⚠️ Heads up! You're looking at an old class website. Click here for the latest version of this class. ⚠️

Week 1 Exercises: Hello world

Welcome to CS 110L! We're so glad you're here.

Purpose

This week's exercises are designed to get you comfortable compiling/running Rust code and using basic Rust syntax. The best way to learn any language (human, computer, or otherwise) is through immersion, so you can consider this your study abroad in Rustland. Make sure to post about it on Insta, get some Döner Kebap, and enjoy the nightlife. We hope that this exercise will let us hit the ground running next week when we discuss concepts that you may not have seen yet in the languages you've studied.

We'll include some advice for the assignment here, but you may also have to look through the Rust docs, Stack Overflow, etc. as well. We (Ryan and Armin) are available to you via Slack to help with the assignment. You are also welcome to reach out to your fellow peers and work together (you must however code up your own solutions for this assignment).

Due date: Tuesday, April 14, 10:30am (Pacific time)

This assignment should take 1-3 hours to complete. Even though this assignment is only meant to acclimate you to the Rust language, we expect it may be challenging; Rust is a pretty quirky language, and it will take some work to navigate that unfamiliarity. If you hit the 2 hour mark and aren't close to finishing, please let us know so that we can address anything that might be tripping you up.

Part 1: Getting oriented

The first part of this week's exercises will be getting a simple "hello world" program to run!

We have gotten Rust tooling set up for you on myth, so if you would like to develop your code there (as you do in CS 110), you should be good to go. However, if you would like to run your code on your local machine (e.g. because you have a poor internet connection or are on the other side of the world), you will need to install the Rust toolchain.

Regardless of whether you decide to work on myth or on your personal computer, you should take a moment to get your development environment set up for Rust. Ryan wrote up a list of tips that may be helpful. If you are using vim, I highly recommend installing the Rust plugins from that tips page, as they will improve your experience dramatically. For other editors, do a quick search to see if there are plugins that others recommend installing.

Now, onto getting the starter code! In this class, we will be using Github to manage assignment submissions. Github is an awesome collaboration platform built on `git`, which is a *version control software*. (Version control software allows you to manage different versions of your code; you can save snapshots across different points in time, and if your code ends up breaking, you can go back to a previous snapshot. It's better than saving `code.c`, `code-working.c`, `code-working-copy.c`, `code-final.c`, `code-final-seriously.c`, and `code-final-i-actually-submitted.c`, and then being confused as to what is what.) `git` and Github are standard tools in industry, and if you haven't encountered them before, we think it would be valuable to get some light and friendly exposure to them.

The starter code is available on GitHub here. You can "clone" (download) that repository to your computer:

```
git clone https://github.com/reberhardt7/cs110l-spr-2020-starter-code.git
```

Then, `cd` into `week1/part-1-hello-world`. This directory contains a Rust *package*. You can see the source code in the `src/` directory; check out `src/main.rs`.

Let's try to compile this code! To do this, run the following command:

```
cargo build
```

Cargo is kind of like `make`, but it does quite a bit more. If your project has dependencies, Cargo will automatically take care of downloading and configuring those dependencies. (It does what `npm` does in Javascript land, and what `setup.py` does in Python land.) Cargo can also run automated tests, generate documentation, benchmark your code, and more. We won't talk about this for now, but you will see some of this come in handy later in the quarter.

When you run `cargo build`, Cargo compiles the executable and saves it at `target/debug/hello-word`. Try running that now:

```
🍌 ./target/debug/hello-world
Hello, world!
```

(Yes, my shell prompt is a banana.)

As a convenience, Cargo offers a `run` command that *both* compiles and runs the program in one shot. Try modifying `src/main.rs` to print something new, then run `cargo run` (without running `cargo build`). Cargo will notice that the file has changed, recompile your code, and run the binary for you.

```
🍌 cargo run
   Compiling hello-world v0.1.0
    Finished dev [unoptimized + debuginfo] target(s) in 0.77s
     Running `target/debug/hello-world`
You rock!
```

Congrats! You've run your first Rust program!

Part 2: Rust warmup

Syntax

First, let's take a whirlwind tour of Rust's syntax. Much of this may feel familiar, but Rust has a few quirks when compared to other languages.

Note: these points are expanded from Will Crichton's CS 242 Rust lab handout.

Numeric types in Rust include `i8`, `i16`, `i32`, and `i64` (all of which store signed – positive or negative – numbers), as well as `u8`, `u16`, `u32`, and `u64` (which store unsigned – strictly nonnegative – numbers). The numbers `8`, `16`, and so on indicate how many bits make up a value. This is intended to avoid problems that C encountered because the C specification did not define standard widths for numeric types. This is something you will encounter in assignment 2 in CS 110; today, on most computers, an `int` stores 4 bytes (corresponding to Rust's `i32`), but you will be working with code from the 70s where an `int` only stored two bytes.

To declare a variable, we use the `let` keyword and specify the variable's type:

```
// Declare a variable containing a signed 32-bit number. (Equivalent to
// "int n = 1" in C)
let n: i32 = 1;
```

Rust has a nifty feature called "type inference." Instead of forcing you to declare the type of every variable, Rust allows you to omit the variable's type when the compiler is able to figure out what the type should be. Most Rust code looks like this, and only includes explicit type annotations when the compiler can't figure out which type should be used.

```
let n = 1;
```

Unlike most languages, variables in Rust are constants *by default*. This is intended to reduce errors; if you modify a variable that you didn't intend to (i.e. didn't explicitly mark as mutable), the compiler will give you an error. Add `mut` to make a variable mutable.

```
let mut n = 0;
n = n + 1;  // compiles fine
```

Strings are a little weird in Rust. There are two string types: `&str` and `String`. `&str` is an immutable pointer to a string somewhere in memory. For example:

```
let s: &str = "Hello world";    // the ": &str" annotation is optional
```

Here, the string `Hello world` is being placed in the program's read-only data segment (that's also how it works in C), and `s` is a read-only pointer to that memory.

The `String` type stores a heap-allocated string. You're allowed to mutate `String`s (so long as you use the `mut` keyword).

```
let mut s: String = String::from("Hello "); // "String" type annotation is optior
s.push_str("world!");
```

The first line allocates memory on the heap; `s` is a `String` object containing a pointer to that memory. The second line appends to the heap buffer, reallocating/resizing the buffer if necessary. The memory for the string will be automatically freed after the last line where `s` is used. We'll talk more about the memory allocation/deallocation on Thursday's lecture. (This is sort of similar to how C++ `string` works.)

The easiest way to store collections of things is in a vector:

```
let mut v: Vec<i32> = Vec::new();
v.push(2);
v.push(3);
// Even here, the ": Vec<i32>" type annotation is optional. If you omit it, the
// compiler will look ahead to see how the vector is being used, and from that, i
// can infer the type of the vector elements. Neat!
```

Rust also supports fixed-size arrays. Unlike C, the length of the array is stored as part of the array type. In addition, array accesses are generally bounds-checked at runtime. No buffer overflows, please!

```
let mut arr: [i32; 4] = [0, 2, 4, 8];
arr[0] = -2;
println!("{}", arr[0] + arr[1]);
```

You can iterate over collections using iterators and some very nice, Python-like syntax:

```
for i in v.iter() { // v is the vector from above
    println!("{}", i);
}
```

While loops:

```
while i < 20 {
    i += 1;
}
```

Rust also has a special loop that should be used instead of `while true`:

```
let mut i = 0;
loop {
    i += 1;
    if i == 10 { break; }
}
```

If you're curious why this loop exists, there is some discussion here. Long story short, it helps the compiler to make some assumptions about variable initialization.

Finally, Rust functions are declared like so:

```
// Function with return type i32
fn sum(a: i32, b: i32) -> i32 {
    a + b
}

// Void function (no "->")
fn main() {
    // do stuff...
}
```

There are two potentially surprising things here:

- Unlike variables (where Rust will happily infer the variable type), you are required to specify the return type for functions that return values.
- There's no `return` keyword in the `sum` function! And... it's missing a semicolon!

