Vector, Strings, and Hashmaps

- One thing to note about all of these are that they are collections
- They are not default types and are dynamically sized
- They are defined in the heap so ownership rules must be taken into consideration when utilizing them

Vector

- To define a vector
  - `let mut vector : Vec<i32> = Vec::new();`
  - This creates an empty vector with only i32 elements
  - since we do not initialize the vector with any elements rust cannot infer it
- We can push elements in a vector easily
  - `vector.push(32);`
- There is a simple way to initialize a vector using array syntax using the `vec!` macro
  - `let v2 = vec![1,2,3,4]`
    - This macro converts a array into a vector and auto assigns the types
- There are 2 ways to access the elements in a vector
  - Directly reference the index in a vector
    - `let third = &v2[2];`
    - This opens up an error to access nonexistent elements
    - The indexing here is different from array
      - Since array size are immutable, the index is out of range error is a compile time error
      - Vectors are dynamically allocated so the index is checked at run time instead. Therefore this error is harder to catch.
      - Another issue is that we pass an immutable reference to the index. An we cannot have mutable references and immutable references at the same time
        - So until we are done with `third` we cannot modify the vector
      - This is why rust provides a safer way to access the elements
  - There is the `v2.get(index)` method which handles the error gracefully
    - The `get()` method returns an **Option** which represents `Some()` value or `None` value
    - You get `Some()` value if the index exists and a `None` value if the index doesn't exist
    - Use a match to handle these two cases
    - Example:

```
match v2.get(2) {
    Some(third) => println!("Get Third -> {}, {}", third,
v2.get(2).unrwrap()),
    None => println!("Index out of range"),
}
```

```
```

            - Match here will handles both cases
- Iterating over vectors
    - You can iterate over the referenece of the vectors using that direct referencing
    - Example:

      ```
      for i in &mut v2{
          *i += 50;
          println!("Current value: {}", i);

      }
      ```

        - Here we are iterating over **mutable** values of `v2`
        - Two things to note here
            - We can access the values and print them out easily
            - We can modify the values in place
        - We can see the **dereference operator** `*`
            - This operator helps to mutate the values in place but we won't be focusing on this right now.
- Tricking Vectors to use multiple types
    - Vectors typically only take in values of the same type, however we can trick it to take multiple types by using Enums
    - Enums variants can bind different types to each variant but all the variants are of the same Enum
    - By making a vector of Enums we can access different types as with the binded values
    - Example:

      ```
      enum SpreadsheetCell {
          Int(i32),
          Float(f64),
          Text(String),
      }

      let row = vec![
          SpreadsheetCell::Int(3),
          SpreadsheetCell::Text(String::from("blue")),
          SpreadsheetCell::Float(10.12),
      ];

      match &row[0]{
          SpreadsheetCell::Int(value) => {
              println!("Row cell 1: {}", value);
          }
          _ => println!("Not an Int value"),
      ```

```
}
```

- Here we have a vector that is of type `SpreadsheetCell` but a spreadsheet cell can be of any type
- So we make each variant a different type and bind values to them when initializing the vector
- **There is a catch** when pulling a value from an Enum, you need to use a match expression. This makes it a little complicated to analyze values

String

- Strings are very complicated in Rust
- Strings are stored as a collection of UTF-8 encoded bytes
  - UTF-8 is a universal encoding, each character can be 1byte, 2byte, 3bytes or 4bytes
- To emphasize that point, a String is a wrapper around `Vec<u8>` so each element of a string is a byte
- String objects defined with `String` are called **Owned String**
- As a review from slices, String slices are called `&str`
- Also note that `&str` and `&String` are different
- Defining a string
  - You can use the `new` method
    - `let s1 = String::new();`
  - You can make a string slice using &str type
    - `let s1: &s1 = "hello, world";`
  - You can convert a string slice into a owned String
    - `let s1 = s1.to_string();`
  - You can create a owned String from a string slice directly using `from`
    - `let s1 = String::from("hello, world");`
- Just like a vector, a `mut String` can grow and shrink in size
  - You can push a stirng slice using `push_str()`
    - Note that it cannot take in a String object directly because we don't want ownership of the object
  - You can also push a charachter using `push()`
  - Example:

```
let mut s1 = String::from("foo");
s1.push_str("bar");
s1.push('!');
```

