

CoE 164

Computing Platforms

03a: Rust Collections





DATA STRUCTURES

Aside from the built-in primitive types, Rust also has *collection* data types, which are variable-sized structures that can have multiple values.

- Vec
- String
- HashMap





VECTORS

A **vector (Vec)** is a collection data type that is the same as an array but it can have a variable number of values. All values should have the same data type.

Vectors can be initialized by using the Vec::new() constructor or the vec! macro.

```
let upd_grades: Vec <f32> = Vec::new();
let upd_scale = vec![1.0, 2.0, 3.0, 4.0, 5.0];
```



VECTORS: VALUE ACCESS

Elements inside vectors can be accessed using bracket indexing or the get () method. Indices start at 0.

Note that bracket indexing works by borrowing, so an & is needed before the indexing operation. In addition, get() returns an Option enum.

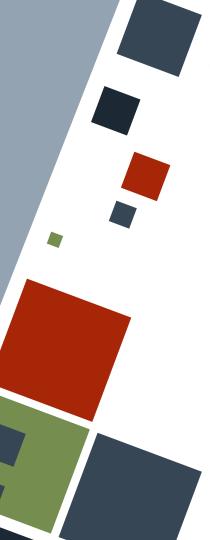
```
Example
let upd scale = vec![
    1.0,
    2.0,
    4.0,
if let Some(v) = upd scale.get(4) {
    println!("{v}!");
else
    println!("Unknown");
```



VECTORS: MUTABILITY

Do not forget the mut keyword if we want the vector to be editable (e.g. resizable and appendable)!

```
let mut upd_grades: Vec <f32> = Vec::new();
let mut upd_scale = vec![1.0, 2.0, 3.0, 4.0, 5.0];
```

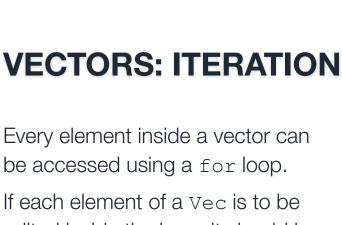


VECTORS: MUTABILITY

Elements can be inserted or removed in a vector. The removed element is usually returned for reference.

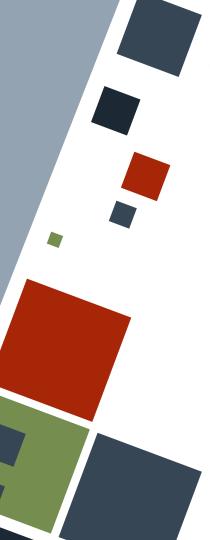
In addition, vectors can also emulate deques.

```
let mut nums = vec![1, 2, 3];
nums.push(4);
nums.push(5);
let v = nums.pop().unwrap(); // 5
nums.insert(0, 11);
let x = nums.remove(2); // 2
```



If each element of a Vec is to be edited inside the loop, it should be 1) mutable, and 2) a &mut keyword should be added in the for loop. Note that the * (deref) operator beside the index is required to access the *value*.

```
let mut upd scale = vec![
    1.0,
    2.0,
    3.0,
    4.0,
    5.0
for i in &mut upd scale {
    *i += 0.5;
for i in &upd scale {
```



VECTORS: MULTIPLE DATA "HACK"

Vectors can "hold" multiple elements by letting its values be an enum.

Using a match or if let can handle the different variants of an enum.

```
enum UserType {
    Admin (bool, u16),
    User { chown: u16 },
let accts = vec![
    UserType::Admin(true, 00777),
    UserType::User { chown: 00444 },
    UserType::Unknown
```



VECTORS: OWNERSHIP

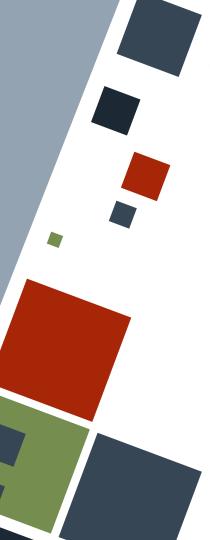
Vector elements follow the standard ownership rules. If they are accessed and saved in variables the Vec should not be edited after and in the same scope.

```
let mut up_grades = vec![1.0, 2.0, 3.0, 4.0];

let first_elm = &up_grades[0];

// Compile error

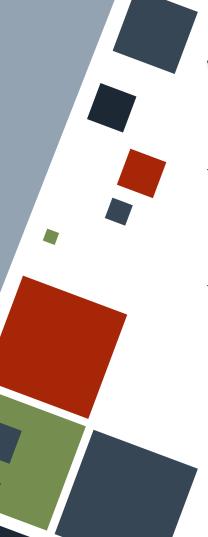
up_grades.push(5.0);
println!("First element: {first_elm}");
```



