Getting Started with Rust for Elvis

The Elvis project is written primarily in Rust, the next generation systems programming language. Rust is fast and memory safe, but the language itself contains concepts that many current programmers coming from prior languages like Python, Java, C or C++ are not familiar with.

To get started with Rust for Elvis, study The Rust Programming Language, then do these assignments.

Three Bit Logical Calculator

This assignment helps with **basic Rust and functions**.

The logical calculator does math, but with logical operators. In logic, we represent a bit with 0 as false and a bit with 1 as true. The logical operators are NOT, AND and OR. Bitwise logical calculations operate on each bit of the input.

The NOT operator works on just one three-bit argument.

```
NOT 011
  = 100
```

The AND operator works on two three-bit arguments:

```
   011
AND 010
  = 010
```

The OR operator also works on two three-bit arguments:

```
   011
OR  010
  = 011
```

As you can see, the calculator works bitwise. That is, for an operation that requires two arguments, the first character (bit) of the first string is operated on with the first character (bit) of the second string, and the second character (bit) of the first string is operated on with the second character (bit) of the second string, and so on.

Task 1

Write a function `one_bit_NOT` that takes one argument, a single character string that is either "0" or "1". It should perform the NOT operation and return a string with a single character as the result. I.e., if the character argument is "0", it returns a "1"'. If the character argument is "1", it returns a "0".

Task 2

Write a function `three_bit_NOT` that takes one argument, a string that has three characters of either "0" or "1", and uses the `one_bit_NOT` function to calculate the result of applying bitwise logical NOT to the string. It returns a string as a result. For example, three_bit_NOT applied to the string argument "000" should return "111".

Task 3

Write a function `one_bit_OR` that takes two arguments, a and b, which are strings of single characters of either "0" or "1". It then performs the OR operation on the two chars (a OR b) and returns a string with a single character as the result.

Task 4

Write a function `three_bit_OR` that takes two arguments. The arguments are two strings that have exactly three characters of either "0"' or "1"', and uses the `one_bit_OR` function to calculate the result of applying bitwise logical OR to the string. It returns a string as a result.

Task 5

Write a function `one_bit_AND` that takes two arguments, a and b, which are strings of single characters of either "0" or "1". It then performs the AND operation on the two characters (a AND b) and returns a string with a single character as the result.

Task 6

Write a function `three_bit_AND` that takes two arguments. The arguments are two strings that have exactly three characters of either 0 or 1, and uses the `one_bit_AND` function to

calculate the result of applying bitwise logical AND to the string. It returns a string as a result.

Task 7

Write a main function that:

- Prompts the user to enter the operation, either NOT, AND or OR.
- If the operation is NOT, prompt the user to enter one number. Calculate and then display the result.
- If the operation is either OR or AND, prompt the user to enter 2 numbers, and use the calculate the right answer and then print it to the user. You can assume the user will only enter 0's or 1's, and the user only enters 3 digits.

Nice Regular Box

This assignment will help with **standard input and working with vectors and strings**.

Your program will read in a number (no need to prompt the user), and then reads in that number of lines from the terminal. Then the program should print a list of strings formatted in a nice regular box.

Example 1

If the user inputs this:

```
5
Grim visaged war has smooth’d his wrinkled front
And now, instead of mounting barded steeds
To fright the souls of fearful adversaries
He capers nimbly in a lady’s chamber
To the lascivious pleasing of a lute
```

The program should print:

```
++++++++++++++++++++++++++++++++++++++++++++++++++
+ Grim visaged war has smooth’d his wrinkled front +
+ And now, instead of mounting barded steeds       +
+ To fright the souls of fearful adversaries       +
+ He capers nimbly in a lady’s chamber             +
+ To the lascivious pleasing of a lute             +
++++++++++++++++++++++++++++++++++++++++++++++++++
```

Example 2

If the user inputs:

```
2

Avengers, assemble!
```

Note the empty line between `2` and `Avengers, assemble`. The program should print:

```
++++++++++++++++++++++
+                    +
+ Avengers, assemble! +
++++++++++++++++++++++
```

Example 3

If the user inputs:

```
0
```

The program should print:

```
++++
++++
```

Greedy Restaurant Hopping

This assignment will really help you work through **structs, references, lists of references and lifetimes**.

