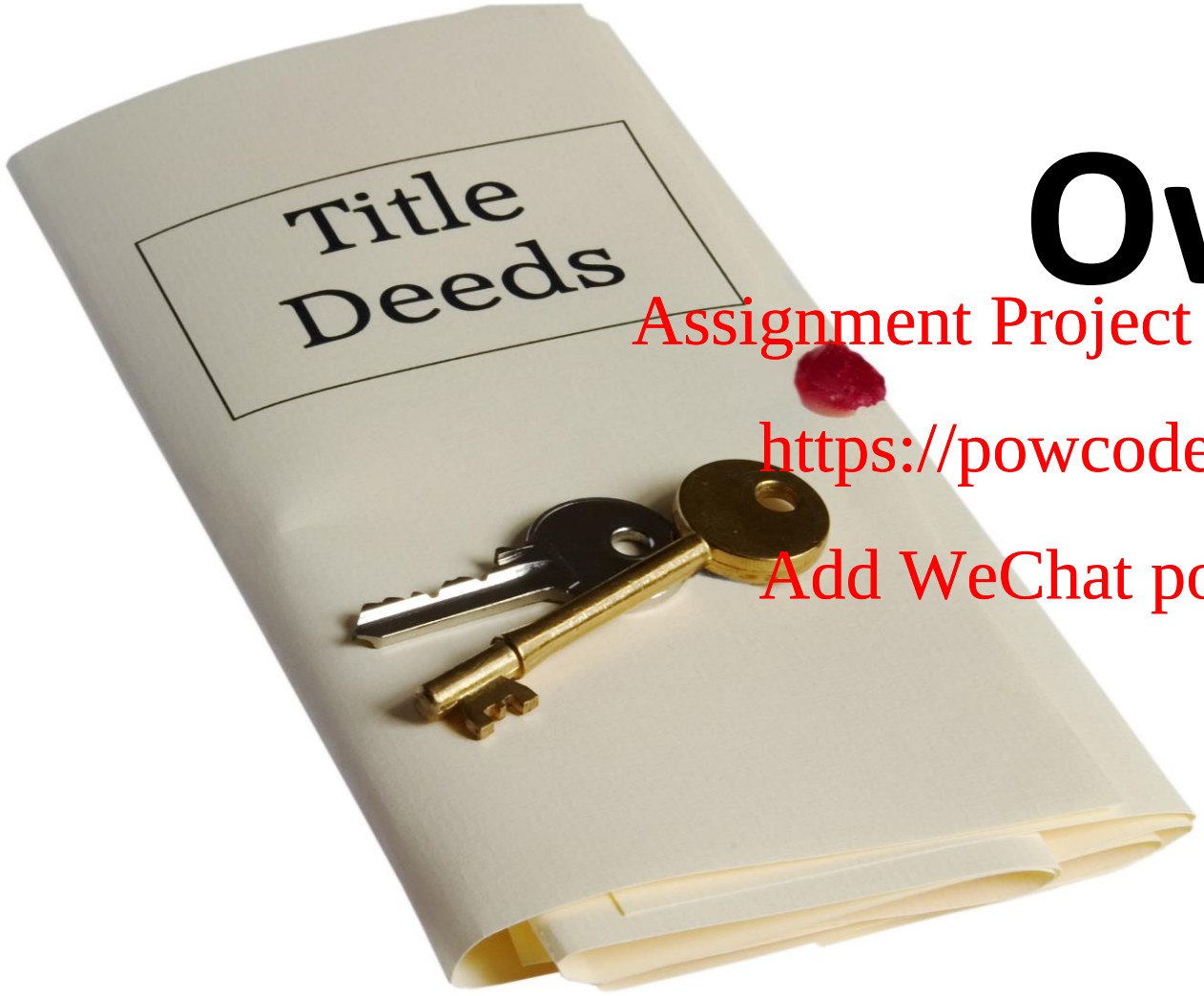

Command Prompt

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

C:\_RustCode>main
10 9 8 7 6 5 4 3 2 1
LIFTOFF!

C:\_RustCode>
```

Moving on….



Ownership

Ownership

Arguably Rust's most unique feature:

- In C, the programmer is responsible for allocating and freeing heap memory. Memory leaks common!

- In Java, garbage collector periodically looks for unused memory and frees it.

- 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, fixed size

Heap:

- Less organized
- Slower access, follow pointers
- Data size can be unknown

- If we dynamically allocate memory in C++, the 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 be one owner at a time.
3. When the owner goes out of scope, the value is dropped.

Scope in Rust

This is normal, nothing new.

