Week 03 Weekly Exercises

Objectives

- Understand and practice Rust's borrowing rules
- Practice use of basic and complex lifetimes
- Understand the different types of smart pointers, their strengths and weaknesses

Activities To Be Completed

The following is a list of all the markedactivities available to complete this week...

- **My first Borrow!**
- **Pusheen**
- **Annotate Lifetimes**
- **Type Lifetimes**
- **Build A (Christmas) Tree with Boxes**

The following practice activities are optional and are not **marked, or required** to be completed for the week.

- **First Home Buyers**
- **Pokedex and Lifetimes!**
- **My Caesar**

Preparation

Before attempting the weekly exercises you should re-read the relevant lecture slides and their accompanying examples.

Getting Started

Create a new directory for this week's exercises called `lab03`, change to this directory, and fetch the provided code for this week by running these commands:

```
$ mkdir lab03
$ cd lab03
$ 6991 fetch lab 03
```

Or, if you're not working on CSE, you can download the provided code as a tar file.

EXERCISE:
My first Borrow!

Outcomes

- Complete a basic immutable borrow
- Understand the difference between moved and borrowed values
- Exposure to common ownership compiler errors

In this exercise we will practice an extremely simple borrow!

You have been given a start code in the form of a crate `my_first_borrow`.

Your job is to make the program compile! There are two given TODOs in the code.

You should not need to modify the code in any other places than the TODOs.

When your code compiles, it should look like:

```
$  6991 cargo run
   Compiling my_first_borrow v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.35s
     Running `target/debug/my_first_borrow`
inside print_strings: hello, world!
I want to use these strings! hello, world!
```

HINT:

Do not use the `clone` method.

You should only need to insert four characters to make the program compile.

This exercise is not about the idiomatic use of `&str` and `String`.

As such, you should not need to use the type `&str` in your solution.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 4 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Pusheen

Outcomes

- Introduction to mutable references
- Understanding of Rust's multiple mutable reference rules
- Theory of ownership

In this exercise, you are given a basic crate `pusheen` which does not compile!

Your task consists of three steps:

- Write a comment explaining why this program does not compile, and what potential problems Rust is trying to protect you from.
- Fix the program so that it compiles.
- Write a comment explaining how you enabled the program to compile.

The above will be manually marked, so please ensure that you have completed all three steps. There will be an opportunity to discuss this with a tutor and get marked during your week 4 workshop. If you cannot attend, your work will instead be manually marked (offline) during week 5.

HINT:

You should only have to change a few lines of code! This exercise **should not take long**, if you get stuck - ask on the course forum! The documentation for the `mutable references` section in the Book may be useful.

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 4 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:
Annotate Lifetimes

The following activity is taken verbatim from exercise 2 of LifetimeKata - a set of open source exercises written by Tom, which you can use as a resource to learn about rust lifetimes.

You **should** read up to and including chapter two **before attempting this exericse**. This exercise will be much easier having done that. For marks in COMP6991, all you need to do is complete this exercise.

In this exercise, your exercise in this section is to annotate lifetimes on two provided functions:

1. Identity

The first function is called identity. It takes in a reference, and returns the same reference! It is a very simple function, but it is a good starting point for your first lifetime annotation.

2. Split

The second function is called split. It takes in a reference to some string (`text`), and a reference to some delimiter string (`delimiter`).

It returns a vector of references to substrings of the **original string**.

For both functions, you you will need to:

- decide how many lifetime parameters are necessary
- name each of those lifetime parameters, and put them inside < angle brackets > after the function's name.
- annotate every reference with the appropriate lifetime
- check the code compiles (`6991 cargo check`)
- think about what region of code each lifetime could be

You will notice each function has the `#[lifetimes_required(!)]` annotation. You will need to leave it there to complete this exercise. This instructs the compiler to throw an error whenever you miss a lifetime; even if the compiler doesn't need the lifetime.

WARNING:

ONLY modify lines 11 and 27 of the starter code.

NOTE:

You may notice this exercise is not in a `main.rs` file.

This is because we are using a special type of test called a doctest - a test that is embedded in the documentation of a function.

We will learn more about doctests and `rustdoc` next week.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 4 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:
Type Lifetimes

The following activity is taken verbatim from exercise 5 of LifetimeKata - a set of open source exercises written by Tom, which you can use as a resource to learn about rust lifetimes.

You **should** read up to and including chapter five **before attempting this exericse**. This exercise will be much easier having done that. For marks in COMP6991, all you need to do is complete this exercise.

In this exercise, we will be modifying a small program which finds the unique words between two strings.

