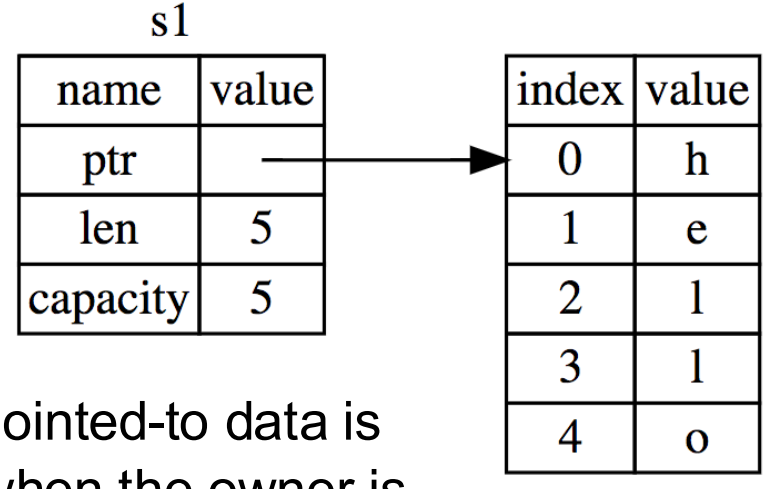
CMSC 330: Organization of Programming Languages

Strings, Slices, Vectors, HashMaps in Rust

`String` Representation

- Rust's **`String`** is a 3-tuple
  - A pointer to a byte array (interpreted as UTF-8)
  - A (current) length
  - A (maximum) capacity

Always: length ≤ capacity



`String` pointed-to data is dropped when the owner is

`String` Representation

- Rust's **`String`** is a 3-tuple
  - A pointer to a byte array (interpreted as UTF-8)
  - A (current) length
  - A (maximum) capacity
    - Always: length ≤ capacity

Code

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

Prints

```
0
5,5
10,10
15,20
20,20
25,40
```

UTF-8 and Rust Strings

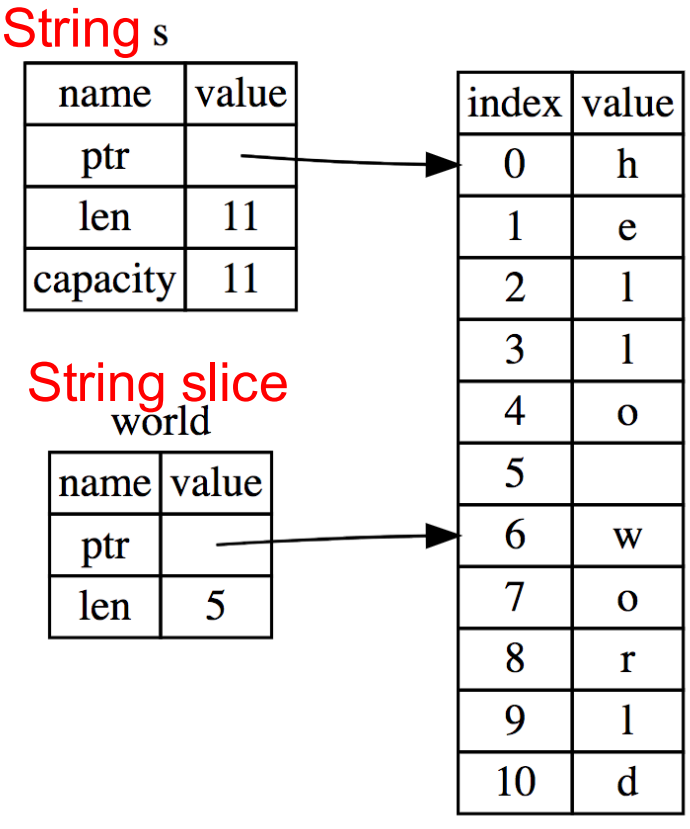
- UTF-8 is a variable length character encoding
  - The first 128 characters (US-ASCII) need one byte
  - The next 1,920 characters need two bytes, which covers the remainder of almost all Latin-script alphabets, … up to 4 bytes

- You may not index a string directly; Rust stops you
  - You could end up in the middle of a character!

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

Slice: Shared Data, Separate Metadata

- What we want is to have both strings share the same underlying data
- Happily, Rust's containers permit a way to reference a portion of an object's contents
  - These are called slices



String Slices in Rust

- If **`s`** is a **`String`**, then **`&s[`***range***`]`** is a string slice, where *range* can be as follows
  - *i*`..`*j* is the range from *i* to *j*, inclusive
  - *i*`..` is the range from *i* to the current length
  - `..`j is the range from `0` to *j*
  - `..` is the range from `0` to the current length

- **`&str`** is the type of a **`String`** slice

String Slice Example

- Here's **`first_word`** in Rust, using slices:

```
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 the UTF-8 encoding

String Slices and Ownership

- A **`&str`** slice borrows from the original string
  - Just like an immutable **`String`** reference
  - This prevents dangling pointers

```
let mut s = String::from("hello world");
let word = first_word(&s); //borrow
s.clear(); // Error! Can't take mut ref
```

- Recall borrowing rules:
  - Multiple immutable refs, or
  - Only one mutable ref (no immut ones)

```
let b = &s[..];
let c = &s[..];
print!("{}{}", b, c);
```

```
let b = &mut s[..];
let c = &mut s[..]; //error
print!("{}{}", b, c);
```

Quiz 1: What is the output?