```rust
fn main()
{
    // s not valid here, not yet declared
    let s = "hello";   // s is valid from this point forward
    // do stuff with s
}                      // this scope is now over, s is no longer valid
```


- 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

```rust
fn main()
{
    // String literals like this are immutable:
    let s1 = "Hello";

    // String declared thusly can be mutable:
    let mut s2 = String::from("Hello");
    s2.push_str(", World!");

    println!("{}", s1);
    println!("{}", s2);
}
```

- String literals are different from regular strings.
- Their size is fixed, encoded directly into the executable.
- Strings not defined as a literal might have unknown size
- They are stored on the heap.


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

C:\_RustCode>main
Hello
Hello, World!
```

Heap Strings

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

- Calling `String::from` makes a memory request.
- Once again, this is normal behavior. In Java we would say: String s = new String("Hello"); to accomplish the same.


```rust
fn main()
{
    let mut s = String::from("Hello");

    println!("{}", s);
}
```

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, variables become invalid. Too late, waste memory. Do it twice by accident? Also a problem.
- We need to pair one `allocate()` to one `free()`.

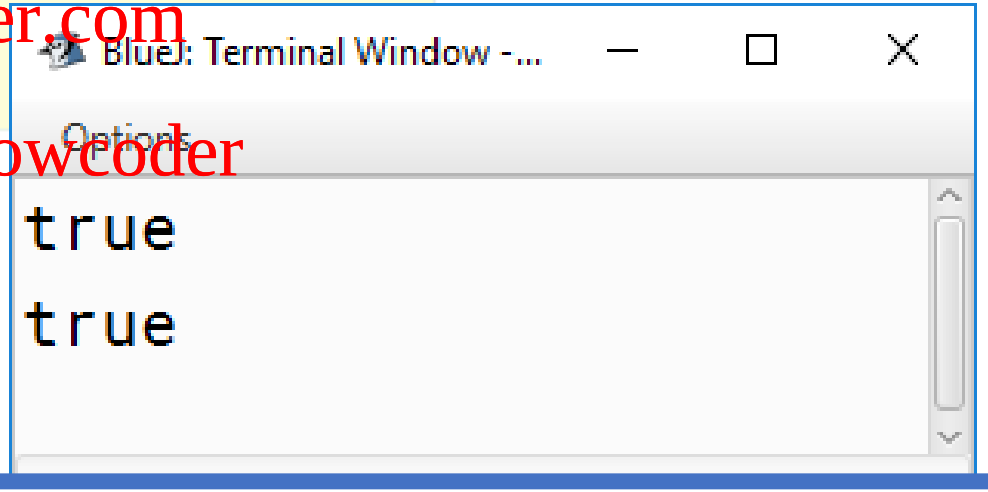
In Rust, memory is automatically returned when the variable that ***owns*** it leaves scope.

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:


```
public static void main(String[] args)
{
    String s1 = new String("Hello");
    String s2 = s1;
    String s3 = s2;

    System.out.println(s1 == s2);
    System.out.println(s2 == s3);
}
```