At the moment, it does not have any lifetime annotations, and therefore does not compile.

Our goal is to return a struct that contains all the unique words from the first string, and all the unique words from the second string. They should have separate lifetimes.

For the struct, and the function, you you will need to:

- decide how many lifetime parameters are necessary
- name each of those lifetime parameters, and put them inside < angle brackets > after the function's name.
- annotate every reference with the appropriate lifetime
- check the code compiles (`6991 cargo check`)
- think about what region of code each lifetime could be

> WARNING:
>
> ONLY modify above line 14.

> NOTE:
>
> You may notice this exercise also has a tests module.
> This is where we have unit tests for the code. You don't need to understand this.
>
> To learn more about unit tests and what specifically the code is doing, you can visit the rust book chapter on testing. We will learn more about tests and modules next week.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 4 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

EXERCISE:

Build A (Christmas) Tree with Boxes

Objectives

- Usage of the `Box` smart pointer
- Exposure to the `ron` crate
- Problem solving with complex references

Tom is getting ready for Christmas! He is simulating a custom Christmas tree he is building. He needs your help to find the average brightness of his simulated Christmas tree.

The lights on this christmas tree form a binary tree; each light has up to two lights that it controls below it. We call those lights "left" and "right". Each light also has a brightness value, between 0 (off), and 100 (keep Zac and Shrey up at night). Tom is starting his tree with just one light, at brightness 0.

We can control the tree by sending it a vector of instructions. In this exercise, we have given you code to scan in this vector from a line of standard input, using rusty object notation There are only four instructions the tree supports:

- `Set(i32)`: Overwrite the current light to the given brightness.
- `Left`: Move to the left from the current light. If it does not exist, install (i.e. create) a new light to the left with 0 brightness.
- `Right`: Move to the right from the current light. If it does not exist, install (i.e. create) a new light to the right with 0 brightness.
- `Reset`: Set the current node to the top-most node of the tree.

Our task is to simulate this unusual Christmas tree, and calculate it's final average brightness.

Here is an example, and a diagram to help explain. You are not required to produce the diagram, it's just to help understand.

```
$  6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/christmas_tree`
[Set(75), Left, Set(33), Reset, Right, Set(67), Left, Set(25)]
50
```

1. Initial State

```
*0
```

2. Set(75)

```
*75
```

3. Left

```
  75
 /
*0
```

4. Set(33)

```
   75
  /
*33
```

5. Reset

```
  *75
  /
33
```

6. Right

```
    75
   /  \
 33    *0
```

7. Set(67)

```
    75
   /  \
 33    *67
```

8. Left

```
      75
    /    \
  33      67
         /
       *0
```

9. Set(25)

```
      75
    /    \
  33      67
         /
      *25
```

10. Calculation

33 + 75 + 67 + 25 / 4 = 50

```
$  6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/christmas_tree`
[Left, Set(100), Reset, Right, Set(100)]
66
```

```
$  6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.01s
     Running `target/debug/christmas_tree`
[Left, Left, Reset, Left, Right, Reset, Right, Left, Reset, Right, Right]
0
```

HINT:

The documentation for the Box type may be useful.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

When you are finished working on this exercise, you must submit your work by running `give`:

```
$ 6991 give-crate
```

The due date for this exercise is **Week 4 Wednesday 21:00:00**.

Note that this is an individual exercise; the work you submit with `give` must be entirely your own.

(OPTIONAL) EXERCISE: First Home Buyers

Objectives

- Practical usage of mutable references
- Exposure to simple derive macros
- Exposure more usages of enums

In this exercise, you are given some starter code in the crate `first_home_buyers` .

The starter code defines the `House` struct, the `Owner` enum and the `Person` struct.

Your job finish the four TODO's, such that our program will compile and run successfully, and the output will match the expected output.

```
$  6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.36s
     Running `target/debug/first_home_buyers`
House1 is owned by: Individual(Person { name: "John", age: 34 })
House2 is owned by: Bank("Bank of Melbourne")
House3 is owned by: Bank("Bank of Melbourne")
```

Due to the problem setup, there will probably be warnings. You can ignore these!

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

(OPTIONAL) EXERCISE:

Pokedex and Lifetimes!

Objectives

- Exposure to a non-trivial lifetime problem
- Understand the importance of **why** rust uses lifetimes to validate references
- Reading and analysing given Rust code

> *"Living is using time given to you. You cannot recall lost time."*
> **an NPC from a Pokemon Game, probably explaning how awesome Rust is.**

In this exercise, we have written some code to query the Pokedex -- a list of every Pokemon! (If you don't know what a pokedex is, think of it like an encyclopedia, but for fictional creatures from the Pokemon universe). Our code, however, does not compile. We have ignored the fact that our code tries to have multiple ownership in many places. So, we need your help to fix the lifetimes and borrowing in this code.

