# Rust Collections

CPSC 310 - Programming Languages

# `String` Representation

- Rust's `String` is a 3-tuple
  - A pointer to a byte array (interpreted as UTF-8); dropped when the owner is
  - A (current) length
  - A (maximum) capacity
- Example

```
let mut s = String::new();
println!("{}", s.capacity());
for _ in 0..5 {
    s.push_str("hello");
    println!("{}, {}", str.len, s.capacity());
}
```

# UTF-8 and Rust `String`

- UTF-8 is a variable length character encoding
  - The first 128 characters need one byte
  - The next 1,920 characters need two bytes, and so on, up to 4 bytes
- Because of the UTF-8 encoding, you cannot index a string directly

```
let s1 = String::from("hello");
let h = s1[0]; // fails
```

# Slices: Motivation

- Suppose we want the first word of a string
- OCaml example

```
let first_word s =
  try
    let i = String.index s ' ' in
    String.sub s 0 i
  with Not_found -> s
```

- `String.sub` allocates new memory and copies the sub-string's contents; this can be wasteful if both `s` and its substring are to be treated as immutable

# Slice: Shared Data, Separate Metadata

- We want to have both strings share the same underlying data
- Containers in Rust permit a way to reference a portion of an object's contents; this is called a slice
- If `s` is a `String`, then `&s[<range>]` is a string slice, where `<range>` can be:
  - `i..j` – range from `i` to `j` inclusive
  - `i..` – range from `i` to the current length
  - `..j` – range from 0 to `j`
  - `..` – range from 0 to the current length
- `&str` is the type of a `String` slice

# String Slice Example

- Example: `first_word`

```
pub fn first_word (s: &String) -> &str {
    for (i, item) in s.char_indices() {
        if item == ' ' {
            return &s[0..i];
        }
    }
    s.as_str()
}
```

- If we used `s.as_bytes()` we could end up examining one byte of a multi-byte character due to UTF-8 encoding

# String Slices and Ownership

- A `&str` borrows from the original string; this prevents dangling pointers

```
let mut s = String::from("Hello, world!");
let word = first_word(&s); // borrow
s.clear(); // error; cannot take mut ref
```

# String Slices are the Default

- String literals are slices

  ```
  let s: &str = "Hello, world!";
  ```

  - Variable `s` is not the owner of the string data
  - `String` does own its data; useful if you want to modify it

- Should use slices where possible

# Vectors

- The `Vec<T>` type is a dynamically sized array of type `T`

```
{  let mut v: Vec<i32> = Vec::new();
   v.push(1);
   v.push("hello"); // type error
   let w  = vec![1, 2, 3]; // vec! is a macro
} // v, w and their elements dropped
```

- The index operation can fail (panic) or return an `Option`

```
let v = vec![1, 2, 3, 4, 5];
let third: &i32 = &v[2]; // panics if out-of-bounds
let third: Option<&i32> = v.get(2); // None if out-of-b
```

# Aside: `Option`

- The `Option<T>` type is an enumerated type, like an OCaml variant, with possible values `Some(v)` and `None`

```
let v = vec![1, 2, 3, 4, 5];
let third: Option<&i32> = v.get(2);
let z =
  match third {
    Some(i) => Some(i+1),
    None => None
  }
```

# Vectors: Updates and Iteration

- Example: update

```
let mut a = vec![1, 2, 3, 4, 5];
{  let p = &mut a[1]; // mutable borrow
   *p = 2; // updates a[1]
} // ownership restored
println!("vector contains {:?}", &a);
```

- Example: iteration; iterator variable can be mutable or immutable

```
let mut v = vec![1, 2, 3, 4, 5];
for i in &v { println!("{}", i); }
for i in &mut v { *i += 10; }
```

# Vector and Strings

- Similar to `Strings`, vectors can have slices

```
let a = vec![1, 2, 3, 4, 5];
let b = &a[1..3]; // [2,3]
let c = &b[1]; // 3
println!("{}", c); // prints 3
```

- `Strings` are implemented internally as a `Vec<u8>`

# HashMap

- A `HashMap<K,V>` is a dictionary data structure with the expected methods
  - `new: () -> HashMap<K,V`
  - `insert: (K, V) -> Option<V>`
  - ‘get: (&K) -> Option<&V>
  - ...