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 be one owner at a time.
3. When the owner goes out of scope, the 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 happens if we have multiple variables interacting with the same data?


```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

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


On the stack

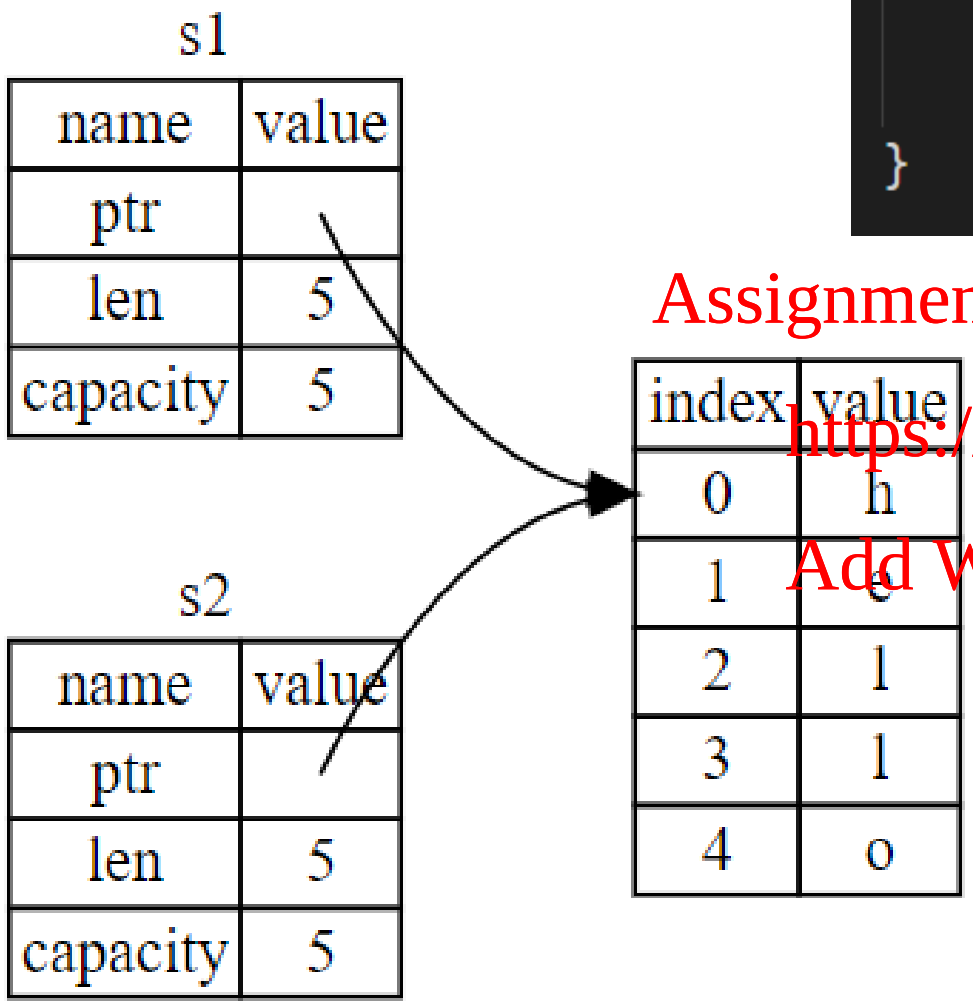
s1

name	value
ptr	→
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

On the heap

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



- Stack data copied; heap data is not.
- Copying heap data is more expensive.
- This is typical 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.

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

    println!("{}", s1);
    println!("{}", s2);
}
```


Command Prompt

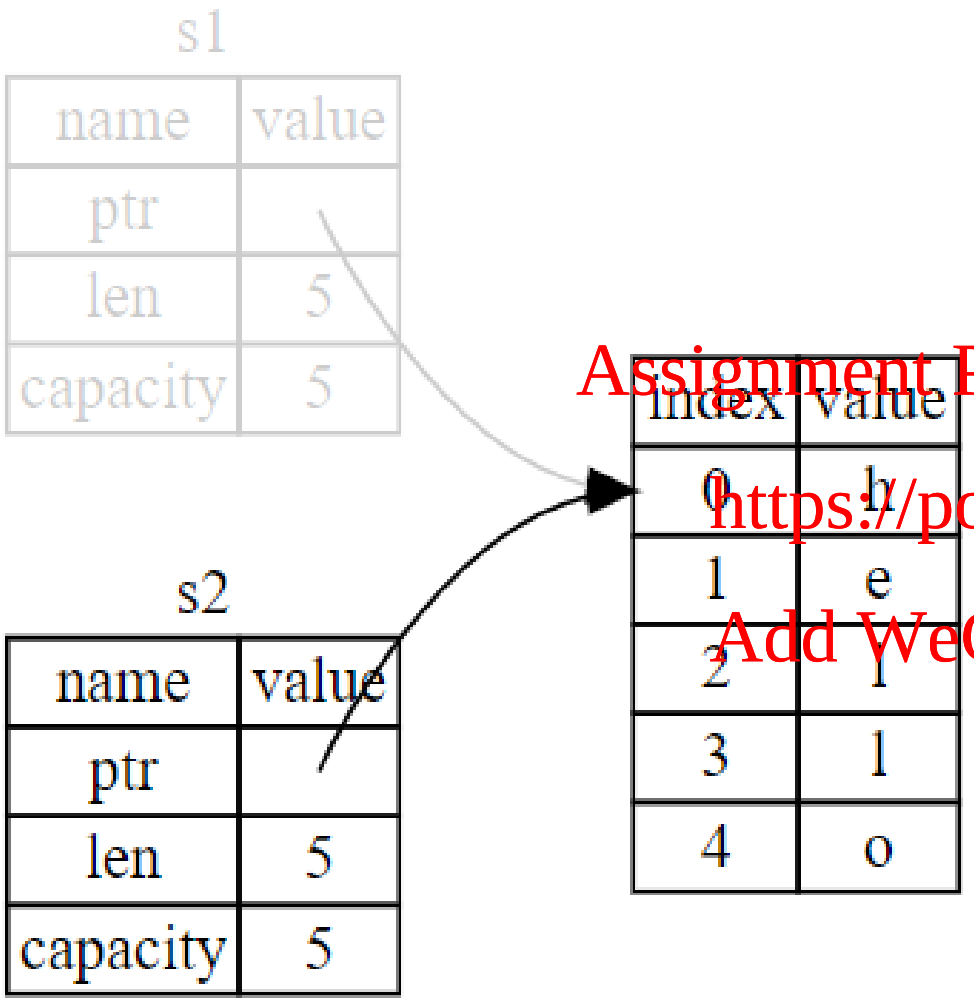
```
C:\_RustCode>rustc main.rs
error[E0382]: use of moved value: `s1`
 --> main.rs:6:20
  |
4 |     let s2 = s1;
  |         -- value moved here
5 |
6 |     println!("{}", s1);
  |                    ^^ value used here after move
  |
```

