PART 4
Imperative Programming with Memory Updates

Mutable Update in Scala

Mutable Variables

- Mutable Variables
  - Use “var” instead of “val” and “def”
  - We can update the value stored in a variable.

```
class Main(i: Int) {
  var a = i
}

val m = new Main(10)
m.a  // 10
m.a = 20
m.a  // 20
m.a += 5  // m.a = m.a + 5
m.a  // 25
```

While loop

- While loop
  - Syntax: `while (`*cond*`)` *body*
    Executes *body* while *cond* holds.
  - It is equivalent to:

```
def mywhile(cond: =>Boolean)(body: =>Unit) : Unit =
  if (cond) { body; mywhile(cond)(body) } else ()
```

- Example

```
var i = 0
var sum = 0
while (i <= 100) { // mywhile (i <= 100) {
  sum += i
  i += 2
}
sum // 2550
```

For loop

➢For loop

- Syntax: `for (i <- ` *collection*`)` *body*
  Executes *body* for each `i` in *collection*.
- It is equivalent to:

```
def myfor[A](xs: Traversable[A])(f: A => Unit) : Unit =
  xs.foreach(f)
```

➢Example

```
var sum = 0
for (i <- 0 to 100 by 2) { // myfor (0 to 100 by 2) { i =>
  sum += i
}
sum // 2550
```

Immutability, Mutability & Ownership

Immutability for Guarantee & Mutability for Efficiency

- Immutable array

```
arr = Array.new([1;2;3])
… using arr …
arr’ = Array.set(arr, 0, 42)
f(arr’)
… using arr’ …
g()
… using arr’ …
```

- Mutable array

```
arr = Array.new([1;2;3])
… using arr …
Array.set(arr, 0, 42);
f(arr)
… using arr …
g()
… using arr …
```

Can we not have both? Mutability with Ownership!

- Immutable array

```
arr = Array.new([1;2;3])

… using arr …

arr' = Array.set(arr, 0, 42)

f(arr')

… using arr' …

g()

… using arr' …
```

- Mutation with Ownership

```
// own
arr = Array.new([1;2;3])

… using arr …
// mutable borrow
Array.set(&mut arr, 0, 42)
// immutable borrow
f(&arr)

… using arr …

g()

… using arr …
```

Types with Ownership

- Ownership Types

  - expresses and guarantees immutability
    - making code behavior more predictable

  - automatically deallocates memory when its ownership has gone
    - guaranteeing absence of use-after-free, double-free, memory-leak

  - disallows mutating the same value at the same time
    - guaranteeing data-race freedom in concurrent code

Programming with Mutation in a Principled Way!

The Rust Programming Language Book

https://doc.rust-lang.org/book/

Types, Values & Memory

Values and Types

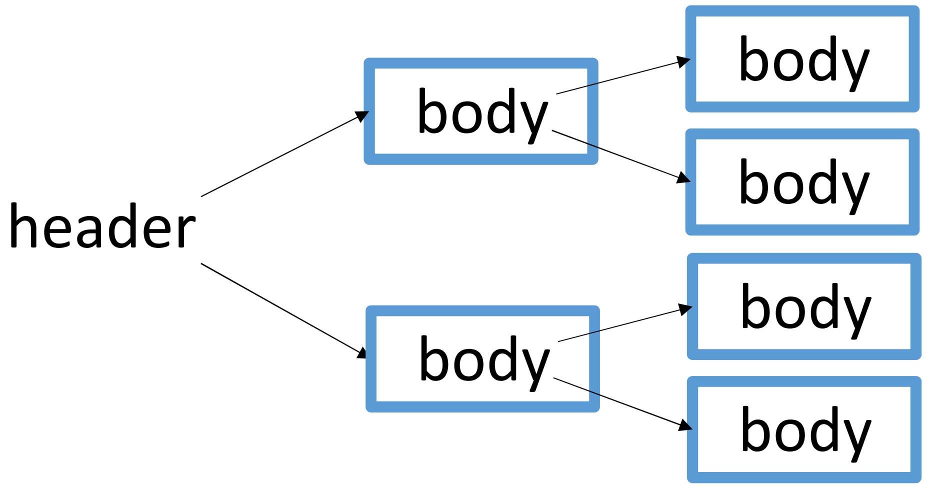
A type defines a set of values.