VECTORS: OWNERSHIP

Inserted elements are moved to the vector. Hence, the element cannot be used anymore after.

```
Example
let mut my strs = vec![
    "hello".to string(),
    "world".to string(),
let c = "!!".to string();
my strs.push(c);
let d = my strs.pop().unwrap();
```



VECTORS: OWNERSHIP

Looping through each element also follows ownership rules. Elements may not be accessible anymore if a reference to the vector is not requested and each element has the Move trait.

```
let mut upd scale = vec![
    String::from("1.0"),
    String::from("2.0"),
];
for i in upd scale {
```

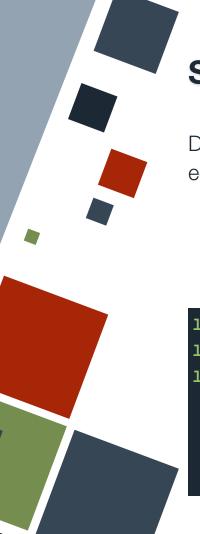


STRINGS

A **string (String)** is a collection data type that is the same as a Vec <u8>. It holds a UTF-8 representation of a human-readable sequence of glyphs.

Strings can be initialized by using the String::new() constructor or converted from an string literal using the String::new() constructor or the to_string() method.

```
let blank_str = String::new();
let hello_v1 = String::from("Hello!");
let hello_v2 = "Hello world!".to_string();
```



STRINGS: MUTABILITY

Do not forget the mut keyword if we want the string to be editable (e.g. editable)!

```
let mut blank_str = String::new();
let mut hello_v1 = String::from("Hello!");
let mut hello_v2 = "Hello world!".to_string();
```



STRINGS: MULTILINGUAL

Because strings are encoded in UTF-8, we can set a variable to any string from any language all over the world without problems.

```
let hello_v1 = String::from("Hello!");
let hello_v2 = "こんにちは".to_string();
let hello_v3 = "안녕하세요".to_string();
```



STRINGS: UPDATE

We can append literals at the end of a string using the push () (character) or push str() (string) methods.

We can concatenate a string with a literal using the + operator.

```
let mut hello_str = String::from("Hell");
hello_str = hello_str + 'o';
hello_str.push(' ');
hello_str.push_str("world!");

// "Hello world!"
println!("{hello_str}");
```



STRINGS: UPDATE OWNERSHIP

When using the + operator to join two strings from two variables, the left operand is moved while the second operand should be borrowed.

```
let mut hello_str = String::from("Hello");
let world_str = " world!";
let hello_str_v2 = hello_str + &world_str;
println!("{hello_str}"); // Compile error!
```



STRINGS: UPDATE MULTIPLE

Multiple calls of the + operator can be reduced to a more readable call using the format! macro. This is the same as the print!/println! macro except that it returns a String.

Unlike the + operator, all variables in the format! macro are implicitly borrowed.

```
let mut hello_v0 = String::from("Hello");
let hello_v1 = hello_v0 + " w" + "or" + "ld";
let hello_v2 = format!("{hello_v1}!!!");
println!("{hello_v2}"); // "Hello world!!!"
```



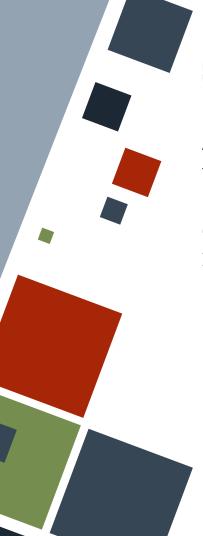
STRINGS: INDEXING

Due to the nature of UTF-8, a glyph can be represented between one to four bytes. Normal indexing will not compile.

For example, an ASCII character needs one byte, but a Japanese hiragana glyph needs three bytes.

```
let hello_en = String::from("Hello");
let hello_jp = String::from("こんにちは");

println!("1st: {}", &hello_en[0]); // 'H'
println!("1st: {}", &hello_jp[0]); // compile error
```



STRINGS: INDEXING

A way to process each character is to iterate through them using a for loop. The string should either be converted as chars() or as bytes().

```
let hello jp = String::from("
    こんにちは
");
for v in hello jp.bytes() {
    println!("{v}");
for v in hello jp.chars() {
    println!("{v}");
```



SLICES

A **slice** is a contiguous sequence of elements in a collection. Range notation is used to get a slice.

