Data Types and Functions

Goals For Today

- Compound Data Types
- Functions
- Vectors

What are Data Types?

- Data types are the method by which we tell the compiler what type of data it is storing
- Built in to the compiler so users can access with predetermined keywords
- There are primarily two types of data types
  - Scalar Data Types
  - Compound Data Types

Scalar Data Types
- Integers
- Floating Points
- Booleans
- Characters

Compound Data Types
- Tuples
- Arrays

Tuples

- Fixed Length
- Can store values of different types
  - Don't need to declare
- Immutable
- 0 indexed
  - Indexing accessed using the dot notation
- Empty tuples with no values are called units
  - The default return value for functions
  - Its value and type are both ()

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {y}");
}
```

Array

- Fixed Length (unlike some other languages!)
- Must store same type values
  - Implicit declaration allowed
- Mutable
- 0 indexed
  - Access elements using [] indexing
  - Invalid Indexes will throw a trace error

```
fn main() {
    let a = [1, 2, 3, 4, 5];
}
```

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

```
fn main() {
    let a = [1, 2, 3, 4, 5];

    let first = a[0];
    let second = a[1];
}
```

```
let a = [3; 5];
```

```
let a = [3, 3, 3, 3, 3];
```

Functions

- Convention in Rust is functions are all lowercase with underscore separating words
  - Snake Case as opposed to Camel Case
- Order of declaration of functions does not matter
- Keyword ***fn*** indicates a function declaration and ***{}*** indicate function body
- You can pass values into a function as a parameter. Parameters **MUST** have a type declaration
  - Parameters do not need to have the same type
  - Parameters are by default passed by value and immutable, meaning if they are edited, changes are **NOT** reflected outside the function

```
fn main() {
    print_labeled_measurement(5, 'h');
}

fn print_labeled_measurement(value: i32, unit_label: char) {
    println!("The measurement is: {value}{unit_label}");
}
```

Parameters and Return Values

- ***Mut***, and eventually *&* and *** are all part of the parameter type and have to be passed to the function for proper assessment of values
- Functions can return a value after their call
  - If you want your function to return a value, you must specify the type of value it is returning in the declaration
  - Done with the **->** symbol and a return type
  - Can only specify a single type to return
  - If you want to return multiple values, **use a tuple or an array**!

```
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {x}");
}
```

Vectors

- Vectors are not a datatype the compiler recognizes
  - Vectors are classes and must be declared explicitly
  - Implicit declarations will cause errors in your code
- Vectors can only store a single type of value
- Vectors have a variable length and are mutable
- Vectors can be declared with Vec::new() and the vec![] macro
  - These two declarations are **NOT** interchangeable and will require transformation functions to use interchangeably
- Indexed using [] notation

```
let v: Vec<i32> = Vec::new();
```

```
let v = vec![1, 2, 3];
```

Vectors Part 2

- Add elements to a vector using the .push() function
- Access elements using .get() (returns an Option!) or []

```
let mut v = Vec::new();

v.push(5);
v.push(6);
v.push(7);
v.push(8);
```

```
let v = vec![1, 2, 3, 4, 5];

let third: &i32 = &v[2];
```

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

Reminders:

extra/0 credit practice problems that are always open

Homework 1 released this week

Chapter 2 in the Rust Docs. Chapter 8 for Vectors