DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 19

1. HEAP: DANGERS OF MANUAL MEMORY MANAGEMENT

2. OWNERSHIP AND BORROWING IN RUST

3. METHODS IN RUST

LAST TIME

- Possible data locations: stack and heap
- Mostly focused on the stack

HEAP MANAGEMENT

Memory allocation:

- ask for a given amount of space
- receives a pointer to it
  (or an out of memory error)

Freeing memory:

- classical manual: explicitly return it
  - more complicated
- automatic: garbage collection
  - comes with additional costs

C: `malloc` / `free`

C++: `new` / `delete` + C

HEAP MANAGEMENT

Memory allocation:

- ask for a given amount of space
- receives a pointer to it
  (or an out of memory error)

Freeing memory:

- classical manual: explicitly return it
  - more complicated
- automatic: garbage collection
  - comes with additional costs

C: `malloc` / `free`

C++: `new` / `delete` + C

Pitfalls of manual memory management:

- leaks: unused memory never returned
- attempting to use a pointer to memory that was deallocated
- returning memory that was already deallocated

HEAP MANAGEMENT

Memory allocation:

- ask for a given amount of space
- receives a pointer to it
  (or an out of memory error)

Freeing memory:

- classical manual: explicitly return it
  - more complicated
- automatic: garbage collection
  - comes with additional costs

C: `malloc` / `free`

C++: `new` / `delete` + C

Pitfalls of manual memory management:

- leaks: unused memory never returned
- attempting to use a pointer to memory that was deallocated
- returning memory that was already deallocated

How does Rust deal with these problems?

ALLOCATING ON THE HEAP IN RUST

- Various methods. The simplest via `Box::new(...)`

In [2]:
```
// placing integers on the heap
let mut pointer = Box::new(2000);
let pointer2 : Box<i32> = Box::new(22);
```

ALLOCATING ON THE HEAP IN RUST

- Various methods. The simplest via `Box::new(...)`

In [2]:
```
// placing integers on the heap
let mut pointer = Box::new(2000);
let pointer2 : Box<i32> = Box::new(22);
```

In [3]:
```
// accessing data via a * operator
println!("sum: {}", *pointer + *pointer2);
```

```
sum: 2022
```

ALLOCATING ON THE HEAP IN RUST

- Various methods. The simplest via `Box::new(...)`

In [2]:
```
// placing integers on the heap
let mut pointer = Box::new(2000);
let pointer2 : Box<i32> = Box::new(22);
```

In [3]:
```
// accessing data via a * operator
println!("sum: {}", *pointer + *pointer2);
```

```
sum: 2022
```

In [4]:
```
*pointer = 3000;
println!("sum: {}", *pointer + *pointer2);
```

```
sum: 3022
```

EXPERIMENT WITH PASSING THE POINTER AROUND

In [5]:
```
fn print_content(pointer:Box<i32>) {
    println!("content: {}", *pointer)
}

let p = Box::new(123);

print_content(p);
```

```
content: 123
```

EXPERIMENT WITH PASSING THE POINTER AROUND

In [5]:
```
fn print_content(pointer:Box<i32>) {
    println!("content: {}", *pointer)
}

let p = Box::new(123);

print_content(p);
```

```
content: 123
```

In [6]:
```
let q = Box::new(321);

print_content(q);
print_content(q);
```

```
print_content(q);
              ^ value moved here
print_content(q);
              ^ value used here after move
let q = Box::new(321);
    ^ move occurs because `q` has type `Box<i32>`, whic
h does not implement the `Copy` trait
use of moved value: `q`
```

WHAT HAPPENED: OWNERSHIP

- Each value in Rust has a variable that is its **owner**
- Only **one** owner
- When the owner goes out of scope, the value is dropped

WHAT HAPPENED: OWNERSHIP

- Each value in Rust has a variable that is its **owner**
- Only **one** owner
- When the owner goes out of scope, the value is dropped

In [7]:
```
fn print_content(pointer:Box<i32>) {
    println!("content: {}", *pointer)
}

let q = Box::new(321);

print_content(q);
print_content(q);
```

```
print_content(q);
              ^ value moved here
print_content(q);
              ^ value used here after move
let q = Box::new(321);
    ^ move occurs because `q` has type `Box<i32>`, whic
h does not implement the `Copy` trait
use of moved value: `q`
```

- First call to `print_content`: `Box::new(321)` is **moved** from `q` to `pointer`
- (if it compiled) at the end of `print_content`:
  - `Box::new(321)` would be dropped
  - its space on the heap deallocated