- For functionality, we only need to know the denotation of values.
- For efficiency, we also need to know the shape of values.

A value has a tree structure and consists of a header and bodies.

- A header is data that may contain scalar values and pointers to bodies.
- A body is data that must be stored in memory and may contain scalar values and pointers to other bodies.
- When passing or storing a value, only its header is passed or stored.

Types in Rust: i64, u64, (T1,T2), [T; 5], String, Vec<T>, …



Stack memory with static size and static lifetime

Stack memory consists of variables of sized types (ie, static header sizes) with scopes (ie, static lifetime).

A stack variable of a type T stores a header of the type T.

```
// variable arg of type i64 (8-byte header) with Scope 1
fn foo(arg: i64) -> i64 {
    let mut x = 10;   // variable x of type i64 with Scope 1
    {
        let y = arg + x;   // variable y of type i64 with Scope 2
        x = x + y;
    } // Scope 2
    let x2 = x; // variable x2 of type i64 with Scope 1
    let z = arg + x2;   // variable z of type i64 with Scope 1
    return z;
}  // Scope 1
```

Heap memory with dynamic size and dynamic lifetime

Heap memory consists of blocks of all types (ie, dynamic sizes) with no scopes (ie, dynamic lifetime).
A heap block can be a body of a value, which can also store headers of other values.

```
// variable sz of type usize (8-byte header) with Scope 1
fn gee(sz: usize) -> Vec<i64> {
// variable v of type Vec<i64> (24-byte header) with Scope 1
// The header points to a heap block of (sz * 8) bytes, filled with zeros
    let v : Vec<i64> = vec![0; sz];
// The header goes out of Scope 1 and the heap block is still alive
    return v;
}   // Scope 1
```

Principles of Ownership & Lifetime

Issues with mutation and deallocation for memory

<Problematic mutations>
Multiple parties modify a value logically simultaneously.

- Problem
  The value may become inconsistent despite valid individual updates.

A value is read logically while being modified.

- Problem
  An inconsistent intermediate state may be read.

<Problematic deallocation>
A value's body is deallocated while its header remains accessible.

- Problem
  The deallocated body may be accessed via the header (use-after-free bug).

Principles of Ownership

<Ownership rules>

- A header must be stored in exactly one location, which owns the header.
- Every heap block must be owned by exactly one header that directly points to it.
- Headers can be moved between locations.
- Headers are dropped when their owner is freed or overwritten.
- When dropped, a header's owned heap blocks are also freed.
- A stack variable is freed when it goes out of scope.

<Borrowing rules> (known as, "mutation-xor-sharing" checking)

- Mutable: single borrower has exclusive read/write access.
- Immutable: multiple borrowers and owner can read.

Principles of Lifetime

<Lifetime definition>

- Stack variables have a static lifetime bound by their scope.
- Heap blocks have a dynamic lifetime that ends when their owner is dropped.
- Headers have a dynamic lifetime that ends when their owner is freed or overwritten.

<Lifetime rules for borrowing> (known as, "borrow lifetime checking")

- A location's lifetime must be longer than any of its borrowers' lifetimes.

An example with ownership and borrowing

An example showing how ownership and borrowing work.

```
// arg owns a string, say ARG
fn gee(arg: String) -> String {
    let mut x = String::from("abc"); // x owns the string "abc"
    {
        let y = x + &arg;  // y owns the string "abc"+ARG, x not accessible
        x = y + &arg;  // x owns the string "abc"+ARG+ARG, y not accessible
    } // y is deallocated
    let z = x; // z owns the string "abc"+ARG+ARG, x not accessible
    return z; // the string "abc"+ARG+ARG is returned, z not accessible
} // arg, x, z are dallocated, the string ARG in arg is deallocated
```

Rust's approach

- References to a location of type T:
  - &T: immutable reference
  - &mut T: mutable reference

- Function signatures specify ownership/lifetime conditions

- Compiler checks ownership/lifetime in Safe Rust at two levels:
  - Stack variables (assuming function signatures)
  - Function implementations (verifying against signatures)

- Heap Memory Management:
  - Experts implement primitive heap types in Unsafe Rust with ownership/lifetime guarantees specified in function signatures
  - Users compose these primitives in Safe Rust to build complex data structures

Datatypes with Ownership & Lifetime

Scalar Types

```
fn main() {
    let mut x = [1,2,3,4];
    println!("{}", x[2]);
    x[2] = 42;
    println!("{}", x[2]);
    let y = x;
    println!("{}", x[0]);
    println!("{}", y[0]);
}
```

String and str types

```
fn main() {
    let mut s : &str = "abc";
    {
        let mut st : String = String::from(s);
        st.push_str("def");   // String::push_str(&mut st, "def");
        println!("{}", st);
        let s2 : &str = &st[1..4];
        // st.push_str("ghi");
        println!("{}", s2);
        // s = s2;
    }
    // println!("{}", s);
}
```

Function Types

```
fn foo(s: &String) -> &str {
    return &s[0..2];
}
fn gee(f: fn(&String) -> &str) {
    let s;
    // {
        let a = String::from("test");
        s = f(&a);
    // }
    println!("{}", s);
}
fn main() {
    gee(foo);
}
```

Function Types with lifetime annotation

```
fn foo<'a>(s: &'a String) -> &'a str {
    return &s[0..2];
}
fn gee (f: for <'a> fn(&'a String) -> &'a str) {
    let s;
    // {
        let a = String::from("test");
        s = foo(&a);
    // }
    println!("{}", s);
}
fn main() {
    gee(foo);
}
```

General lifetime annotation

```
fn foo1<'a>(s1: &'a String, s2: &'a String) -> &'a str {
    …
}
fn foo2<'a,'b>(s1: &'a String, s2: &'b String) -> &'a str {
    …
}
fn foo3<'a,'b,'c>(s1: &'a String, s2: &'b String) -> &'c str {
    …
}
fn foo4<'a,'b,'c, 'd, 'e>(s1: &'a String, s2: &'b String, s3: &'c String)
     -> (&'d str, &'e str)
where 'a: 'd, 'b:'d, 'b:'e, 'c:'e {
    …
}
```

General lifetime annotation

```
fn foo1<'a>(s1: &'a String, s2: &'a String) -> &'a str {
    return if rand::random() { &s1[0..1] } else { &s2[0..2] }; }
fn foo2<'a,'b>(s1: &'a String, s2: &'b String) -> &'a str {
    return &s1[0..1]; }
fn foo3<'a,'b,'c>(s1: &'a String, s2: &'b String) -> &'c str {
    return "abc"; }
fn foo4<'a,'b,'c, 'd, 'e>(s1: &'a String, s2: &'b String, s3: &'c String)
     -> (&'d str, &'e str)
where 'a: 'd, 'b:'d, 'b:'e, 'c:'e {
    let r1 = if rand::random() { &s1[0..1] } else { &s2[0..1] };
    let r2 = if rand::random() { &s2[0..1] } else { &s3[0..1] };
    (r1,r2)
}
```

Struct

```
struct User {
    active: bool,
    name: String,
}
fn main() {
    println!("Size: {} bytes", std::mem::size_of::<User>());
    let mut user_name = String::from("gil");
    let mut u = User { active: true, name: user_name };
    u.name.push_str(" hur");
    println!("{}, {}", u.active, u.name);
    user_name = u.name;
    println!("{}, {}", u.active, user_name);
    // println!("{} {}", u.active, u.name);
}
```

Struct with lifetime annotation

```
// Key Idea: Track the lifetime of each field of a struct separately.
struct User<'a,'b> { name: &'a mut String, email: &'b String }
fn main() {
    println!("Size: {} bytes", std::mem::size_of::<User>());
    let mut user_name = String::from("gil");
    let temp : &String;
    {   let mut user_email = String::from("gil.hur@sf.snu.ac.kr");
        let mut u = User { name: &mut user_name, email: &user_email };
        u.name.push_str(" hur");
        println!("{}, {}", u.name, u.email);
        temp = u.name; // temp = u.email;
        // u.name.push_str("xxx");
    }
    println!("{}", temp);
}
```

Struct with lifetime annotation (Variations)

```
// Key Idea: Track the lifetime of each field of a struct separately.
// Easy to understand if you inline the definition of a struct.
struct User1<'a> {
  name: &'a String,
  email: &'a String }
// User1 : for <'a> fn(&'a String, &'a String) : User1<'a>
// User1::name : for <'a> fn(User1<'a>) : &'a String
// User1::email : for <'a> fn(User1<'a>) : &'a String

struct User2<'a,'b> {
  user: User1<'a>,
  addr: &'b String }
// User2 : for <'a,'b> fn(User1<'a>, &’b String) : User2<'a,'b>
// User2::user : for <'a,'b> fn(User2<'a,'b>) : User1<'a>
// User2::addr : for <'a,'b> fn(User2<'a,'b>) : &'b String
```

Struct with lifetime annotation (Variations)

```
struct Foo<'a,'b> {
    x: &'a i32,
    y: &'b i32,
}

fn main() {
    let x = 1;
    let v;
    {
        let y = 2;
        let f = Foo { x: &x, y: &y };
        v = f.x;
    }
    println!("{}", *v);
}
```

Enum

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
use IpAddr::*;
fn main() {
    let ip1 = V4(147,36,52,255);
    match ip1 {  // match &ip1 // match &mut ip1
        V4(a1,a2,a3,a4) => println!("{a1:?} {a2:?} {a3:?} {a4:?}"),
        V6(addr) => println!("{addr:?}")   }
    let ip2 = V6(String::from("123.123.123.123.123.123"));
    match &ip2 {
        V4(a1,a2,a3,a4) => println!("{a1:?} {a2:?} {a3:?} {a4:?}"),
        V6(addr) => println!("{addr:?}")   }
}
```

Recursive Enum with Box

```
enum List<T> { Nil, Cons(T, Box<List<T>>) }
impl <T> List<T> {
    fn new() -> List<T> { List::Nil }
    fn cons(hd: T, tl: List<T>) -> List<T> {  List::Cons(hd, Box::new(tl)) }
}
fn mylen<T>(l: &List<T>) -> u64 {
    match l {
        List::Nil => 0,
        List::Cons(_, tl) => 1+mylen(tl) // 1+mylen(tl.as_ref())
    }
}
fn main() {
    let l = List::cons(0, List::cons(1, List::cons(2, List::new())));
    println!("{}", mylen(&l));
}
```

Update via a mutable reference

```
use std::mem;
fn test(x: &mut List<i32>) {
    // *x = List::cons(0, *x)
    let old = mem::replace(x, List::new());
    *x = List::cons(0, old)
}

fn main() {
    let mut l = List::cons(0, List::cons(1, List::cons(2, List::new())));
    println!("{}", mylen(&l));
    test(&mut l);
    println!("{}", mylen(&l));
}
```

Caveat: subtle immutability in let (seems a design bug)

- “let x : T” only guarantees its owned parts are unchanged
- “&x” guarantees that all data reachable from x are unchanged

```
struct Foo<'a> {
    x: i64,
    s: &'a mut String
}
fn main() {
    let mut str = "abc".to_string();
    let foo = Foo {x: 42, s: &mut str};
    println!("{} {}", foo.x, foo.s);
    foo.s.push_str("def");
    // (&foo).s.push_str("def");
    // foo.x = 37;
    println!("{} {}", foo.x, foo.s);
}
```

Hack: achieving proper immutability

```
struct Foo<'a> {
    x: i64,
    s: &'a mut String
}
fn main() {
    let mut str = "abc".to_string();
    let foo = Foo {x: 42, s: &mut str};
    let _foo_immutable = &foo;
    println!("{} {}", foo.x, foo.s);
    // foo.s.push_str("def");
    println!("{} {}", foo.x, foo.s);
    let _foo_end = _foo_immutable;
}
```

Smart Pointers

Box type

```
enum List<T> {
  Nil,
  Cons(T, Box<List<T>>)
}
```

Rc type

Use Rc type to replace static borrow lifetime checking with dynamic checking.

<Problem>

```
enum List<T> {
    Cons(T, Rc<List<T>>),
    Nil,
}
use crate::List::*;
fn test() -> (List<i64>,List<i64>) {
    let a = Cons(5, Box::new(Nil));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a));
    (b,c)
}
```

Rc type

```
use std::rc::Rc;
enum List<T> { Cons(T, Rc<List<T>>), Nil, }
use crate::List::*;
impl <T> List<T> {
    fn unfold(&self) -> Option<(&T,&List<T>)> {
        if let Cons(hd,tl) = self { Some((hd, tl.as_ref())) } else { None }
    } }
fn test() -> (List<i64>,List<i64>) {
    let a = Rc::new(Cons(5, Rc::new(Nil)));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
    (b,c)    }
fn main() {
    let (l1, l2) = test();    { let _t = l1; }
    println!("{}", l2.unfold().unwrap().1.unfold().unwrap().0)    }
```

RefCell type

Use RefCell type to replace static mutation-xor-sharing checking with dynamic checking.

<Problem>

Want to mutate the shared list in the previous example.

RefCell type

```
use std::rc::Rc; use std::cell::{RefCell, Ref, RefMut};
enum List<T> {
    Cons(T, Rc<RefCell<List<T>>>),
    Nil,
}
use crate::List::{Cons, Nil};
impl <T> List<T> {
    fn unfold<'a>(&'a self) -> Option<(&'a T, Ref<'a, List<T>>)> {
        if let Cons(hd,tl) = self { Some((hd, tl.borrow())) } else { None }
                                                        // tl.as_ref().borrow()
    }
    fn unfold_mut<'a>(&'a mut self) -> Option<(&'a mut T, RefMut<'a,
List<T>>)> {
        if let Cons(hd,tl) = self { Some((hd,tl.borrow_mut())) } else { None }
    } }
```

RefCell type

```
fn test() -> (List<i64>,List<i64>) {
    let a = Rc::new(RefCell::new(Cons(5, Rc::new(RefCell::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a));
    (b,c)
}
fn main() {
    let (mut l1, l2) = test();
    {
        let mut r = l1.unfold_mut().unwrap().1;
        *r = Cons(42, Rc::new(RefCell::new(Nil)));
    }
    println!("{}", l2.unfold().unwrap().1.unfold().unwrap().0)
}
```

Type class and Closure

Type class

```
trait Describable {
 fn describe(&self) -> String;
}
struct Book { title: String, author: String, }
impl Describable for Book {
  fn describe(&self) -> String {
    format!("{} by {}", self.title, self.author)
  } }

//fn print_desc(a: &impl Describable) -> ()
fn print_desc<T>(a: &T) -> () where T : Describable
{ println!("{}", a.describe()) }
fn main() {
  print_desc(&Book{title: "PP".to_string(), author: "Gil Hur".to_string()})
}
```

Dyn: dynamic dispatch

```
impl Describable for i64 {
    fn describe(&self) -> String {
        format!("64-bit-signed-integer: {}", *self)
    }
  }

fn main() {
    let book = Book{title: "PP".to_string(), author: "Gil Hur".to_string()};
    let ds : Vec<Box<dyn Describable>> =
        vec![Box::new(42), Box::new(book)];
    for d in &ds {
        println!("{}", d.describe())
        // cannot use print_desc : why?
    }
}
```

Closure type: Fn

```
fn test1<F>(f: F) -> usize
where F: Fn(usize)->usize {
    f(10) + f(20)
}
fn main() {
    let mut s = "abc".to_string();
    println!("{}", test1(|n|n+s.len()));
}
```

Closure type: FnMut

```
fn test2<F>(mut f: F)
where F: FnMut(&str)->() {
    f("gil");
    f("hur");
}
fn main() {
    let mut s = "abc".to_string();
    println!("{}", test1(|n|n+s.len()));
    test2(|x|s.push_str(x));
    println!("{}", test1(|n|n+s.len()));
}
```

Closure type: FnOnce

```
fn test3<F>(f: F)->String
where F: FnOnce(&str)->String {
    f("test")
}
fn main() {
    let mut s = "abc".to_string();
    println!("{}", test1(|n|n+s.len()));
    test2(|x|s.push_str(x));
    println!("{}", test1(|n|n+s.len()));
    let s2 = test3(|x|{s.push_str(x); s});
    println!("{}", s2);
}
```

Closure type: move

```
fn main() {
    let clo;
    {
        let mut s = "abc".to_string();
        println!("{}", test1(|n|n+s.len()));
        test2(|x|s.push_str(x));
        println!("{}", test1(|n|n+s.len()));
        let s2 = test3(|x|{s.push_str(x); s});
        println!("{}", s2);
        clo = move |n|n+s2.len();
    }
    println!("{}", test1(clo));
}
```

Thank you for your hard work!