    - Just note that to represent a character you use single quotes `''`
  - You can also append strings with the `+` operator

- Example:

  ```
  let s1 = String::from("Hello");
  let s2 = String::from(" World!");
  let s3 = s1 + &s2;
  ```

  - When appending s1 and s2 and putting into s3, you can see there is a reference for s2 and not s1
  - s1 is giving away **ownership** of its value "Hello" and then we **borrow** the characters in s2 and append it to s3
  - You cannot append two string slices or borrowed strings with the `+` operator
  - The `+` operator only works with `String + &String` or `String + &str`
- An easier way of doing it is with the `format!()` macro
  - The `format!()` macro can format a string by **copying** rather than taking ownership or borrowing
  - Example:
  - `let s3 = format!("{}{}",s1, s2);`
  - Here s1 and s2 are copied and formatted together to form an Owned String and we can still use s1 and s2

- **Indexing a string**
  - This is more complicated than most languages as normally we index like an array
  - But since a String is made from UTF-8 bytes it isn't that simple
  - For instance take this as an example
    - `let s1 = String::from("hello");`
    - Here we see that s1 is made up of 5 charachters and since it is using the ASCII values of UTF-8 thats true, each character is one byte
    - But take a look at the following
    - `let s1 = String::from(Здравствуйте)`
    - This is hello in russian, and we can see that this is 12 characters, but it is made up of 24 bytes
    - This is because russian letters require more bytes than the ASCII values to represent them
    - In this case the length of the string is actually 24
    - In short using `s1[0]` does not give the first character but the first byte, and in certain cases that doesn't make sense when the character is made up of more than one byte
  - There are 3 different ways a word can be represented in unicode
    - Let's start with a hindi word: `let hello = String::from("नमस्ते")`
      - Bytes:
        - `[224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141, 224, 164, 164, 224, 165, 135]`
        - There are 18 bytes in the word

    - Scalars:
        - `['न', 'म', 'स', '्', 'त', 'े']`
        - These represent whole characters or parts of a character
        - In rust, the `char` type refers to the Scalar representation
    - Grapheme Clusters:
        - `["न", "म", "स्", "ते"]`
        - These represent whole characters and as we can see, this word uses 4 characters or symbols to make the word "hello"

- In rust we cannot directly index a string because it doesn't know which of the three types of representation we are using and depending on the representation, the size of the string changes

- We can index into a specific representation however

- Byte and Scalar Example:

  ```
  let hello  = String::from("नमस्ते");
  //Bytes [224, 164, 168, 224, 164, 174, 224, 164, 184, 224, 165, 141,
  224, 164, 164, 224, 165, 135]
  for b in hello.bytes(){
      println!("{}",b);
  }
  //Scalar ['न', 'म', 'स', '्', 'त', 'े']
  for c in hello.chars(){
      println!("{}",c);
  }
  ```

    - Note that for scalar, as we mentioned is how `char` is defined, we get a list of `chars`

- Of course the one we want all along is the **Grapheme Clusters**

- However, the ability to iterate over Grapheme clusters is not there by default, and this is to keep the Rust library cleaner

    - To do this we need to import a crate in `cargo.toml`

      ```
      [package]
      name = "tutorial14_vector_string_hashmap"
      version = "0.1.0"
      edition = "2021"

      # See more keys and their definitions at https://doc.rust-
      lang.org/cargo/reference/manifest.html

      [dependencies]
      unicode-segmentation = "1.7.1"
      ```

    - Bring it into scope

        - `use unicode_segmentation::UnicodeSegmentation;`
    - Grapheme Example:

    ```
    for g in "नमस्ते".graphemes(true){
      println!("{}", g);
    }
    ```

        - The `true` is for extended grapheme cluster, safer to put it there if needed
    - Remember you cannot do this on a company network

Hashmap

- Hashmaps are a collection that uses key value pairs.
- Those keys and values can be of any type
- There is a hashing function to determine how to place the value for each key in memory
- To start, you need to import the Hashmap library from the standard library
    - `use std::collections::HashMap;`
- Lets use a hashmap to keep score of two differnt teams
    - `let blue = String::from("blue");`
    - `let yellow = String::from("yellow");`
- To define a hashmap we can use the `new` method
- And to add into our HashMap we use the `insert` method
- Example:

```
let blue = String::from("blue");
let yellow = String::from("yellow");
let mut scores = HashMap::new();
scores.insert(blue, 10);
scores.insert(yellow, 50);
```

    - Note that if we didn't insert one key and value pair in the HashMap, we would have an error
    - Rust needs to know the type for the key and value, and inserting will allow Rust to determine that
    - Here each team is a key and their score is a value
    - Also note that we are not passing in a reference of `blue` and `yellow` which means that `scores` now owns both of those `Owned String`
- To get a value from a HashMap, you just need the key and the `get` method

- Example:

  ```
  let team_name = String::from("blue");
  let score: Option<&i32> = scores.get(&team_name);
  let score: Option<&i32> = scores.get("yellow");
  let mut val = 0;
  match score{
        Some(sc) => {
            val = *sc;
        }
        None => {
            println!("Key Value pair doesn't exist.")
        }
    }
  ```

  - Note here that we need to pass in a reference to an Owned String or we need to use a string slice.
  - Also note that the return type here is an Option Enum, because there is a chance that the value or key doesn't exist. In which case we would return `None`
  - We need to use a match to extract the value

- We also can iterate over the keys and values as a tuple `(key, value)`

  - Example:

    ```
    for (key, value) in &scores{
        println!("{}, {}", key, value);
    }
    ```

    - Each entry is a reference of an owned string and i32.
    - Also note that we are iterating over a reference of scores, we don't want to lose ownership to the HashMap

- Updating a Hashmap uses the `entry` method

  - This method expects an existing key/value pair
  - You can use the `or_insert()` method to make a complete method call
  - Note that when you use `insert()` method, you are overwriting the an existing key/value pair **or** putting in a new one
  - When you use the `entry()` method, you are don't overwrite the existing key. If the key exists you do nothing. If it doesn't exist then you can insert using `or_insert()` method
  - `scores.entry(String::from("yellow")).or_insert(30);`
    - Here when we query for `yellow`, entry returns an **Entry Enum** that returns the value of a given key
    - If the value doesn't exist then we insert 30 and the key `yellow`
    - Since yellow does exist, we don't do anything

        - Another thing to note is that or_insert() returns a `mutable` **reference** to the **value** of the entry

- Let's see how this works in an example:

```
scores.entry(String::from("yellow")).or_insert(30);

//Updating example
let mut map = HashMap::new();
let text  = "hello world wonderful world";

for word in text.split_whitespace(){
    let counter = map.entry(String::from(word)).or_insert(0);
    *counter += 1;
}
println!("{:?}", map);
```

    - Here we are breaking down the text and making a map where each **unique** word is a key and the count of the word is the value
    - We see that we are using `map.entry(String::from()).or_insert()` to make an insert where we are uncertain whether that word has been made into a key already
    - In other words we don't want to overwrite the word each time we see it because we need the previous value of count the word had
    - Using the or_insert method we set a default value to 0 if the word is not there already
        - However, whether a word is there or not or_insert() returns a mutable reference to the value
        - This allows us to easily update the counter for that particular entry/key