Note that a slice is a reference (&str), and hence, is borrowed. String literals are slices!

"Indexing" can be done by *slicing* a portion of the string to get a glyph.

```
let hello_jp = String::from("こんにちは");

let kon = &hello_jp[..6]; // こん
let nichi = &hello_jp[6..12]; // にち
let wa = &hello_jp[12..];
```



SLICES: COLLECTIONS

A slice stores the starting element and the length.

In general, any collection can be sliced. Arrays and Vecs can be sliced, which will be annotated with data type & [dtype].

A slice can also be mutable. Indexing from 0 will be relative to the slice itself.

```
let mut vec ints = vec![3, 4, 5, 6, 7];
let first = &vec ints[0];
let after first = &vec ints[1..]; // type &[i64]
let mut after second = &mut vec ints[2..]; // &mut [i64]
after second[0] = 8;
```



HASHMAP

A **hashmap** (**HashMap**) is a collection data structure that stores values that can be indexed with a *key*. This key is unique across the whole hashmap and is not limited to integers. Do not forget the mut keyword if we want the hashmap to be editable!

The HashMap module needs to be used to use hashmaps in your code.



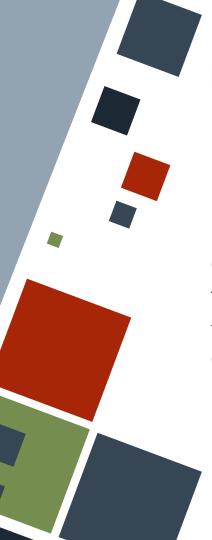
HASHMAP: VALUE ACCESS

Similar to vectors, bracket indexing or the get () method can be used to access the value associated with a key.

The key to look up should be provided as a reference.

```
// -1
let v = scores.get("Average").copied().or_unwrap(-1);

// 5.0
let v_v2 = scores["Poor"];
```

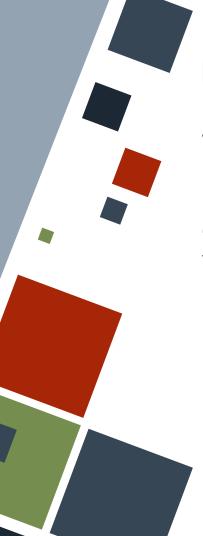


HASHMAP: ITERATION

Every key-value pair (an *entry*) inside a hashmap can be accessed using a for loop. Note that the loop accesses entries in arbitrary order.

There is also an option to iterate through the keys only or the values only.

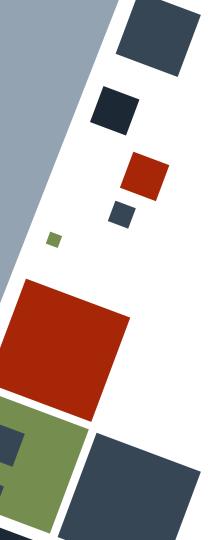
```
for (k, v) in &scores {
    println!("{k}: {v}");
for &v in scores.values() {
    println!("Grade {v}");
```



HASHMAP: MUTABILITY

We can insert and remove key-value pairs in a hashmap. If a key exists, its value will be overwritten with the newly-inserted value.

```
scores.insert(
    "Average".to string(),
    3.0,
scores.insert(
    "Excellent".to string(),
    1.5,
let a = scores.remove(
    "Excellent".to string()
 .unwrap();
```



HASHMAP: UPDATE

We can get an Entry enum of a hashmap to update its values under specific cases. One is to insert a value only if the key does not exist.

```
Example
scores.entry(
    "Incomplete".to string()
).or default();
scores.entry(
    "Very Good".to string()
).or insert(1.5);
scores.entry(
    "Average".to string()
).or insert(3.0);
```



HASHMAP: OWNERSHIP

Keys and values are moved when inserted in a hashmap. Hence, they are not available for use after.

```
let my_str = "Pass".to_string();
let my_val = 3.0;
scores.insert(my_str, my_val);

// Compile error
println!("{my_str}");
```

RESOURCES

The Rust Book





CoE 164

Computing Platforms

03a: Rust Collections