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.


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

    ~~println!("{}", s1);~~
    println!("{}", s2);
}
```


- When we say `let s2=s1`, s1 becomes invalid.
- Thus, 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;
}
```

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

```
C:\_RustCode>rustc main.rs
error[E0382]: use of moved value: `s1`
 --> main.rs:6:20
  |
4 |     let s2 = s1;
  |         -- value moved here
5 |
6 |     println!("{}", s1);
```

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

Different from a shallow copy, since the old reference is invalidated.

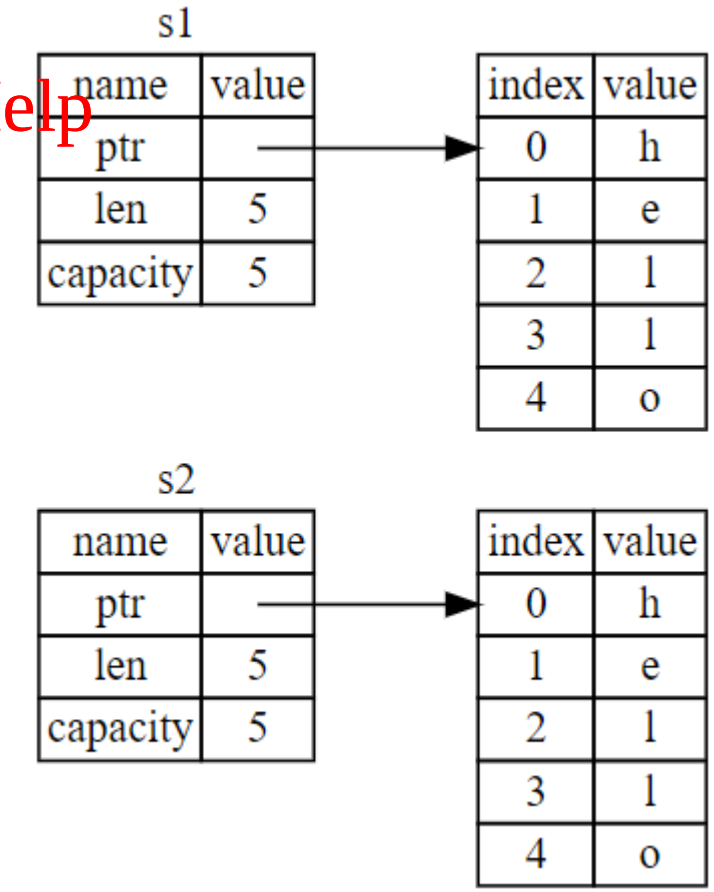
Only one reference can free the heap memory.

clone()

Like most languages, Rust can clone:

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

    println!("{}", s1);
    println!("{}", s2);
}
```

clone()

Like most languages, Rust can clone:

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

    println!("{}", s1);
    println!("{}", s2);
}
```

Command Prompt

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

C:\_RustCode>main
Hello
Hello

C:\_RustCode>
```

Ownership and Functions

Passing an argument moves or copies, just like assignment:

```rust
fn main()
{
    let s = String::from("Weird");

    stringPass(s);

    println!("{}", s);
}

fn stringPass (word: String)
{
    println!("{}", word);
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0382]: use of moved value: `s`
 --> main.rs:7:20
  |
5 |     stringPass(s);
  |                - value moved here
6 |
7 |     println!("{}", s);
  |                    ^ value used here after move
  |
  = note: move occurs because `s` has type `std::string
which does not implement the `Copy` trait
```

Ownership and Functions

Passing an argument moves or copies, just like assignment:

```rust
fn main()
{
    let s = String::from("Weird");

    stringPass(s);

    println!("{}", s);
}

fn stringPass (word: String)
{
```


- Ownership moved from **s** to **word**!
- **s** is now invalid!
- This is very different from any other language we're used to.
- This doesn't 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

```rust
fn main()
{
    let mut s = String::from("Weird");

    s = string_pass(s);

    println!("{}", s);
}

fn string_pass (word: String) -> String
{
    println!("{}", word);
    word
}
```

Command Prompt

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

C:\_RustCode>main
Weird
Weird

C:\_RustCode>
```

Returning Ownership

```rust
fn main()
{
    let mut s = String::from("word");

    s = string_pass(s);

    println!("{}", s);
}

fn string_pass (word: String) -> String
{
    println!("{}", word);
    word
}
```

- Ownership moved from **s** to **word** and back to **s**
- **s** is invalid when we move to **word**
- **word** is invalid when moved to **s**
- Allowed because **s** is mutable.
- When string_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.

```rust
fn main()
{
    let s1 = String::from("Weird");

    let (len, s2) = string_len(s1);

    println!("{} has {} characters", s2, len);
}

fn string_len (word: String) -> (usize, String)
{
    (word.len(), word)
}
```


- **s1** moves to **word**, **word** moves to **s2**
- Return a tuple consisting of the length of word, and word itself.
- **len()** function 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:

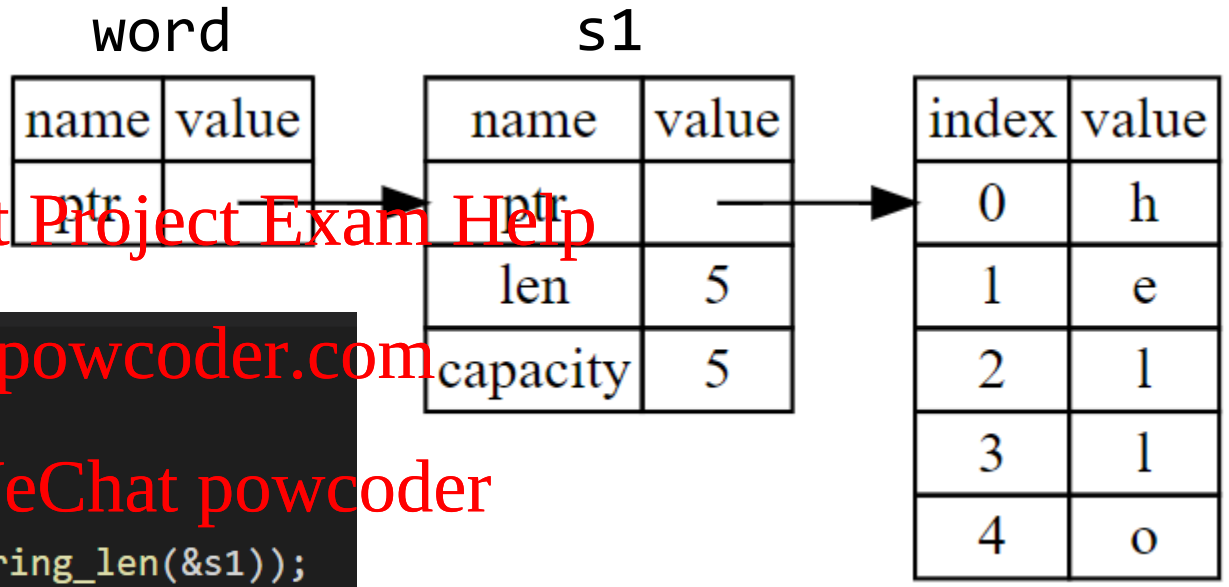
```
fn main()
{
    let s1 = String::from("Weird");

    println!("{} has {} characters", s1, string_len(&s1));
}

fn string_len (word: &String) -> usize
{
    word.len()
}
```


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

Ownership: Moving VS *Borrowing*




```rust
fn main()
{
    let s1 = String::from("Weird");

    println!("{} has {} characters", s1, string_len(&s1));
}

fn string_len (word: &String) -> usize
{
    word.len()
}
```

- 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:

```rust
fn main()
{
    let mut s1 = String::from("Weird");

    println!("{} has {} cha
}

fn string_len (word: &Strin
{
    word.push_str(", or wha
    word.len()
}
```


Command Prompt

```
  = note: #[warn(unused_mut)] on by default
error[E0596]: cannot borrow immutable borrowed content `*word` as
 mutable
  --> main.rs:10:5
   |
8  | fn string_len (word: &String) -> usize
   |                      ------- use `&mut String` here to make
mutable
9  | {
10 |     word.push_str(", or what?");
   |     ^^^^ cannot borrow as mutable
```

?

```
------- use `&mut String` here to make mutable
```

```rust
fn main()
{
    let mut s1 = String::from("Weird");
    let len = string_len(&mut s1);

    println!("{} has {} characters", s1, len);
}

fn string_len (word: &mut String) -> usize
{
    word.push_str(", or what?");
    word.len()
}
```


Command Prompt

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

C:\_RustCode>main
Weird, or what? has 15 characters

C:\_RustCode>
```

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

Borrowing Rules

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

```
fn main()
{
    let mut s1 = String::from("Weir
    let r = &mut s1;
    let len = string_len(&mut s1)

    println!("{} has {} characters"
}
```

Command Prompt

```
error[E0499]: cannot borrow `s1` as mutable more than once at a time
 --> main.rs:5:31
  |
4 |     let r = &mut s1;
  |                 -- first mutable borrow occurs here
5 |     let len = string_len(&mut s1);
  |                          ^^ second mutable borrow occurs here
...
8 | }
  | - first borrow ends here
```


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

Borrowing Rules

Can only have one mutable borrow at a time:

```rust
fn main()
{
    let mut s1 = String::from("Weird");
    let r3 = &mut s1;

    s1.push_str(" test1 ");
    r3.push_str(" test2 ");
}
```

- push_str must make mutable borrow of s1
- Not allowed!

Select Command Prompt

```
C:\RustCode>rustc main.rs
error[E0499]: cannot borrow `s1` as mutable more than once at a time
 --> main.rs:6:5
  |
4 |     let r3 = &mut s1;
  |                   -- first mutable borrow occurs here
5 |
6 |     s1.push_str(" test1 ");
  |     ^^ second mutable borrow occurs here
```

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

```
fn main()
{
    let mut s1 = String::from("Weird");

    {
        let r1 = &mut s1;
    }

    let r2 = &mut s1;
}
```

Scope of r1

Scope of r2

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

```rust
fn main()
{
    let mut s1 = String::from("Weird");
    s1.push_str(" test1 ");

    let r3 = &mut s1;
    r3.push_str(" test2 ");

    println!("{}", r3);
}
```


Command Prompt

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

C:\_RustCode>main
Weird test1  test2

C:\_RustCode>
```

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

Borrowing Rules

Using an immutably borrowed value prevents mutable borrow:

```rust
fn main()
{
    let mut word = String::fro
    let r1 = &word;

    word.push_str(", or what?"

    println!("{}", r1);
}
```


Windows PowerShell

```
PS D:\GoogleDrive\Teaching - Ryerson\(C)CPS 506\Resources\Code\R
error[E0596]: cannot borrow `word` as mutable because it is also
 --> borrow.rs:8:5
  |
6 |     let r1 = &word;
  |              ----- immutable borrow occurs here
7 |
8 |     word.push_str(", or what?");
  |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ mutable borrow occurs here
9 |     println!("{}", r1);
  |                    -- immutable borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0596`.
PS D:\GoogleDrive\Teaching - Ryerson\(C)CPS 506\Resources\Code\R
```

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 any number of immutable borrows

These restrictions keep mutation under control

Slices

Slices

Reference to a subset of an array

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8];

    let tail = &nums[4..8];

    for n in tail.iter() {
        print!("{} ", n);
    }
}
```


Command Prompt

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

C:\_RustCode>main
5 6 7 8
C:\_RustCode>
```

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

Slices, Arguments, Functions

```rust
fn main()
{
    let nums = [1, 2, 3, 4, 5, 6, 7, 8];
    let subset = get_slice(&nums, 1, 5);

    for n in subset.iter() {
        print!("{} ", n);
    }
}

fn get_slice(a: &[i32], s: usize, e: usize) -> &[i32]
{
    &a[s..e]
}
```

- **Reminder:** indexes must be **usize**
- Pass in reference to array
- Return slice (reference to subarray)
- Array only exists once in memory
- **subset** and **nums** point to different parts of the same memory.

