Advanced programming in Rust - Mandatory 1

Exercise 1

- **Ownership:** Ownership is concept of a variable having the exclusive rights to the singular pointer to a place in memory. At any point in time, only a singular variable can point to a given piece of memory. If the variable is passed to a function as an argument, the the function will take ownership of the pointer.
- **Borrowing:** Giving away the ownership of a pointer for a duration of time, for it to be returned when the pointer goes out of it's new scope.
- **Difference between the two:**

Exercise 2

1)

In this original snippet, as seen bellow, we are not allowed to to `v.push(4)`, since we are already borrowing v as immuteable on the line before. We can't have, that one variable borrows the reference immutably, and then another one mutably. That breaks the assumptions that `first` borrows the reference under.

```
fn main() {
    let mut v = vec![1, 2, 3];
    let first = &v[0];
    v.push(4);
    println!("{}", first);
}
```

2)

Observe this new version of the same snippet:

```
fn main() {
    let mut v = vec![1, 2, 3];
    let first = v[0];
    v.push(4);
    println!("{}", first);
}
```

In the above snippet, first does not borrow v, but is assigned the value of the integer at index 0 of the vector. As such, we are later allowed to borrow it mutable on the next line. Another solution would have been to move the `first` declaration to the next line, since this would make `v.push(4)` would occour before the immuteable borrow.

Exercise 3

1)

- Program A: Does not compile -> when `s` is passed as an argument to `f()`, then `s` loses ownership without ever regaining it, since `f` does not borrow `s`. Before the function call `s` owns the reference. After the function call `f()` owns it, and since `f()` has completed, it is out of scope.
- Program B: Compiles -> `g()` borrows `s` when `s` is passed as an argument, and in the original scope, the variable `s` will regain ownership once `g()` returns. Before the function call `s` owns the reference. Same after.
- Program C: Compiles -> `h()` correctly borrows `s` as a muteable reference before mutating it with `push_str()`. `s` is also given as a muteable reference to `h()` and as such, this is completely legal. Before the function call `s` owns the reference. Same afterwards.

Exercise 4

The below version ensures `main` retains access to s, no cloning occours and behaviour is unchanged.

```
fn process(s: &String) -> usize {
    println!("{}", s);
    s.len()
}

fn main() {
    let s = String::from("rust");
    let n = process(&s);
}
```

In this second version, `process` borrows the s from main, and upon doing it's thing, it returns ownership to main.

Exercise 5

Part A

```
fn main() {
    let data = vec![1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 6];
    let longest_subsequence = longest_increasing_run_owned(data);

    for number in longest_subsequence {
        println!("{}", number);
    }
}

fn longest_increasing_run_owned(data: Vec<i32>) -> Vec<i32> {
    let mut current_sequence: Vec<i32> = Vec::new();
    let mut sequences: Vec<Vec<i32>> = Vec::new();

    for i in 0..data.len() {
        let current_element = data[i];
        let previous_element = current_sequence.last();


        match previous_element {
            None => current_sequence.push(current_element),
            Some(num) => {
                if current_element > *num {
                    current_sequence.push(current_element);
                } else {
                    sequences.push(current_sequence);
                    current_sequence = vec![current_element];
                }
            }
        }
    }

    sequences.push(current_sequence);

    let mut longest: Vec<i32> = Vec::new();

    for sequence in sequences {
        if sequence.len() > longest.len() {
            longest = sequence;
        }
    }

    longest
}
```

This versions finds the longest subsequence and returns it.

Part B

This version does the same, but with index pointers, lengths and a slice at the end.

```
fn main() {
    let data = [1, 2, 3, 4, 1, 2, 3, 4, 5, 6];
    let longest_subsequence = longest_increasing_run_owned(&data);

    for number in longest_subsequence {
        println!("{}", number);
    }
}

fn longest_increasing_run(data: &[i32]) -> &[i32] {
    let mut current_start = 0;
    let mut current_length = 1;
    let mut longest_start = 0;
    let mut longest_length = 1;

    for i in 1..data.len() {



        if data[i] > data[i - 1] {
            current_length += 1;

            if current_length > longest_length {
                longest_start = current_start;
                longest_length = current_length;
            }
        } else {
            current_start = i;
            current_length = 1;
        }
    }

    &data[longest_start..longest_start + longest_length]
}
```