DS-210: PROGRAMMING FOR DATA SCIENCE

LECTURE 20

1. COPYING INSTEAD OF MOVING
2. MULTIPLE REFERENCES IN PARALLEL
3. GENERICS

LAST TIME: OWNERSHIP AND MOVING

In [2]:

```rust
#[derive(Debug)]
struct BoxSize {
    height: f64,
    width: f64,
    depth: f64,
}

impl BoxSize {
    fn new(height: f64, width: f64, depth: f64)
        -> BoxSize {
        BoxSize{
            height: height,
            width, // = width: width
            depth, // = depth: depth
        }
    }
}
```

LAST TIME: OWNERSHIP AND MOVING

In [2]:

```
#[derive(Debug)]
struct BoxSize {
    height: f64,
    width: f64,
    depth: f64,
}

impl BoxSize {
    fn new(height: f64, width: f64, depth: f64)
        -> BoxSize {
        BoxSize{
            height: height,
            width, // = width: width
            depth, // = depth: depth
        }
    }
}
```

In [3]:

```
let xl_box = BoxSize::new(24.0,18.0,24.0);
println!("{:?}", xl_box);
let move_it_here = xl_box;
//println!("{:?}", xl_box);
```

```
BoxSize { height: 24.0, width: 18.0, depth: 24.0 }
```

LAST TIME: OWNERSHIP AND MOVING

In [2]:
```
#[derive(Debug)]
struct BoxSize {
    height: f64,
    width: f64,
    depth: f64,
}

impl BoxSize {
    fn new(height: f64, width: f64, depth: f64)
        -> BoxSize {
        BoxSize{
            height: height,
            width, // = width: width
            depth, // = depth: depth
        }
    }
}
```

In [4]:
```
let xl_box = BoxSize::new(24.0,18.0,24.0);
println!("{:?}", xl_box);
let move_it_here = xl_box;
println!("{:?}", xl_box);
```

```
let move_it_here = xl_box;
                   ^^^^^^ value moved here
println!("{:?}", xl_box);
                 ^^^^^^ value borrowed here after move
let xl_box = BoxSize::new(24.0,18.0,24.0);
    ^^^^^^ move occurs because `xl_box` has type `BoxSi
ze`, which does not implement the `Copy` trait
borrow of moved value: `xl_box`
```

CLONING

How to make a copy of data?

CLONING

How to make a copy of data?

Option 1: Implement yourself

In [5]:
```
impl BoxSize {
    fn give_me_a_copy(&self) -> BoxSize {
        let BoxSize{height,width,depth} = *self;
        BoxSize{height,width,depth}
    }
}
```

CLONING

How to make a copy of data?

Option 1: Implement yourself

In [5]:
```
impl BoxSize {
    fn give_me_a_copy(&self) -> BoxSize {
        let BoxSize{height,width,depth} = *self;
        BoxSize{height,width,depth}
    }
}
```

In [6]:
```
let box_1 = BoxSize::new(1.1,2.2,3.3);
println!("{:?}",box_1);
let box_2 = box_1.give_me_a_copy();
println!("================");
println!("{:?}",box_1);
println!("{:?}",box_2);
```

```
BoxSize { height: 1.1, width: 2.2, depth: 3.3 }
================
BoxSize { height: 1.1, width: 2.2, depth: 3.3 }
BoxSize { height: 1.1, width: 2.2, depth: 3.3 }
```

CLONING

Option 2: Default cloning (with some extra benefits)

- Use `#[derive(Clone)]` in the definition
- Use method `.clone()` to clone an object

In [7]:
```
#[derive(Clone,Debug)]
struct CloneablePoint {
    x: f64,
    y: f64,
}
```

CLONING

Option 2: Default cloning (with some extra benefits)

- Use `#[derive(Clone)]` in the definition
- Use method `.clone()` to clone an object

In [7]:
```
#[derive(Clone,Debug)]
struct CloneablePoint {
    x: f64,
    y: f64,
}
```

In [8]:
```
let point_1 = CloneablePoint{x:2.2,y:-1.4};
let point_2 = point_1.clone();
println!("{:?}\n{:?}",point_1,point_2);
```

```
CloneablePoint { x: 2.2, y: -1.4 }
CloneablePoint { x: 2.2, y: -1.4 }
```