We are hungry college students, and we are developing an app to help us find the closest restaurants.

Restaurant

A `Restaurant` is an object that has a name, a longitude coordinate, and a latitude coordinate. We create a restaurant like this with a constructor with the given name, longitude and latitude.

Distance

Since we're hungry college students, we need to know the distance between two restaurants. Given two restaurants, a and b, it calculates the Cartesian distance (also known as Eucliean distance) between the coordinates of the two restaurants. Recall from geometry that the distance between two points, (x1, y1) and (x2, y2) is given by distance = $\sqrt{((x_1-x_2)_2+(y_1-y_2)_2)}$

- `fn distance(&self, rest: &Restaurant)` -- Return Cartesian distance between the current object and the given restaurant

This calculation works great when the restaurants exist on flat plane, hence this homework's original name was "Flat Earth Restaurants". When you use latitude and longitude as coordinates, the x-coordinate is longitude, and the y-coodinate is latitude.

Closer Restaurant

Writing code is a hungry business, so we need to plan some restaurant hopping. We need to find out which two restaurants are closer to us.

- `fn closer<'a>(&self, rest1: &'a Restaurant, rest2: &'a Restaurant) -> &'a Restaurant` -- Return the restaurant that is closer to this object. If the distances are equal, return rest1. Note that you should return the restaurant object, not the name of the restaurant.

It's easiest to reuse the distance member function you wrote earlier.

Restaurant Hopping

Now create a file called greedy.rs, and implement the `greedy_path()` function as described below. This is not a member function of the Restaurant class. It is a standalone function that uses methods of the Restaurant class.

We want to figure out a way to visit all restaurants. We've decided to follow a greedy algorithm. This means that we will visit the closest restaurant to our location, and then from there visit the next restaurant closest to the restaurant we are at, and so on. A

greedy algorithm just chooses the best option at the moment, even though the total distance traveled may be more than if we were smarter about it.

- `fn greedy_path<'a>(lon: f32, lat: f32, restaurants: &mut Vec<&'a Restaurant>) -> Vec<&'a Restaurant>` -- return a list of restaurants in the order that we want to visit starting from the coordinates given.

If we created three restaurants, r1, r2 and r3, and the greedy path algorithm determines that we should first visit r2, then r1, then r3, the return value would be the list with first r2, then r1, then e3.

The greedy algorithm you implement should go as follows (this is pseudocode that you have to implement in actual code):

- Initialize an empty list of restaurants to represent the greedy list we will return as a result
- While there are still restaurants in the list of restaurants to visit:
- Find the closest restaurant to my current location
- Add the closest restaurant to the greedy list, since that's the greedy choice
- Set my location to the closest restaurant (you're moving to the closest restaurant and visiting it)
- Delete the closest restaurant from the list of restaurants to visit (since you visited this restaurant, you can now remove it from the list of restaurants that you need to visit).
- Return the greedy list as the result once the while loop ends.

Compleat Adventure

This assignment helps with **structs, methods, traits and modules**.

Implement Compleat Adventure but in Rust. Split the implementation into separate files.

Binary Trees in Rust

This assignment helps with the **use of heap allocation, smart pointers with Box, as well as the use of Option**. Note: don't try to do doubly-linked lists in Rust. Rust's ownership rules make doubly linked anything very hard. When something is held by two pointers, it makes the issue of who owns the item difficult. Getting around this requires unsafe Rust, or the use of reference counted types.

Implement a simple binary tree that holds integers in Rust. Use Option<Box<T>> for left and right children.

Implement Tree::`print()` (in order traversal), and Tree::`add()`.

Skip Lists

This assignment helps with the **use of heap allocation, smart pointers with Box, as well as the use of Option**.

Implement a Skip List, but in Rust.

Snake

This assignment helps with advanced topics like **integrating Rust with C and using OS libraries**.

Implement Snake but in Rust. The C++ starter code is here.

You have to figure out how to bind Rust to C and ncurses, and implement the equivalent of the GUI library in Rust.