  These two things actually go together. Rust is an *expression-based* language. In most languages that you are probably used to, there are *expressions* (which evaluate to values) and *statements* (which do not). For example, in C++, the ternary operator is an expression; it evaluates to a value that you can store in a variable:

```
int x = someBool ? 2 : 4;
```

By contrast, `if` statements are statements because they do *not* evaluate to a value. This code doesn't compile:

```
int x = if (someBool) {
    2;
} else {
    4;
}
```

However, in Rust, *everything* is an expression. (This has nothing to do with safety. It is because Rust is heavily influenced by functional programming languages, which are outside the scope of this class.) This is valid Rust code:

```
let x = if someBool { 2 } else { 4 }
```

Functions are sequences of expressions separated by semicolons, evaluating to the value of the last expression. The `sum` function above has a single expression, `a + b`, so the `sum` function will evaluate to (i.e. return) whatever `a + b` ends up being.

If you had included a semicolon and written the following code, you would actually get a compilation error:

```
fn sum(a: i32, b: i32) -> i32 {
    a + b;
}
```

Remember that functions are expressions separated by semicolons. As such, this function actually contains *two* expressions: `a + b` (before the semicolon) and an *empty* expression (after the semicolon). Because the last expression is void, this function ends up returning nothing. The compiler will give you an error because the function was declared with an `i32` return type.

Since everything is an expression, you can end up writing functions like this:

```
fn fib(n: i32) -> i32 {
    if n <= 1 { n } else { fib(n-1) + fib(n-2) }
}
```

This may end up being one of the weirdest things to get used to coming from languages like C, Java, and Python, but it can be an elegant and concise way of writing programs once you acclimate. If you're interested in the influences behind this design, consider taking CS 242 (Programming Languages)! We will talk more about this syntax next week, so don't worry too much if it's confusing.

That's it for syntax we think is crucial for you to know! Let's write some code!

Practice

These exercises were written by Will Crichton for CS 242.

`cd` into `part-2-warmup/` and have a look at `src/main.rs`. There are three functions you should implement:

- Implement `add_n`, which takes a vector of numbers and some number `n`. The function should return a new vector whose elements are the numbers in the original vector `v` with `n` added to each number.
- Implement `add_n_inplace`, which does the same thing as `add_n`, but modifies `v` directly (in place) and does not return anything.
- Implement `dedup` that removes duplicate elements from a vector in-place (i.e. modifies `v` directly). If an element is repeated anywhere in the vector, you should keep the element that appears first. You may want to use a HashSet for this.

At the end of `src/main.rs`, you'll find a series of functions marked `#[test]`. These are *unit tests*, provided to help you verify that your functions work correctly. You can run these tests by running `cargo test`. Think of this as a `sanitycheck` equivalent!

You must write your solutions by yourself, but you should feel free to Google how to do things in Rust. We are also more than happy to answer questions on Slack. This is intended to be a friendly, quick warm-up, so if you find yourself getting hung up on anything, let us know!

Part 3: Hangman

Your goal is to implement a command-line hangman game. The following is an example of a possible run of the game:

```
Welcome to CS110L Hangman!
The word so far is -------
You have guessed the following letters:
You have 5 guesses left
Please guess a letter: r

The word so far is ------r
You have guessed the following letters: r
You have 5 guesses left
Please guess a letter: s
```

```
The word so far is ---s--r
You have guessed the following letters: rs
You have 5 guesses left
Please guess a letter: t

The word so far is ---st-r
You have guessed the following letters: rst
You have 5 guesses left
Please guess a letter: l

The word so far is l--st-r
You have guessed the following letters: rstl
You have 5 guesses left
Please guess a letter: a
Sorry, that letter is not in the word

The word so far is l--st-r
You have guessed the following letters: rstla
You have 4 guesses left
Please guess a letter: b

The word so far is l-bst-r
You have guessed the following letters: rstlab
You have 4 guesses left
Please guess a letter: c
Sorry, that letter is not in the word

The word so far is l-bst-r
You have guessed the following letters: rstlabc
You have 3 guesses left
Please guess a letter: o

The word so far is lobst-r
You have guessed the following letters: rstlabco
You have 3 guesses left
Please guess a letter: e

Congratulations you guessed the secret word: lobster!
```

Alternatively, you may not have gotten the word:

```
Welcome to CS110L Hangman!
The word so far is --------
You have guessed the following letters:
You have 5 guesses left
Please guess a letter: a

The word so far is --a-----
You have guessed the following letters: a
You have 5 guesses left
Please guess a letter: b
Sorry, that letter is not in the word

The word so far is --a-----
You have guessed the following letters: ab
You have 4 guesses left
Please guess a letter: c

The word so far is c-a-----
You have guessed the following letters: abc
You have 4 guesses left
Please guess a letter: d
Sorry, that letter is not in the word

The word so far is c-a-----
You have guessed the following letters: abcd
You have 3 guesses left
Please guess a letter: e
Sorry, that letter is not in the word

The word so far is c-a-----
You have guessed the following letters: abcde
You have 2 guesses left
Please guess a letter: f

The word so far is c-a-f---
You have guessed the following letters: abcdef
You have 2 guesses left
Please guess a letter: g
Sorry, that letter is not in the word

The word so far is c-a-f---
You have guessed the following letters: abcdefg
You have 1 guesses left
```

```
Please guess a letter: h

The word so far is c-a-f--h
You have guessed the following letters: abcdefgh
You have 1 guesses left
Please guess a letter: i

The word so far is c-a-fi-h
You have guessed the following letters: abcdefghi
You have 1 guesses left
Please guess a letter: j
Sorry, that letter is not in the word

Sorry, you ran out of guesses!
```

The program exits either when you correctly complete the word or when you run out of guesses.

Advice and Helpful Hints

- Remember that variables are immutable by default. If you plan on changing your variable in the foreseeable future, you need to mark it as mutable, e.g. `let mut counter = 0`.
- You can compare `String`s using `==`.
- Use `println!(...)` to print things to standard out. To print a variable, you can write something like this:

```
println!("Variable contents: {}", some_variable);
```

- To read input from the user, you can write something like this:

```
print!("Please guess a letter: ");
// Make sure the prompt from the previous line gets displayed:
io::stdout()
    .flush()
    .expect("Error flushing stdout.");
let mut guess = String::new();
io::stdin()
    .read_line(&mut guess)
    .expect("Error reading line.");
```

- You aren't expected to match our output exactly. Just make a game that works!
- We don't expect you to worry about error-handling or reprompting. However, doing so will give you better practice with the language.

- We haven't talked about `Option`s or `Result`s yet, which are a big part of Rust and will be the topic of next week's lecture. If you Google how to do things, you may encounter suggestions that involve calling functions like `unwrap()`. You're welcome to use such code, but there is probably a simpler way to do it. Ping us on Slack and we'd be happy to give you a suggestion.
- The Rust compiler is your friend and enemy. Its criticism is often constructive, unlike most C++ compilers :) The Rust team spent a long time trying to make error messages as helpful as possible, and we hope you find them useful.

Part 4: Weekly survey

As you know, this is the first time we're teaching this class, and we want to make it as enjoyable and meaningful of an experience as possible. Please let us know how you're doing using this survey.

When you have submitted the survey, you should see a password. Put this code in `part-4.txt` before submitting.

Submitting your work

As you work, you may want to *commit* your code to save a snapshot of your work. That way, if anything goes wrong, you can always revert to a previous state.

If you have never used git before, you may need to configure it by running these commands:

```
git config --global user.name "Firstname Lastname"
git config --global user.email "yourSunetid@stanford.edu"
```

Once you've done this, you won't need to do it again.

Then, to commit your work, run this command:

```
git commit -am "Type some title here to identify this snapshot!"
```

In order to submit your work, commit it, then run `git push`. This will upload your commits (snapshots) to Github, where we can access them. You can verify that your code is submitted by visiting https://github.com/cs110l/week1-yourSunetid and browsing the code there. You can `git push` as many times as you'd like.

Grading

There are four components to this assignment, each worth 25%. You'll earn the full 25% for each part if we can see that you've made a good-faith effort to complete it.