CLONING

Option 2: Default cloning (with some extra benefits)

- Use `#[derive(Clone)]` in the definition
- Use method `.clone()` to clone an object

In [7]:
```
#[derive(Clone,Debug)]
struct CloneablePoint {
    x: f64,
    y: f64,
}
```

In [8]:
```
let point_1 = CloneablePoint{x:2.2,y:-1.4};
let point_2 = point_1.clone();
println!("{:?}\n{:?}",point_1,point_2);
```

```
CloneablePoint { x: 2.2, y: -1.4 }
CloneablePoint { x: 2.2, y: -1.4 }
```

Can then be used recursively:

In [9]:
```
// will work
let tuple_point = (1,CloneablePoint{x:1.1,y:1.1});
let copy_tuple_point = tuple_point.clone();
```

CLONING

Option 2: Default cloning (with some extra benefits)

- Use `#[derive(Clone)]` in the definition
- Use method `.clone()` to clone an object

In [7]:
```
#[derive(Clone,Debug)]
struct CloneablePoint {
    x: f64,
    y: f64,
}
```

In [8]:
```
let point_1 = CloneablePoint{x:2.2,y:-1.4};
let point_2 = point_1.clone();
println!("{:?}\n{:?}",point_1,point_2);
```

```
CloneablePoint { x: 2.2, y: -1.4 }
CloneablePoint { x: 2.2, y: -1.4 }
```

Can then be used recursively:

In [9]:
```
// will work
let tuple_point = (1,CloneablePoint{x:1.1,y:1.1});
let copy_tuple_point = tuple_point.clone();
```

In [10]:
```
// won't work
let tuple_box = (1,BoxSize::new(1.1,1.2,1.3));
let copy_tuple_box = tuple_box.clone();
```

```
let copy_tuple_box = tuple_box.clone();
                               ^^^^^ method cannot be c
alled on `({integer}, BoxSize)` due to unsatisfied trai
t bounds
the method `clone` exists for tuple `({integer}, BoxSiz
e)`, but its trait bounds were not satisfied
help: consider annotating `BoxSize` with `#[derive(Clon
```

IMPLICIT COPYING

- Works for intergers, floats, booleans, ...
- Also for tuples made of items for which it works

In [11]:
```
let int = 3;
let int_2 = int;
println!("{}\n{}",int,int_2);
```

```
3
3
```

IMPLICIT COPYING

- Works for intergers, floats, booleans, ...
- Also for tuples made of items for which it works

In [11]:
```
let int = 3;
let int_2 = int;
println!("{}\n{}",int,int_2);
```

```
3
3
```

In [12]:
```
let tuple = (1.2,3.1);
let tuple_2 = tuple;
println!("{:?}\n{:?}",tuple,tuple_2);
```

```
(1.2, 3.1)
(1.2, 3.1)
```

IMPLICIT COPYING

- Works for intergers, floats, booleans, ...
- Also for tuples made of items for which it works

In [11]:
```
let int = 3;
let int_2 = int;
println!("{}\n{}",int,int_2);
```

```
3
3
```

In [12]:
```
let tuple = (1.2,3.1);
let tuple_2 = tuple;
println!("{:?}\n{:?}",tuple,tuple_2);
```

```
(1.2, 3.1)
(1.2, 3.1)
```

To make it work: use `#[derive(Copy)]` in the definition

- `(Clone)` needed a swell

In [13]:
```
#[derive(Copy,Clone,Debug)]
enum SearchResult {
    DidntFindIt,
    FoundIt(usize),
}
```

IMPLICIT COPYING

- Works for intergers, floats, booleans, ...
- Also for tuples made of items for which it works

In [11]:
```
let int = 3;
let int_2 = int;
println!("{}\n{}",int,int_2);
```

```
3
3
```

In [12]:
```
let tuple = (1.2,3.1);
let tuple_2 = tuple;
println!("{:?}\n{:?}",tuple,tuple_2);
```

```
(1.2, 3.1)
(1.2, 3.1)
```

To make it work: use `#[derive(Copy)]` in the definition

- `(Clone)` needed a swell

In [13]:
```
#[derive(Copy,Clone,Debug)]
enum SearchResult {
    DidntFindIt,
    FoundIt(usize),
}
```