```
let s = String::from("Rust is fun!");
let h = &s[0..4];
println!("{}",h);
```

A. Rust
B. is
C. fun!
D. Type Error

Quiz 1: What is the output?

```
let s = String::from("Rust is fun!");
let h = &s[0..4];
println!("{}",h);
```

A. Rust
B. is
C. fun!
D. Type Error

String Slices are (should be) the Default

- String literals are slices

```
let s:&str = "hello world";
```

  - Variable **`s`** is *not* the owner of this string data
    - the compiler establishes a static owner to permit free immutable sharing
  - **`String`**s *do* own their data; useful if you want to modify it

- Should use slices where possible
  - E.g., earlier example: **`fn first_word(s:&str) -> &str`**
    - Can convert **`String s`** to a slice via **`&s[..]`**. Oftentimes, this coercion is done automatically (due to **`Deref`** trait)

Quiz 2: What is the output?

```
let mut s1 = String::from("Hello");
let s2 = " World";
s1.push_str(s2);
print!("{}",s2);
```

A. World
B. Hello World
C. Error because s2 transferred the ownership

Quiz 2: What is the output?

```
let mut s1 = String::from("Hello");
let s2 = " World";
s1.push_str(s2);
print!("{}",s2);
```

A. World. **push_str()** function does not take the ownership of the parameter
B. Hello World
C. Error because s2 transferred the ownership

Quiz 3: What is the output?

```
let s1 = String::from(”CMSC");
let s3; //deferred init
{
    let s2 = String::from(”330");
    s3 = s1+&s2;
}
print!("{}",s3);
print!("{}",s1);
```

A. CMSC330
B. CMSC
C. CMSC330CMSC
D. Error.

Quiz 3: What is the output?

```
let s1 = String::from(”CMSC");
let s3; //deferred init
{
    let s2 = String::from(”330");
    s3 = s1+&s2;
}
print!("{}",s3);
print!("{}",s1);
```

A. CMSC330
B. CMSC
C. CMSC330CMSC
D. Error. s1 lost ownership

Vectors: Basics

- **`Vec<T>`** in Rust is **`Arraylist<T>`** in Java

```
{ let mut v:Vec<i32> = Vec::new();
  v.push(1); // adds 1 to v
  v.push("hi"); //error - v contains i32s
  let w = vec![1, 2, 3]; //vec! is a macro
} // v,w and their elements dropped
```

- Indexing can fail (panic) or return an **`Option`**

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

https://doc.rust-lang.org/book/second-edition/ch08-01-vectors.html

Aside: Options

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

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

- We'll see more about enumerated types later
  - For now, follow your nose

Vectors: Updates and Iteration

```
let mut a = vec![10, 20, 30, 40, 50];
{ let p = &mut a[1]; //mutable borrow
  *p = 2; //updates a[1]
}//ownership restored
println!("vector contains {:?}",&a);
```

- If we remove the `{}` block around the def of `p`, above, then the code fails
  - Not allowed to print via `a` while mutable borrow `p` is out
- Iterator variable can be mutable or immutable:

```
let mut v = vec![100, 32, 57];
for i in &v { println!("{}", i); }
for i in &mut v { *i += 50; }
```

Vector and Strings

- Like **`String`**s, vectors can have slices

```
let a = vec![10, 20, 30, 40, 50];
let b = &a[1..3]; //[20,30]
let c = &b[1];    //30
println!("{}",c); //prints 30
```

- **`String`**s implemented internally as a **`Vec<u8>`**
  - But: don't mess with the byte-level representation of UTF-8 strings.

HashMaps

- `HashMap<K,V>` has the expected methods (roughly – see manual for gory details)
  - `new : () -> HashMap<K,V>`
  - `insert : (K,V) -> Option<V>`
  - `get : (&K) -> Option<&V>`
- See also
  - `get_mut`, `entry`, and `or_insert`

https://doc.rust-lang.org/book/second-edition/ch08-03-hash-maps.html

https://doc.rust-lang.org/std/collections/struct.HashMap.html

Quiz 4: What is the output?

```
use std::collections::HashMap;
fn main(){
    let mut h = HashMap::new();
    h.insert("Alice", "1");
    h.insert("Bob", "2");
    match h.get(&”Alice") {
        Some(&id) => println!(”Alice:{}",id),
        _ => println!("Not Found"),
    }
}
```

A. Alice:1
B. Not Found
C. Error

Quiz 4: What is the output?

```
use std::collections::HashMap;
fn main(){
    let mut h = HashMap::new();
    h.insert("Alice", "1");
    h.insert("Bob", "2");
    match h.get(&”Alice") {
        Some(&id) => println!(”Alice:{}",id),
        _ => println!("Not Found"),
    }
}
```

A. Alice:1
B. Not Found
C. Error