Rust Ownership, References, and Lifetimes

CPSC 310 - Programming Languages

Rust Memory Management

- Rust's heap memory is managed without a garbage collector
- Type checking ensures there are no dangling pointers or buffer overflows
- Key features that ensure safety: ownership and lifetimes
  - Data has a single owner; immutable aliases are OK, but mutation only via owner or a single mutable reference
  - A lifetime determines how long data is alive

Rules of Ownership

1. Each value in Rust has a variable that is its owner
2. There can only be one owner at a time
3. When the owner goes out of scope, the value will be dropped

- Example

```
{ let mut s = String::from("hello"); // s is the owner
  s.push_str(", world!");
  println!("{}", s);
} // s's data is free by calling s.drop()
```

Assignment Transfers Ownership

- By default, an assignment moves data

```
let x = String::from("hello");
let y = x; // x moved to y
```

- A move leaves only one owner: `y`

```
println!("{}, world!", y); // ok
println!("{}, world!", x); // fails
```

- This prevents double free and use after free errors

Copy Trait

- Primitives do not transfer ownership on assignment

```
let x = 5;
let y = x;
println!("{} = 5", y); // ok
println!("{} = 5", x); // ok
```

- This works because primitives derive the `Copy` trait
  - This says that an assignment copies the entire object

Traits

- A trait is a way of saying that a type has a particular property
  - `Copy`: objects with this trait do not transfer ownership on assignment
  - `Move`: objects with this trait do transfer ownership on assignment
- Traits can also be used to indicate functions that a type must implement
  - Similar to Java interfaces

Explicit Copy

- Objects with the `Move` trait may be explicitly cloned with `.clone()`
  - Avoids loss of ownership, but at the cost of a copy
- Example

```
let x = String::from("hello");
let y = x.clone(); // x ownership is not moved
println!("{}, world!", y); // ok
println!("{}, world!", x); // ok
```

Ownership Transfer in Function Calls

- On a call, ownership passes from:
  - argument to called function's parameter
  - returned value to caller's receiver
- Example

```
fn main() {
    let s1 = String::from("hello");
    let s2 = id(s1); // s1 moved to arg
    println!("{}", s2); // ok
    println!("{}", s1); // fails
}

fn id(s: String) -> String {
    s // s moved to caller on return
}
```

References and Borrowing

- Create an alias by making a reference
  - an explicit, non-owning pointer to the original value
  - called borrowing; done with the & operator
- References are immutable by default

```
fn main() {
    let s1 = String::from("hello");
    let len = length(&s1); // lends reference
    println!("length: {}", len);
}

fn length(s: &String) -> usize {
    s.push_str("hi"); // fails due to immutability
    s.len()  // s dropped, but not its referent
}
```

Rules of References

1. At any given time, you can have either but not both of:
   - one mutable reference
   - any number of immutable references
2. References must always be valid (pointed-to value not dropped)

Borrowing and Mutation

- Make immutable references to mutable values
  - Shares read-only access through owner and borrowed references
  - Mutation is disallowed on original value until borrowed reference(s) dropped
- Example

```
{ let mut s1 = String::from("hello");
  { let s2 = &s1;
    println!("String is {} and {}", s1, s2); // ok
    s1.push_str(", world!"); // disallowed
  } // drops s2
  s1.push_str(", world!"); // ok
  println!("String is {}", s1); // prints updated s1
}
```

Mutable References

- To permit mutation via a reference, use `&mut` on a mutable reference

```
let mut s1 = String::from("hello");
{   let s2 = &s1;
    s2.push_str(", world!"); // fails; s2 immutable
} // s2 dropped
let s3 = &mut s1; // ok since s1 mutable
s3.push_str(", world!"); // ok since s3 mutable
println!("String is {}", s3); // ok
```

Ownership and Mutable References

- Can make only one mutable reference
- Doing so blocks use of the original
  - restored when the reference is dropped
- Example

```
let mut s1 = String::from("hello");
{   let s2 = &mut s1; // ok
    let s3 = &mut s1; // fails; second borrow
    s1.push_str(", world!"); // fails; second borrow
} // s2 dropped; s1 owner again
let s3 = &mut s1; // ok since s1 mutable
s1.push_str(", world!"); // ok
println!("String is {}", s1); // ok
```