In [14]:
```
let result = SearchResult::DidntFindIt;
let will_it_move = result;

println!("{:?}\n{:?}",result,will_it_move);
```

```
DidntFindIt
DidntFindIt
```

WHAT REALLY HAPPENS WITH `derive(Copy)` AND `derive(Clone)`

WHAT REALLY HAPPENS WITH `derive(Copy)` AND `derive(Clone)`

- Defining a specific method or methods (i.e., `clone`)

WHAT REALLY HAPPENS WITH `derive(Copy)` AND `derive(Clone)`

- Defining a specific method or methods (i.e., `clone`)
- It tells Rust that the type meets specific requirements
  - they are called a trait
  - to be covered later in class (next lecture?)

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [15]:
```
// auxiliary functions

fn display(x:&i32) {
    println!("{}",x);
}

fn double(x:&mut i32) {
    *x *= 2;
}
```

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [15]:
```
// auxiliary functions

fn display(x:&i32) {
    println!("{}",x);
}

fn double(x:&mut i32) {
    *x *= 2;
}
```

In [16]:
```
// two immutable references
let mut integer = 1;
{
    let ir = &integer;
    let ir2 = &integer;
    display(ir);
    display(ir2);
};
```

```
1
1
```

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [17]:
```
// one mutable reference
{
    let mr = &mut integer;
    double(mr);
    display(mr);
};
```

```
2
```

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [17]:
```
// one mutable reference
{
    let mr = &mut integer;
    double(mr);
    display(mr);
};
```

```
2
```

In [18]:
```
// two mutable references
{
    let mr = &mut integer;
    let mr2 = &mut integer;
    double(mr);
    double(mr2);
};
```

```
    let mr = &mut integer;
             ^^^^^^^^^^^^ first mutable borrow occurs h
ere
    let mr2 = &mut integer;
              ^^^^^^^^^^^^ second mutable borrow occurs
here
    double(mr);
           ^^ first borrow later used here
cannot borrow `integer` as mutable more than once at a
time
```

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [19]:
```
// immutable and mutable references
{
    let mr = &integer;
    let mr2 = &mut integer;
    display(mr);
    double(mr2);
};
```

```
    let mr2 = &mut integer;
              ^^^^^^^^^^^^ mutable borrow occurs here
    let mr = &integer;
             ^^^^^^^^ immutable borrow occurs here
    display(mr);
            ^^ immutable borrow later used here
cannot borrow `integer` as mutable because it is also b
orrowed as immutable
```

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [19]:

```
// immutable and mutable references
{
    let mr = &integer;
    let mr2 = &mut integer;
    display(mr);
    double(mr2);
};
```

```
    let mr2 = &mut integer;
              ^^^^^^^^^^^^ mutable borrow occurs here
    let mr = &integer;
             ^^^^^^^^ immutable borrow occurs here
    display(mr);
            ^^ immutable borrow later used here
cannot borrow `integer` as mutable because it is also b
orrowed as immutable
```

In [20]:

```
// immutable and mutable references
{
    let ir = &integer;
    display(ir);
    let mr2 = &mut integer;
    double(mr2);
    let ir2 = &integer;
    display(ir2);
};
```

```
2
4
```

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same thing from multiple places
- they can be passed around like values

In [19]:
```
// immutable and mutable references
{
    let mr = &integer;
    let mr2 = &mut integer;
    display(mr);
    double(mr2);
};
```

```
    let mr2 = &mut integer;
                ^^^^^^^^^^^^ mutable borrow occurs here
    let mr = &integer;
               ^^^^^^^^ immutable borrow occurs here
    display(mr);
             ^^ immutable borrow later used here
cannot borrow `integer` as mutable because it is also b
orrowed as immutable
```

In [20]:
```
// immutable and mutable references
{
    let ir = &integer;
    display(ir);
    let mr2 = &mut integer;
    double(mr2);
    let ir2 = &integer;
    display(ir2);
};
```

```
2
4
```

Rust can figure out which references no longer used

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same data from multiple places
- they can be passed around like values

RULES

- At most one mutable reference at a time
- Multiple immutable references allowed
- No mutable and immutable references at the same time

MULTIPLE REFERENCES AT ONCE

- useful for when we may want to access the same data from multiple places
- they can be passed around like values