WHAT HAPPENED: OWNERSHIP

- Each value in Rust has a variable that is its **owner**
- Only **one** owner
- When the owner goes out of scope, the value is dropped

```
In [7]: fn print_content(pointer:Box<i32>) {
            println!("content: {}", *pointer)
        }

        let q = Box::new(321);

        print_content(q);
        print_content(q);
```

```
print_content(q);
              ^ value moved here
print_content(q);
              ^ value used here after move
let q = Box::new(321);
    ^ move occurs because `q` has type `Box<i32>`, whic
h does not implement the `Copy` trait
use of moved value: `q`
```

- First call to `print_content`: `Box::new(321)` is **moved** from `q` to `pointer`
- (if it compiled) at the end of `print_content`:
  - `Box::new(321)` would be dropped
  - its space on the heap deallocated

Second call can't proceed: **the content of `q` is gone**

MORE EXAMPLES OF OWNERSHIP

In [8]:
```
// won't work, value moved as well
let x = Box::new(123);
println!("x = {}",*x);
let y = x;
println!("x = {}",*x);
```

```
let y = x;
        ^ value moved here
println!("x = {}",*x);
                  ^^ value borrowed here after move
let x = Box::new(123);
    ^ move occurs because `x` has type `Box<i32>`, whic
h does not implement the `Copy` trait
borrow of moved value: `x`
```

MORE EXAMPLES OF OWNERSHIP

In [8]:
```
// won't work, value moved as well
let x = Box::new(123);
println!("x = {}",*x);
let y = x;
println!("x = {}",*x);
```

```
let y = x;
        ^ value moved here
println!("x = {}",*x);
                  ^^ value borrowed here after move
let x = Box::new(123);
    ^ move occurs because `x` has type `Box<i32>`, whic
h does not implement the `Copy` trait
borrow of moved value: `x`
```

Fix our previous example by returning the pointer

In [9]:
```
fn print_content(pointer:Box<i32>) -> Box<i32> {
    println!("content: {}", *pointer);
    pointer
}

let q = Box::new(321);

let q = print_content(q);
let q = print_content(q);
let q = print_content(q);
```

```
content: 321
content: 321
content: 321
```

AVOIDING MOVING VALUES A LOT: BORROWING

In [10]:

```rust
#[derive(Debug)]
struct Road {
    intersection_1: u32,
    intersection_2: u32,
    max_speed: u32,
}

// adding a function in the namespace of Road
impl Road {
    // very useful constructor
    fn new(i1:u32,i2:u32,speed:u32) -> Road {
        Road {
            intersection_1: i1,
            intersection_2: i2,
            max_speed: speed,
        }
    }
}

let road = Road::new(13,23,25);
println!("{}",road.max_speed);
```

```
25
```

AVOIDING MOVING VALUES A LOT: BORROWING

In [10]:
```
#[derive(Debug)]
struct Road {
    intersection_1: u32,
    intersection_2: u32,
    max_speed: u32,
}

// adding a function in the namespace of Road
impl Road {
    // very useful constructor
    fn new(i1:u32,i2:u32,speed:u32) -> Road {
        Road {
            intersection_1: i1,
            intersection_2: i2,
            max_speed: speed,
        }
    }
}

let road = Road::new(13,23,25);
println!("{}",road.max_speed);
```

```
25
```

In [11]:
```
// checking whether it moves
let another = road;
println!("{}",road.max_speed);
```

```
let another = road;
                ^^^^ value moved here
println!("{}",road.max_speed);
                ^^^^^^^^^^^^^^^ value borrowed here after
move
borrow of moved value: `road`
```

AVOIDING MOVING VALUES A LOT: BORROWING

In [10]:
```
#[derive(Debug)]
struct Road {
    intersection_1: u32,
    intersection_2: u32,
    max_speed: u32,
}

// adding a function in the namespace of Road
impl Road {
    // very useful constructor
    fn new(i1:u32,i2:u32,speed:u32) -> Road {
        Road {
            intersection_1: i1,
            intersection_2: i2,
            max_speed: speed,
        }
    }
}

let road = Road::new(13,23,25);
println!("{}",road.max_speed);
```

```
25
```

In [11]:
```
// checking whether it moves
let another = road;
println!("{}",road.max_speed);
```

```
let another = road;
               ^^^^ value moved here
println!("{}",road.max_speed);
               ^^^^^^^^^^^^^^ value borrowed here after
move
borrow of moved value: `road`
```

In [12]:
```
fn display_1(r:Road) {
    println!("{:?}",r);
}

let road = Road::new(101,102,30);
display_1(road);
// display_1(road);
```

```
Road { intersection_1: 101, intersection_2: 102, max_sp
eed: 30 }
```

AVOIDING MOVING VALUES A LOT: BORROWING

In [10]:
```
#[derive(Debug)]
struct Road {
    intersection_1: u32,
    intersection_2: u32,
    max_speed: u32,
}

// adding a function in the namespace of Road
impl Road {
    // very useful constructor
    fn new(i1:u32,i2:u32,speed:u32) -> Road {
        Road {
            intersection_1: i1,
            intersection_2: i2,
            max_speed: speed,
        }
    }
}

let road = Road::new(13,23,25);
println!("{}",road.max_speed);
```

```
25
```

In [11]:
```
// checking whether it moves
let another = road;
println!("{}",road.max_speed);
```

```
let another = road;
              ^^^^ value moved here
println!("{}",road.max_speed);
              ^^^^^^^^^^^^^^^ value borrowed here after
move
borrow of moved value: `road`
```

In [13]:
```
fn display_1(r:Road) {
    println!("{:?}",r);
}

let road = Road::new(101,102,30);
display_1(road);
display_1(road);
```

```
display_1(road);
          ^^^^ value moved here
display_1(road);
          ^^^^ value used here after move
let road = Road::new(101,102,30);
    ^^^^ move occurs because `road` has type `Road`, wh
ich does not implement the `Copy` trait
use of moved value: `road`
```

AVOIDING MOVING VALUES A LOT: BORROWING

In [10]:
```
#[derive(Debug)]
struct Road {
    intersection_1: u32,
    intersection_2: u32,
    max_speed: u32,
}

// adding a function in the namespace of Road
impl Road {
    // very useful constructor
    fn new(i1:u32,i2:u32,speed:u32) -> Road {
        Road {
            intersection_1: i1,
            intersection_2: i2,
            max_speed: speed,
        }
    }
}

let road = Road::new(13,23,25);
println!("{}",road.max_speed);
```

```
25
```

In [11]:
```
// checking whether it moves
let another = road;
println!("{}",road.max_speed);
```

```
let another = road;
                ^^^^ value moved here
println!("{}",road.max_speed);
                ^^^^^^^^^^^^^^ value borrowed here after
move
borrow of moved value: `road`
```

In [14]:
```
fn display_1(r:Road) {
    println!("{:?}",r);
}

let road = Road::new(101,102,30);
display_1(road);
// display_1(road);
```

```
Road { intersection_1: 101, intersection_2: 102, max_sp
eed: 30 }
```

AVOIDING MOVING VALUES A LOT: BORROWING

Read–only reference:

- Reference type becomes `&Type`
- To create: `&value`
- To access content: `*reference`

AVOIDING MOVING VALUES A LOT: BORROWING

Read–only reference:

- Reference type becomes `&Type`
- To create: `&value`
- To access content: `*reference`

In [15]:
```
fn display_2(r:&Road) {
    println!("{:?}",*r);
}

let road = Road::new(101,102,30);
display_2(&road); // <- have to explicitly create a reference
display_2(&road);
```

```
Road { intersection_1: 101, intersection_2: 102, max_speed: 30 }
Road { intersection_1: 101, intersection_2: 102, max_speed: 30 }
```

AVOIDING MOVING VALUES A LOT: BORROWING

Mutable reference:

- Reference type becomes `&mut Type`
- To create: `&mut value`
- To access content: `*reference`

In [16]:
```
// regular references won't work
fn update_speed(r:&Road, new_speed: u32) {
    // r.max_speed equivalent to (*r).max_speed
    // because Rust is smart
    r.max_speed = new_speed;
}
```

```
    r.max_speed = new_speed;
    ^^^^^^^^^^^^^^^^^^^^^^^ `r` is a `&` reference, so the data it refers to cannot be written
cannot assign to `r.max_speed`, which is behind a `&` reference
help: consider changing this to be a mutable reference

&mut Road
```

AVOIDING MOVING VALUES A LOT: BORROWING

Mutable reference:

- Reference type becomes `&mut Type`
- To create: `&mut value`
- To access content: `*reference`

In [16]:
```
// regular references won't work
fn update_speed(r:&Road, new_speed: u32) {
    // r.max_speed equivalent to (*r).max_speed
    // because Rust is smart
    r.max_speed = new_speed;
}
```

```
    r.max_speed = new_speed;
    ^^^^^^^^^^^^^^^^^^^^^^^ `r` is a `&` reference, so
the data it refers to cannot be written
cannot assign to `r.max_speed`, which is behind a `&` r
eference
help: consider changing this to be a mutable reference

&mut Road
```

In [17]:
```
fn update_speed(r:&mut Road, new_speed: u32) {
    // r.max_speed equivalent to (*r).max_speed
    // because Rust is smart
    r.max_speed = new_speed;
}
```

AVOIDING MOVING VALUES A LOT: BORROWING

Mutable reference:

- Reference type becomes `&mut Type`
- To create: `&mut value`
- To access content: `*reference`

In [16]:
```
// regular references won't work
fn update_speed(r:&Road, new_speed: u32) {
    // r.max_speed equivalent to (*r).max_speed
    // because Rust is smart
    r.max_speed = new_speed;
}
```

```
    r.max_speed = new_speed;
    ^^^^^^^^^^^^^^^^^^^^^^^ `r` is a `&` reference, so
the data it refers to cannot be written
cannot assign to `r.max_speed`, which is behind a `&` r
eference
help: consider changing this to be a mutable reference

&mut Road
```

In [17]:
```
fn update_speed(r:&mut Road, new_speed: u32) {
    // r.max_speed equivalent to (*r).max_speed
    // because Rust is smart
    r.max_speed = new_speed;
}
```

In [18]:
```
let mut road = Road::new(100,200,30);
display_2(&road);
update_speed(&mut road, 25);
display_2(&road);
```

```
Road { intersection_1: 100, intersection_2: 200, max_sp
eed: 30 }
Road { intersection_1: 100, intersection_2: 200, max_sp
eed: 25 }
```

METHODS

- We can add functions that are directly associated with structs and enums!
    - Then we could call them: `road.display()` or `road.update_speed(25)`
- How?
    - Put them in the namespace of the type
    - make `self` the first argument

METHODS

- We can add functions that are directly associated with structs and enums!
    - Then we could call them: `road.display()` or `road.update_speed(25)`
- How?
    - Put them in the namespace of the type
    - make `self` the first argument

```
In [19]: impl Road {

             // note &self: immutable reference
             fn display(&self) {
                 println!("{:?}",*self);
             }
         }
```

METHODS

- We can add functions that are directly associated with structs and enums!
    - Then we could call them: `road.display()` or `road.update_speed(25)`
- How?
    - Put them in the namespace of the type
    - make `self` the first argument

In [19]:
```
impl Road {

    // note &self: immutable reference
    fn display(&self) {
        println!("{:?}",*self);
    }
}
```

In [20]:
```
let mut road = Road::new(1,2,35);
road.display();
(&road).display();
```

```
Road { intersection_1: 1, intersection_2: 2, max_speed:
35 }
Road { intersection_1: 1, intersection_2: 2, max_speed:
35 }
```

METHODS (CONTINUED)

In [21]:
```
impl Road {
    fn update_speed(&mut self, new_speed:u32) {
        self.max_speed = new_speed;
    }
}
```

METHODS (CONTINUED)

In [21]:
```
impl Road {
    fn update_speed(&mut self, new_speed:u32) {
        self.max_speed = new_speed;
    }
}
```

In [22]:
```
road.display();
road.update_speed(25);
road.display();
```

```
Road { intersection_1: 1, intersection_2: 2, max_speed:
35 }
Road { intersection_1: 1, intersection_2: 2, max_speed:
25 }
```

METHODS (CONTINUED)

In [21]:
```
impl Road {
    fn update_speed(&mut self, new_speed:u32) {
        self.max_speed = new_speed;
    }
}
```

In [22]:
```
road.display();
road.update_speed(25);
road.display();
```

```
Road { intersection_1: 1, intersection_2: 2, max_speed:
35 }
Road { intersection_1: 1, intersection_2: 2, max_speed:
25 }
```

In [23]:
```
impl Road {

    fn this_will_move(self) -> Road {
        self
    }
}
```

METHODS (CONTINUED)

In [21]:
```
impl Road {
    fn update_speed(&mut self, new_speed:u32) {
        self.max_speed = new_speed;
    }
}
```

In [22]:
```
road.display();
road.update_speed(25);
road.display();
```

```
Road { intersection_1: 1, intersection_2: 2, max_speed:
35 }
Road { intersection_1: 1, intersection_2: 2, max_speed:
25 }
```

In [23]:
```
impl Road {

    fn this_will_move(self) -> Road {
        self
    }
}
```

In [24]:
```
let r = Road::new(1,2,35);
let r2 = r.this_will_move();
r.display()
```

```
let r2 = r.this_will_move();
         ^^^^^^^^^^^^^^^^ `r` moved due to this metho
d call
let r = Road::new(1,2,35);
    ^ move occurs because `r` has type `Road`, which do
es not implement the `Copy` trait
borrow of moved value: `r`
```

METHODS (SUMMARY)

- Make first parameter `self`
- Various options:
  - `self`: move will occur
  - `&self`: self will be immutable reference
  - `&mut self`: self will be mutable reference

NEXT TIME

Additional topics related to what was covered today:

- Specifying type to be always copied
- Having multiple references at the same time