Ownership

- Ownership of data in Rust
- Enforcing state transitions

Ownership

- Systems programs care about ownership of resources
  - To control memory management, close files, etc. → lecture 5
  - To model state machines
- Programmer maintains a mental model of what part of the code owns each resource
  - What function is responsible for calling **`free()`**, **`close()`**, etc.
  - Garbage collected languages still require understanding of ownership – but make **`free()`** call automatic when lifetime ends
  - C++ and Python tie resource ownership to scoping:

```
with open(filename) as file:
    data = file.read()
    ...
```

    gives automatic resource clean-up at end of scope

Ownership in Rust

- Rust tracks ownership of data – enforces that every value has a single owner
- Function calls explicitly manage ownership of values
  - Take explicit ownership of a value

```
fn consume(r : Resource) {
  ...
}
```

Function *takes ownership* of parameter passed by value

No longer accessible to caller; freed at end of function

  - Borrow a value

```
fn borrow(r : &Resource) {
  ...
}
```

Function *borrows* the parameter passed via reference

Ownership remains with caller

  - Return ownership of a value

```
fn generate() -> Resource {
  ...
}
```

Function *passes ownership* of return value to caller

Ownership in Rust: Example

```
struct Resource {
  value : u32
}
```

```
fn consume(r : Resource) {
  println!("consumed");
}

fn main() {
  let r = Resource{value: 42};
  consume(r);
  println!("{}", r.value);
}
```

Function *takes ownership* of parameter passed by value

No longer accessible to caller; freed at end of function

- The **`consume()`** function takes ownership of the resource – doesn't return it to the caller
- Above code won't compile: **`println!()`** cannot access **`r.value`**, since **`main()`** no longer has access to **`r`** because it gave ownership to **`consume()`**

State Machines and Ownership

- State machines manage resources
  - A network protocol manages connections, and the data sent over them
  - A device driver manages hardware resource
  - ...
- State transitions indicate when resources created/go out-of-scope

Ownership and state machines (1/2)

- **`struct`**-based approach to state machines uses ownership rules to enforce state transitions
  - Methods that change state take ownership of **`self`**, return new **`struct`**:

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, (self, LoginError)> {
    ...
  }

  fn disconnect(self) -> NotConnected {
    ...
  }
}
```

  - e.g., the **`login()`** function consumes its **`UnauthenticatedConnection`** and returns a new **`AuthenticatedConnection`** on success
    - The compiler enforces this – the **`UnauthenticatedConnection`** is not accessible after this call; all resources it owned are reclaimed
    - Except any values the **`login()`** method explicitly copies to the **`AuthenticatedConnection`**

Ownership and state machines (2/2)

- **`struct`**-based approach to state machines uses ownership rules to enforce state transitions
  - Guarantees resource cleanup on state transition
  - Better for ensuring resources are cleaned-up after use
- **`enum`**-based approach to state machines makes the state transition diagram clearer, but relies on programmer discipline to clean-up
  - Better for ensuring complex state machines correctly reflected in code

Type-driven Development – Recap

- **Define the types** first
  - Define concrete numeric types, identifiers
  - Define `enum` types to represent alternatives
  - Indicate optional values, results, error types

Type-driven Development – Recap

- **Define the types** first
- Using the types as a guide, **write the functions**
  - Write the input and output types
  - Write the function, using the structure of the types as a guide
  - Make state machines explicit
  - Consider ownership of data

Type-driven Development – Recap

- **Define the types** first
- Using the types as a guide, **write the functions**
- **Refine** and edit types and functions as necessary
  - Use the compiler as a tool to help you debug your design

Type-driven Development – Recap

- **Define the types** first
- Using the types as a guide, **write the functions**
- **Refine** and edit types and functions as necessary
- Don't think of the types as checking the code, think of them as a plan, a model, for the solution – as machine checkable documentation – use the compiler as a debugging tool before running the code

Summary

- Type-drive development
- