Data Types

Scalar vs Compound Data Types

Scalar

- A scalar data type is something that has a finite set of possible values following some scale. i.e. The values can be compared to eachother as equal to, or less than, or greater than
- Example Data Types:
- `let x: i32 = 2` // signed integer 32bits, 32 is the default value
    - `let x: i64 = 2` // signed integer 64bits
    - `let x: i128 = 2` // signed integer 128bits
    - `i8`
    - `i16`
- `let x :u32 = 2` // unsigned integer 32bits
    - `let x :u64`
    - `let x :u128`
    - `:u8`
    - `:u16`
- `let x :f32 = 5.0` //float 32bits single precision
    - `let x :f64` //float 64bits double precision
    - **Note** that floats need a decimal point to be a floating point number
- `let x: bool = true = 1` //boolean
    - `let x: bool = false = 0`
- `let x :char = 'c'` //character

Compound Data Types

- A compound data type is a composite data type of either scalar or other compound data types.
- Example Data Types:
    - Tuple: very similar to a struct type in C
        - Fixed length sequence of elements that are immutable
        - Can contain any mix of types
        - Can be mutable by using the `mut` key word
        - Implicit Definition
            - `let x = (1,true,'s')`
        - Explicit Definition
            - `let x: (i32, bool, char) = (1,true,'s')`
        - Tuples type is defined by the type within the sequence
        - Becareful of setting tuple to tuple, types must match exactly

        - Ex.

```
let mut x: (i32,bool, char) = (1,true,'s')
let y: (i8,bool, char) = (1,true, 'C')
x = y // error, i32 and i8 does not match
```

    - However if one tuple is only defined explicitly, and the values **can be of the same type** then the implicit will convert to the explicit
        - Ex.

```
let mut x = (1,true,'s') //implicit
let y: (i8,bool, char) = (1,true, 'C')//explicit
x = y // no error, implicit i32 becomes i8
```

    - accessing tuples is similar to member fields of objects.
    - Example
        - `x.0 -> x[0], x.1-> x[1], x.2-> x[2] etc...`
    - You cannot print a whole tuple at once, you have to print the elements individually
    - **Print** you can print whole tuple using {:?} within the bracket
        - Ex.
        - `println!("x = {:?}", x)`
  - **Note** You cannot add elements to tuples
- Arrays:
  - Fixed length sequence of elements that are immutable
  - Can be mutable by using the `mut` key word
  - **Note** Must be homogeneous unlike the tuple
  - Since arrays are fixed length and have the same type, they can be easily iterated over
  - Explicit initialization:
    - Unlike a tuple, the initialization only needs the type and # of elements
    - Example:
      - `let arr : [i32; 3] = [1,2,3];`
      - **Note** the type and # of elements are separated by semi-colons
    - You cannot initialize an empty array, you need to place some values to start
  - You can access the array the same way in all other languages
    - Ex.
      - `arr[0] -> first element of array`
  - If you want to print the whole array
    - `println!("Array = {:?}", arr)`

Strings

- Strings can be represented in two main ways
    - String literals:
        - `let x : &str = "hello world";`
        - This will set up a string literal
        - Typically immutable strings and not recommended use
    - String objects
        - `let mut x = String::new();//empty string`
        - `let mut y = String::from("hello world");//non-empty declaration`
        - String objects have much more functionality than string literals and is a growable collection. You can use many functions like push(), split(), trim(), len(), replace()
        - To change a string literal into a string object use .to_string() function
            - `let x = "hello world".to_string();`