RULES

- At most one mutable reference at a time
- Multiple immutable references allowed
- No mutable and immutable references at the same time

HOW IT COULD BE USEFUL

- More clear what is happening
  - Potential early bug detection
- Additional optimizations possible
- Multithreading (running things in parallel):
  - each thread accesses things through references
  - potentially very unpredictable behaviour without these rules

NOT COVERED TODAY: LIFETIMES

- how long a reference lives
- important for making sure that references passed around are not in conflict
- useful for dealing with some data processing patterns

NEW TOPIC: AVOIDING COPYING CODE FOR DIFFERENT TYPES

Python:

```
def max(x,y):
    return x if x > y else y
```

```
max(3,2)
```

```
3
```

```
max(3.1,2.2)
```

```
3.1
```

NEW TOPIC: AVOIDING COPYING CODE FOR DIFFERENT TYPES

Python:

```
def max(x,y):
    return x if x > y else y
```

```
max(3,2)
```

```
3
```

```
max(3.1,2.2)
```

```
3.1
```

Very flexible! Any downsides?

NEW TOPIC: AVOIDING COPYING CODE FOR DIFFERENT TYPES

Python:

```
def max(x,y):
    return x if x > y else y
```

```
max(3,2)
```

```
3
```

```
max(3.1,2.2)
```

```
3.1
```

Very flexible! Any downsides?

- Requires checking each time what types are used
- Runtime penalty

NEW TOPIC: AVOIDING COPYING CODE FOR DIFFERENT TYPES

Possible Rust "equivalent": create a copy for each type

In [21]:
```
fn max_i32(x:i32,y:i32) -> i32 {
    if x > y {x} else {y}
}

max_i32(3,8)
```

Out[21]: 8

In [22]:
```
fn max_f64(x:f64,y:f64) -> f64 {
    if x > y {x} else {y}
}

max_f64(3.3,8.1)
```

Out[22]: 8.1

NEW TOPIC: AVOIDING COPYING CODE FOR DIFFERENT TYPES

Possible Rust "equivalent": create a copy for each type

In [21]:
```
fn max_i32(x:i32,y:i32) -> i32 {
    if x > y {x} else {y}
}

max_i32(3,8)
```

Out[21]: 8

In [22]:
```
fn max_f64(x:f64,y:f64) -> f64 {
    if x > y {x} else {y}
}

max_f64(3.3,8.1)
```

Out[22]: 8.1

Lots of work! Make the compiler do it!

In [23]:
```
fn max<T>(x:T,y:T) -> T {
        if x > y {x} else {y}
}
```

```
        if x > y {x} else {y}
           ^ T
        if x > y {x} else {y}
               ^ T
        if x > y {x} else {y}
             ^
binary operation `>` cannot be applied to type `T`
help: consider restricting type parameter `T`

: std::cmp::PartialOrd
```

NEW TOPIC: AVOIDING COPYING CODE FOR DIFFERENT TYPES

Possible Rust "equivalent": create a copy for each type

In [21]:
```
fn max_i32(x:i32,y:i32) -> i32 {
    if x > y {x} else {y}
}

max_i32(3,8)
```
Out[21]: 8

In [22]:
```
fn max_f64(x:f64,y:f64) -> f64 {
    if x > y {x} else {y}
}

max_f64(3.3,8.1)
```
Out[22]: 8.1

Lots of work! Make the compiler do it!

In [23]:
```
fn max<T>(x:T,y:T) -> T {
        if x > y {x} else {y}
}
```
```
        if x > y {x} else {y}
           ^ T
        if x > y {x} else {y}
               ^ T
        if x > y {x} else {y}
             ^
binary operation `>` cannot be applied to type `T`
help: consider restricting type parameter `T`

: std::cmp::PartialOrd
```

In [24]:
```
// add info that elements of T are comparable
fn max<T:PartialOrd>(x:T,y:T) -> T {
        if x > y {x} else {y}
}

println!("{}",max(3,8));
println!("{}",max(3.3,8.1));
println!("{}",max('a','b'));
```
```
8
8.1
b
```

GENERICS / GENERIC DATA TYPES

In other programming languages:

- C++: templates
- Java: generics
- Go: generics
- ML, Haskell: parametric polymorphism

GENERICS / GENERIC DATA TYPES

In other programming languages:

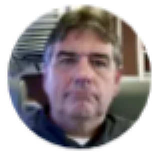
- C++: templates
- Java: generics
- Go: generics
- ML, Haskell: parametric polymorphism

Earlier this week:

Home > Software Development > Google Go

Go 1.18 arrives with much-anticipated generics

Now available in a production release, Go 1.18 introduces 'the most significant change' to Go since the programming language debuted in 2012.