Non Lexical Lifetimes

- Rust has been updated to support lifetimes that end before the surrounding scope

```
let mut s1 = String::from("hello");
{   let s2 = &mut s1; // ignored; never used
    let s3 = &mut s1; // ignored; never used
    s1.push_str(", world!"); // ok
    s1.push_str(", world!"); // fails; 2nd mutable ref
} // s2 dropped; s1 owner again
let s3 = &mut s1; // ok since s1 mutable
s1.push_str(", world!"); // ok
println!("String is {}", s1); // ok
```

The * Operator

- Given a value of type `T&` (or `T&mut`) use the `*` operator to read or write its underlying contents
- Example

```
let mut x = 2;
let mut y = 3;
let mut r = &mut x;
*r = 4;
r = &mut y;
*r = 5;
```

  - Note two uses of `mut` for `r` with different meanings

Immutable and Mutable References

- Cannot make a mutable reference if immutable references exist
  - Holders of an immutable reference assume the object will not change
- Example

```
let mut s1 = String::from("hello");
{   let s2 = &s1; // ok; s2 is immutable
    let s3 = &s1; // ok; multiple immutable refs allowe
    let s4 = &mut s1; // fails; immutable ref already
} // s2, s3, s4 dropped; s1 owner again
let s3 = &mut s1; // ok since s1 mutable
s1.push_str(", world!"); // ok
println!("String is {}", s1); // ok
```

Lifetimes

- References must always be to valid memory, not to memory that has been dropped
- A Rust lifetime is a type-level parameter that names the scope in which the data is valid
- Example: dangling reference

```
fn main() {
    let ref_invalid = dangle();
    println!("{}", ref_invalid);
}
fn dangle() -> &String {
    let s1 = String::from("hello");
    &s1
} // s1's value has been dropped
```

Lifetimes (continued)

- A lifetime corresponds with scope; `r`'s lifetime is 2-6, `x`'s lifetime is 4-5

```
1 {
2     let r = 5;
3     {
4         let x = &r;
5         println!("r: {}", r); // ok
6     }
7 }
```

- Variable `x` in scope while `r` is
  - A lifetime is a type variable that identifies a scope
  - `r`'s lifetime `'a` exceeds `x`'s lifetime `'b`

Lifetimes (continued)

- Changing the previous example; `r`'s lifetime `'a` is 2-7, `x`'s lifetime `'b` is 4-5

```
1 {
2     let r; // deferred init
3     {
4         let x = 5;
5         r = &x;
6     }
7     println!("r: {}", r); // fails
8 }
```

- Variable `x` goes out of scope while `r` still exists
  - `r`'s lifetime `'a` exceeds `x`'s lifetime `'b` so not safe to assign `x` to `r`

Lifetimes and Functions

- The lifetime of a reference is not always explicit

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() { x } else { y }
}

...

{  let x = String::from("hello");
   let z;
   {  let y = String::from(", world!");
      z = longest(&x, &y); // result is &y
   } // drop y, and also z
   println!("z = {}", z); // fails
```

Lifetime Parameters

- Each reference to a value of type `t` has a lifetime parameter
  - `&t` (and `&mut t`) – lifetime is implicit
  - `&'a t` (and `&'a mut t`) -- lifetime'a' is explicit
- Lifetime names
  - Implicit lifetime names are generated by the compiler
  - Global variables have the lifetime name `'static`
- Lifetimes can also be generic

```
// this says x and y must have the same lifetime
fn longest<'a>(x: &'a str, y:&'a str) -> &'a str {
    if x.len() > y.len() { x } else { y }
}
```

Lifetimes FAQ

- When do we use explicit lifetimes?
  - When more than one variable/type needs the same lifetime
- How do I tell the compiler exactly which lines of code lifetime `'a` covers?
  - You cannot; the compiler will calculate it.
- How does lifetime subsumption work?
  - If lifetime `'a` is longer than `'b`, we can use `'a` where `'b` is expected; we can require this with `'b: 'a`
- Can we use lifetimes in data definitions?
  - Yes, we can do this when we define structs, enums, etc.

Recap: Rules of References

1. At any given time, you can have either but not both of:
   - one mutable reference
   - any number of immutable references
2. References must always be valid (pointed-to value not dropped)