ECE326

PROGRAMMING LANGUAGES

Lecture 16 : Ownership and References

Kuei (Jack) Sun

ECE

University of Toronto

Fall 2020

RAII

- Resource Acquisition is Initialization
  - Initialization of object only succeeds if it gets all its resources
- Other names
  - Constructor Acquires, Destructor Releases
  - Scope-based Resource Management

```
int main() {
    vector<Point> vps = { Point(1, 2), Point(3, 4) };
    …
    return 0;
    // Destructor of vps is called here. No resource or memory leak
    // is possible because C++ guarantees all stack objects are
    // destroyed (destructor called) at the end of enclosing scope
}
```

unique_ptr

- Introduced in C++11
- Overloads -> and * (dereference) operator
- Automatically deletes the heap object it contains

```
#include <memory>

int main() {
        unique_ptr<Point> point(new Point(1, 2));

        // point behaves like a normal pointer
        cout << "point is " << *point << endl;

        return 0;
        // Destructor of point deletes memory automatically,
        // no need to explicitly write “delete point;”
}
```

Move Semantic

- Transfers *ownership* of contained object to another

```
void foo(unique_ptr<Point> & giver) {
      unique_ptr<Point> taker = std::move(giver);
      cout << "Took over " << *taker << endl;
}
int main() {
      unique_ptr<Point> point(new Point(1, 2));
      foo(point);
      if (point)
            cout << "main still owns point" << endl;
      } else
            cout << "point is null" << endl;
      return 0;
}
```

heap object is now deleted here

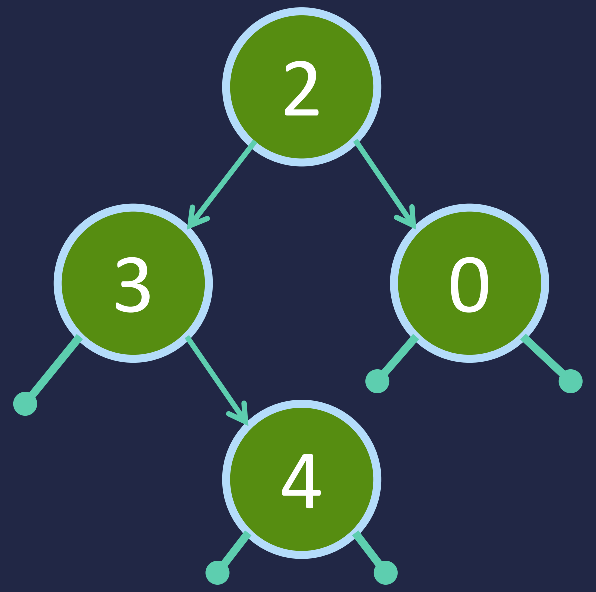
Took over (1, 2)
point is null

Box

- Rust’s unique_ptr
- Contains a heap allocated object
- Guarantees contained object exists
  - Unlike unique_ptr, its content may be null
- Usages
  - To implement recursive data structures (e.g. linked list)
  - To avoid copying (e.g. the contained object implements Copy)
  - To use dynamic dispatch (i.e. *trait objects*)

Recursive Structure

```
enum Tree {
  Leaf,
  Node(i64, Box<Tree>, Box<Tree>)
}

fn add_values(tree: Tree) -> i64 {
    match tree {
        Tree::Node(v, a, b) => {
            v + add_values(*a) +
            add_values(*b)
      },
      Tree::Leaf => 0
    }
}

assert_eq!(add_values(tree), 9);
```



Box is required to implement recursive structures otherwise the size of a tree node may be infinitely large! By using Box, Tree is now a fixed size object.

Mutability

- Mutability can change upon ownership transfer

```rust
let immutable_box = Box::new(5u32);

println!("immutable_box contains {}", immutable_box);
// *immutable_box = 4; <- cannot do this

// *Move* the box, changing ownership (and mutability)
let mut mutable_box = immutable_box;
// cannot access immutable_box from this point forward

println!("mutable_box contains {}", mutable_box);

// Modify the contents of the box
*mutable_box = 4;
println!("mutable_box now contains {}", mutable_box);
```

Return

- A function can return an object and give ownership

```
fn main() {
        let s1 = gives_ownership();
        let s2 = String::from("hello");
        // s2 is moved into takes_and_gives_back, which also
        // moves its return value into s3
        let s3 = takes_and_gives_back(s2);
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope
  // but was moved, so nothing happens. s1 goes out of scope and is
  // dropped.

fn gives_ownership() -> String {
        let some_string = String::from("hello");
        some_string
}

fn takes_and_gives_back(a_string: String) -> String {
        a_string
}
```

Borrowing

- Access to data without taking ownership
- Object is passed *by reference*
- Compiler guarantees reference will always be valid
  - This comes with a few restrictions and caveats

1. Cannot move an object if others hold reference to it
2. Only one mutable borrow at a time
3. Cannot mix mutable and immutable borrows
4. Cannot modify mutable object with immutable borrow

Valid Reference

- Object cannot be moved if another holds reference

```rust
// This function takes ownership of a box and destroys it
fn eat_box_i32(boxed_i32: Box<i32>) {
      println!("Destroying box that contains {}", boxed_i32);
}

// This function borrows an i32
fn borrow_i32(borrowed_i32: &i32) {
      println!("This int is: {}", borrowed_i32);
}

let boxed_i32 = Box::new(5_i32);
let ref_to_i32: &i32 = &boxed_i32;
// error: inner value is borrowed later in scope.
eat_box_i32(boxed_i32);
borrow_i32(ref_to_i32); // borrowing content of boxed_i32
```