By Paul Krill

Editor at Large, InfoWorld | MAR 15, 2022 2:00 PM PDT

USE WITH DATA TYPES

In [25]:

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
```

USE WITH DATA TYPES

In [25]:

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
```

In [26]:

```
let point_int = Point {x: 2, y: 3};
println!("{:?}", point_int);

let point_float = Point {x: 4.2, y: 3.1};
println!("{:?}", point_float);
```

```
Point { x: 2, y: 3 }
Point { x: 4.2, y: 3.1 }
```

USE WITH DATA TYPES

In [25]:
```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
```

In [26]:
```
let point_int = Point {x: 2, y: 3};
println!("{:?}", point_int);

let point_float = Point {x: 4.2, y: 3.1};
println!("{:?}", point_float);
```

```
Point { x: 2, y: 3 }
Point { x: 4.2, y: 3.1 }
```

Functions and methods for generic data types

In [27]:
```
impl<T> Point<T> {
    fn create(x:T,y:T) -> Point<T> {
        Point{x,y}
    }
}
```

USE WITH DATA TYPES

In [25]:

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}
```

In [26]:

```
let point_int = Point {x: 2, y: 3};
println!("{:?}", point_int);

let point_float = Point {x: 4.2, y: 3.1};
println!("{:?}", point_float);
```

```
Point { x: 2, y: 3 }
Point { x: 4.2, y: 3.1 }
```

Functions and methods for generic data types

In [27]:

```
impl<T> Point<T> {
    fn create(x:T,y:T) -> Point<T> {
        Point{x,y}
    }
}
```

In [28]:

```
let point = Point::create('a','b');
let point2 = Point::<char>::create('c','d');
let point3 : Point<char> = Point::create('c','d');
```

USE WITH DATA TYPES

Implementing a method

In [29]:
```
impl<T:Copy> Point<T> {
    fn swap(&mut self) {
        let z = self.x;
        self.x = self.y;
        self.y = z;
    }
}
```

USE WITH DATA TYPES

Implementing a method

In [29]:

```
impl<T:Copy> Point<T> {
    fn swap(&mut self) {
        let z = self.x;
        self.x = self.y;
        self.y = z;
    }
}
```

In [30]:

```
let mut point = Point::create(2,3);
println!("{:?}",point);
point.swap();
println!("{:?}",point);
```

```
Point { x: 2, y: 3 }
Point { x: 3, y: 2 }
```

USE WITH DATA TYPES

Specialized versions for different types

In [31]:
```
impl Point<i32> {
    fn do_you_use_f64(&self) -> bool {
        false
    }
}
```

In [32]:
```
impl Point<f64> {
    fn do_you_use_f64(&self) -> bool {
        true
    }
}
```

USE WITH DATA TYPES

Specialized versions for different types

In [31]:
```
impl Point<i32> {
    fn do_you_use_f64(&self) -> bool {
        false
    }
}
```

In [32]:
```
impl Point<f64> {
    fn do_you_use_f64(&self) -> bool {
        true
    }
}
```

In [33]:
```
let p_i32 = Point::create(2,3);
p_i32.do_you_use_f64()
```

Out[33]:
```
false
```

USE WITH DATA TYPES

Specialized versions for different types

In [31]:
```
impl Point<i32> {
    fn do_you_use_f64(&self) -> bool {
        false
    }
}
```

In [32]:
```
impl Point<f64> {
    fn do_you_use_f64(&self) -> bool {
        true
    }
}
```

In [33]:
```
let p_i32 = Point::create(2,3);
p_i32.do_you_use_f64()
```

Out[33]: false

In [34]:
```
let p_f64 = Point::create(2.1,3.1);
p_f64.do_you_use_f64()
```

Out[34]: true