You will at least need to:

- Replace uses of the `Pokemon` type with references.
- Replace uses of the `String` type with references.

However you can expect that more changes to the code will be required.

You should not:

- Construct any extra `String`s.
- Construct any extra `Pokemon` structs.
- Edit the code marked with "DO NOT EDIT THIS CODE" -- if you want to, you have misunderstood the task.

You should also think about, from a compilers point of view, why are lifetimes needed here?

NOTE:

This exercise is doable without writing any new **lines of code**, but you may add lines of code if you wish. For reference, the sample solution has changes on only 7 lines.

When your program is behaving correctly, the user should be able to type in multiple queries (each being a string representing part of a name in English, Chinese or Japanese), and the program should print out all pokemon which matched any query, and which queries they matched:

```
$ 6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/pokedex`

wit
sha
フシギ
妙蛙
Clawitzer            : wit;
Oshawott             : sha;
Musharna             : sha;
Mienshao             : sha;
Bisharp              : sha;
Marshadow            : sha;
Bulbasaur            : フシギ; 妙蛙;
Ivysaur              : フシギ; 妙蛙;
Venusaur             : フシギ; 妙蛙;
```

HINT:

This is a small, yet complicated exercise, as lifetimes are usually a tricky subject for most Rust programmers when they first start. Linked below are some amazing resources on lifetimes:

- The book
- Easy Rust
- Rust by Example: lifetimes
- Ryan Levick: Understanding lifetimes
- Slightly more advanced: Crust of Rust: Lifetime Annotations

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

(OPTIONAL) EXERCISE:

My Caesar

In this activity, your task is to write a program that encrypts a message using a Caesar cipher.

The program should read in a single command-line argument, which is the number of positions to shift each letter (`i32`). If there is no command-line argument, or if the argument is not a number, the program should use a default shift of `5`.

The program should then iterate over each line of standard input, shifting each letter by the specified number of positions. The shifted line should then be printed to standard output.

However, only letters of the alphabet (i.e. `'A'..='Z'`, `'a'..='z'`) should be shifted. All other characters should be printed *as-is*.

Shifted letters should wrap around the alphabet, so that 'A' shifted by 1 becomes 'B', 'Z' shifted by 1 becomes 'A', 'y' shifted by 3 becomes 'b'.

If the provided shift is *negative*, you should treat the shift as operating in the opposite direction.

For example, 'A' shifted by -1 becomes 'Z', 'Z' shifted by -1 becomes 'Y', 'b' shifted by -3 becomes 'X'.

A shift of `0` should not modify the input text at all.

For example:

```
$ 6991 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/my_caesar`
hello!
Shifted ascii by 5 is: mjqqt!
Dr. Taylor Swift is the worlds greatest musician!
Shifted ascii by 5 is: Iw. Yfdqtw Xbnky nx ymj btwqix lwjfyjxy rzxnhnfs!
Ctrl-D
$ 6991 cargo run -- 10
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/my_caesar`
And I know it's long gone and 👋 That magic's not here no more 😭 🫀BREAK
Shifted ascii by 10 is: Kxn S uxyg sd'c vyxq qyxo kxn 👋 Drkd wkqsm'c xyd robo xy wybo 😭 🫀BREAK
T_T crying
Shifted ascii by 10 is: D_D mbisxq
Ctrl-D
```

HINT:

Try attempting this with iterators! Ranges may also be useful.
The documentation for the `chars` method may be useful.

The documentation for the `for_each` method may be useful.

The documentation for the `lines` method may be useful.

The documentation for the `char::from_u32` method may be useful.

When you think your program is working, you can use `autotest` to run some simple automated tests:

```
$ 6991 autotest
```

Submission

When you are finished each exercise make sure you submit your work by running `give`.

You can run `give` multiple times.

Don't submit any exercises you haven't attempted.

If you are working at home, you may find it more convenient to upload your work via give's web interface.

The due date for this week's exercises is **Week 4 Wednesday 21:00:00**.

You cannot obtain marks by e-mailing your code to tutors or lecturers.

Automarking will be run continuously throughout the term, using test cases different to those `autotest` runs for you. (Hint: do your own testing as well as running `autotest`.)

After automarking is run you can view your results here or by running this command on a CSE machine:

```
$ 6991 classrun -sturec
```