Valid Reference

- Object cannot be moved if another holds reference
- Solution: Ensure reference goes out of scope first

```
fn eat_box_i32(boxed_i32: Box<i32>) {
      println!("Destroying box that contains {}", boxed_i32);
}
let boxed_i32 = Box::new(5_i32);
{
      let ref_to_i32: &i32 = &boxed_i32;
      // borrowing inner value of boxed_i32
      borrow_i32(ref_to_i32);
}     // ref_to_i32 goes out of scope here
// OK: no reference still in scope, safe to move
eat_box_i32(boxed_i32);
```

Mutability

- Object can be borrowed immutably many times
- But can only be mutably borrowed one at a time
  - The previous mutable borrow must go out of scope first
- Object cannot be borrowed both mutably and immutably at the same time
- Cannot borrow immutable objects as mutable
- Can borrow mutable objects as immutable
  - Cannot modify mutable objects while borrowed immutably

Mutability

- Object can be borrowed immutably many times

```
let mut point = Point { x: 0, y: 0, z: 0 };
let borrowed_point = &point;
let another_borrow = &point;

// Can access via the references and the original owner
println!("Point has coordinates: ({}, {}, {})",
            borrowed_point.x, another_borrow.y, point.z);

// NO! cannot borrow mutably, currently borrowed as immutable
let mutable_borrow = &mut point;

// NO! cannot modify, currently borrowed as immutable
point.x = 3;
```

Mutability

- Object can only be borrowed mutably one at a time

```
let mutable_borrow = &mut point;

// Change data via mutable reference
mutable_borrow.x = 5;
mutable_borrow.y = 2;

// NO! Can't borrow `point` as immutable because it's
// currently borrowed as mutable.
let y = &point.y;

// NO! `println!` takes an immutable reference.
println!("Point Z coordinate is {}", point.z);

// Ok! Mutable references can be passed in as immutable
println!("Point has coordinates: ({}, {}, {})",
      mutable_borrow.x, mutable_borrow.y, mutable_borrow.z);
```

Freezing

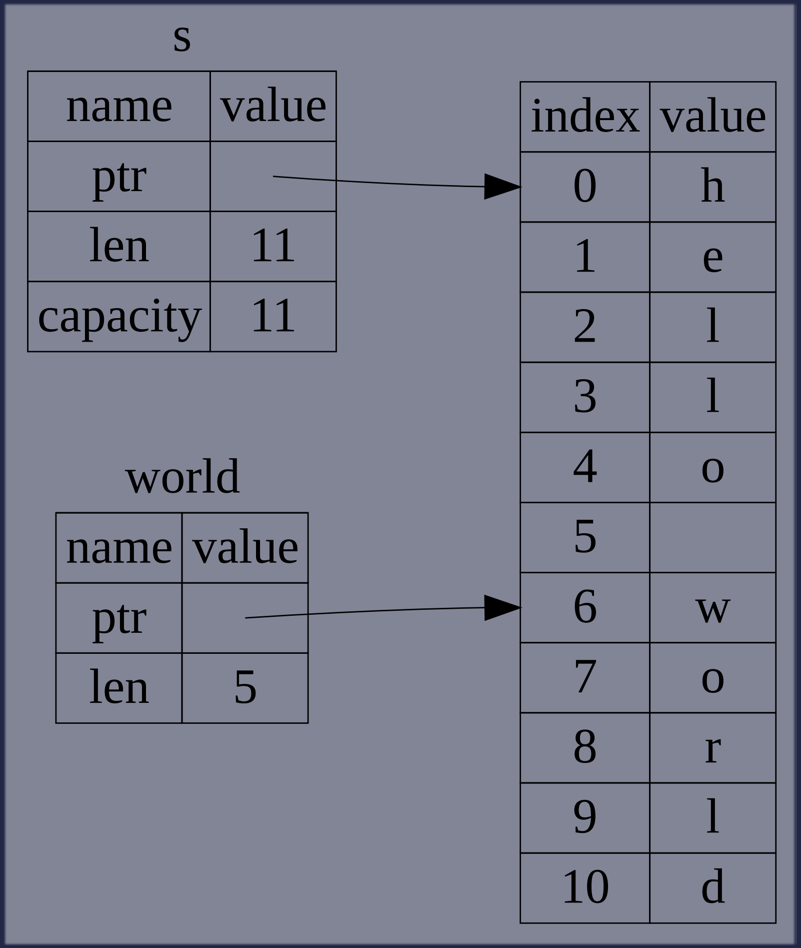
- Cannot modify a mutable while borrowed immutably
  - Solution: ensure immutably reference go out of scope first

```
let mut _mutable_integer = 7i32;
{
      // Borrow `_mutable_integer`
      let large_integer = & _mutable_integer;

      // Error! `_mutable_integer` is frozen in this scope
      _mutable_integer = 50;

      println!("Immutably borrowed {}", large_integer);
}     // `large_integer` goes out of scope

// Ok! `_mutable_integer` is not frozen in this scope
_mutable_integer = 3;
```

Slices

- A reference to parts of an object
  - E.g. string slice references a substring
  - E.g. array slice references a part of an array
- Created using range syntax

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```



Slices

- Syntax:
  - Can drop leading zero or trailing length

```rust
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];     // same as above

let len = s.len();
let slice = &s[3..len];
let slice = &s[3..];     // same as above

let slice = &s[0..len];
let slice = &s[..];      // same as above
```

Example

- Returns first word of a string

```
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[..]
}
```

Borrow

- Slices also borrow – must obey all borrowing rules

```
fn main() {
        let mut s = String::from("hello world");

        // borrowing as immutable here
        let word = first_word(&s);

        // error! borrowing as mutable here!
        s.clear();
}
```

```
error[E0502]: cannot borrow `s` as mutable because it is also
borrowed as immutable
```