CMSC 330: Organization of Programming Languages

Strings, Slices, Vectors, HashMaps
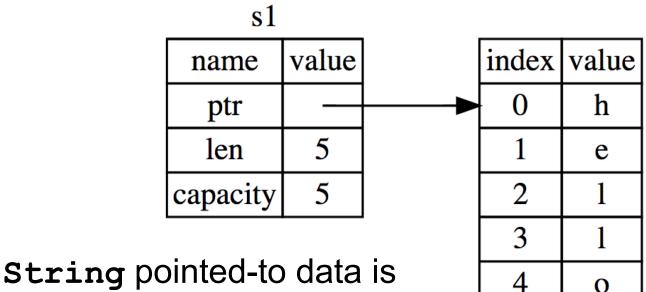in Rust

CMSC330 Summer 2018

`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

s1

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

`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	Prints

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

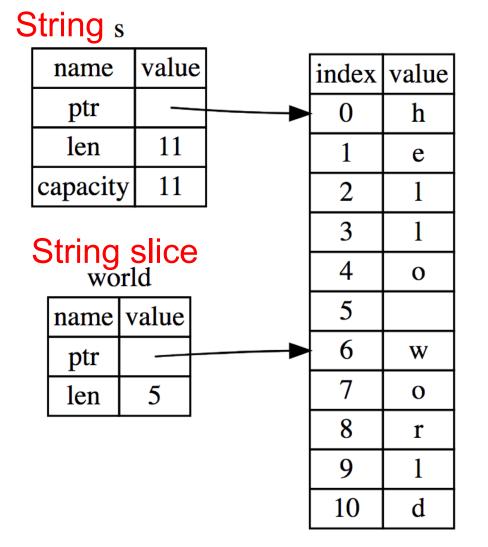| Prints |
|---|
| 0 |
| 5,5 |
| 10,10 |
| 15,20 |
| 20,20 |
| 25,40 |

Slices: Motivation

- Suppose we want the first word of a string. Here's how we might do it in OCaml

```
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 is a waste (especially with a large string) if both **s** and its substring are to be treated as immutable

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 present a slice of an object's contents

String s

name	value
ptr	
len	11
capacity	11

String slice

world

name	value
ptr	
len	5

index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

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:

```rust
fn first_word(s: &String) -> &str {
  let bytes = s.as_bytes();
  for (i, &item) in
       bytes.iter().enumerate() {
    if item == b' ' {
      return &s[0..i];
    }
  }
  &s[..]
}
```

Using String Slices

- 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
```

- String literals are slices

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

- Should use slices where possible
  - E.g., **`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)

Strings Miscellany

- **push_str(&mut self, string: &str)**
  - **string** argument is a slice, so doesn't take ownership, while **self** is a mutable reference, implying it is the only such reference

- Iteration over chars, bytes, etc.

Code   Prints

```
let s = String::from("hello");
for (i,c) in s.char_indices() {
 println!("{},{}",i,c);
}
```

0,h

1,e

2,l

3,l

4,o

- See also **split_at_whitespace**

https://doc.rust-lang.org/std/string/struct.String.html

9

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];
} // 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 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>**

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