String Slices

… are a little bit different.

```rust
fn main()
{
    let msg = String::from("Hello, World!");
    let hello = &msg[..5];  // same as &msg[0..5]
    let world = &msg[7..];  // same as &msg[7..msg.len()]

    println!("{}", hello);
    println!("{}", world);
}
```


Command Prompt

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

C:\_RustCode>main
Hello
World!

C:\_RustCode>
```

Normal so far

String Slice Type

```rust
fn main()
{
    let msg = String::from("Hello, World!");

    let slc = get_slice(&msg, 0, 5);

    println!("{}", slc);
}

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

- &str is a reference to a string slice
- &String is a reference to a String
- String VS string slice: different types
- Other than that, the function works the same as with numeric arrays.
- A string slice is effectively a ***read-only*** view of a String.

String Slice Type

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


Better to do this:

```
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 fact, string literals are ***slices***:


```
fn main()
{
    let msg = "Hello, World!";
}
```

- The 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:

- *Rust is designed to be memory safe*
- *Null or dangling pointers are not permitted.*

Dangling References

Rust prevents them:

```rust
fn main()
{
    let ref_to_nothing = dangle();
}

fn dangle() -> &String
{
    let s = String::from("Hello");
    &s
}
```


dangle()

- Create String s
- Return 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:

```rust
fn main()
{
    let ref_to_nothing = dang
}

fn dangle() -> &String
{
    let s = String::from("Hel
    &s
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0106]: missing lifetime specifier
 --> main.rs:6:16
6   fn dangle() -> &String
                   ^ expected lifetime parameter
  |
  = help: this function's return type contains a bo
rrowed value, but there is no value for it to be bo
rrowed from
  = help: consider giving it a 'static lifetime

error: aborting due to previous error
```

Lifetime?

Lifetime is a very distinct feature of Rust:

Every reference in Rust has *lifetime*

The lifetime of a reference is the scope for which that reference is valid.

Lifetimes are typically implicit and inferred, but can be defined explicitly

Just like variable types!

Example

```rust
fn main()
{
    let r: &i32;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

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


```
C:\_RustCode>rustc main.rs
error[E0597]: `x` does not live long enough
  --> main.rs:7:14
   |
7  |         r = &x;
   |              ^ borrowed value does not live long e
nough
8  |     }
   |     - `x` dropped here while still borrowed
...
11 | }
```

The Borrow Checker

- The Rust compiler has a “Borrow Checker” that compares scope to determine if borrows are valid
- If one variable borrows another, the variable being borrowed must have a lifetime at least as long as the variable doing the borrowing.

What happens if the borrow checker gets confused?

Generic Lifetimes

Consider:

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1, s2));
}

fn longest (x: &str, y: &str) -> &str {
    if x.len() > y.len() { x }
    else { y }
}
```


Simple program:

- Function accepts two string slices, returns the slice that is longer.
- Recall that slices are just references
- There's no ownership changing here
- No moves

Generic Lifetimes

Consider:

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1,
}

fn longest (x: &str, y: &str)
    if x.len() > y.len() { x }
    else { y }
}
```


Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0106]: missing lifetime specifier
 --> main.rs:9:34
9 | fn longest (x: &str, y: &str) -> &str {
  |                                  ^ expected lifetime
 parameter
  |
  = help: this function's return type contains a borrowe
d value, but the signature does not say whether it is bo
rrowed from `x` or `y`

error: aborting due to previous error
```

Generic Lifetimes

```
  |
  = help: this function's return type contains a borrowe
d value, but the signature does not say whether it is bo
rrowed from `x` or `y`
```


The Borrow Checker can't determine lifetime of the return 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

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1, s2));
}

fn longest (x: &str, y: &str) -> &str {
    if x.len() > y.len() { x }
    else { y }
}
```


- We as programmers know that this function is perfectly safe.
- x, y refer to string literals which live the entire duration of the program.
- **HOWEVER**
- 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:

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1,
}

fn longest (x: &str, y: &str)
    x
    //if x.len() > y.len() { x
    //else { y }
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0106]: missing lifetime specifier
 --> main.rs:9:34
  |
9 | fn longest (x: &str, y: &str) -> &str {
  |                                  ^ expected lifetime
 parameter
  |
  = help: this function's return type contains a borrowe
d value, but the signature does not say whether it is bo
rrowed from `x` or `y`

error: aborting due to previous error
```

Lifetime Annotation Syntax

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

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1, s2));
}

fn longest<'a> (x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x }
    else { y }
}
```


Specify generic lifetime

- Similar to generic type: **<T>**
- **<'a>** specifies a generic lifetime, **a**
- **&'a** says this reference has lifetime **a**

Command Prompt

```
C:\_RustCode>main
abcde

C:\_RustCode>
```

Generic Lifetimes

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1, s2));
}

fn longest<'a> (x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x }
    else { y }
}
```

What does mean precisely?

- The function accepts two arguments
- Both live at least as long as lifetime **a**
- Also, the string slice returned will live at least as long as lifetime **a**
- We don't know what **a** is, just that both arguments and return value have the same lifetime.

Generic Lifetimes

```rust
fn main()
{
    let s1 = "abcde";
    let s2 = "abc";

    println!("{}", longest(s1, s2));
}

fn longest<'a> (x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() { x }
    else { y }
}
```

However!

- We're ***NOT*** actually changing any lifetimes!
- We're just explicitly indicating them to help the confused Borrow Checker.
- The borrow checker will reject any values that don't adhere to these constraints.

So how can we break this?

Consider

```rust
fn main()
{
    let s1 = "abc";
    {
        let s2 = "abcde";
        let s3 = longest(s1, s2);

        println!("{}", s3);
    }
}

fn longest<'a> (x: &'a str, y: &'a str) -> &'a str
{
    if x.len() > y.len() { x }
    else { y }
}
```

- Lifetime of **s1** is different from **s2** and **s3**.
- Lifetime **'a** is the scope in which **x** and **y** are both valid. I.e., when **s1** and **s2** are valid.
- When we last use **s3**, **s1** and **s2** are valid.
- Thus, the borrow checker accepts this code.
- **s3** references something that is valid until after the last time **s3** is used.

Now This:

```rust
fn main()
{
    let s1 = "abc";
    let s3;
    {
        let s2 = "abcde";
        s3 = longest(s1, s2);
    }
    println!("{}", s3)
}

fn longest<'a> (x: &'a str, y: &'a str) -> &'a str
{
    if x.len() > y.len() { x }
    else { y }
}
```

- 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.
- Although ***in this case*** it doesn't matter, because we've declared both s1 and s2 as string slices.
- Slices aren't on the heap, and thus references to them will always be valid.

Oops. Let's try again with Strings instead…

Command Prompt

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

C:\_RustCode>main
abcde
```

```rust
fn main()
{
    let s1 = String::from("ab
    let s3;
    {
        let s2 = String::from
        s3 = longest(s1.as_st
    }
    println!("{}", s3);
}

fn longest<'a> (x: &'a str, y
{
    if x.len() > y.len() { x
    else { y }
}
```

Command Prompt

```
C:\_RustCode>rustc main.rs
error[E0597]: `s2` does not live long enough
  --> main.rs:7:35
   |
7  |         s3 = longest(s1.as_str(), s2.as_str());
   |                                   ^^ borrowed value
   |                                   does not live long enough
8  |     }
   |     -  `s2` dropped here while still borrowed
9  |     println!("{}", s3);
10 | }
   - borrowed value needs to live until here
```

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.


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


This is fine, albeit 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 added them as patterns for the compiler to recognize without requiring explicit lifetime annotations.


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

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

Lifetime Considerations

The compiler first checks its list of known patterns

If none are found, we get a compile error such as we've been seeing

What are these patterns?

Lifetime Inference Rules

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


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


is seen as:

```
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 ***o*Ac**ingut reference parameter, it is assigned the same lifetime as any output references.


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

Is seen as:

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

Lifetime Inference Rules

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

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

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


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:

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

```rust
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.
2. If there is ***one*** input reference parameter, it is assigned the same lifetime as any output references.
3. If there 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:

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

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

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

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


```rust
fn main()
{
    let _x: &'static str = "I AM FOREVER";
    let _y = "I am also forever...";
}
```

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 the program? Probably not.
- It's a lazy solution, much like adding dozens of global variables to avoid using pointers or references.

Fantastic Rust Reference:

https://doc.rust-lang.org/book/second-edition/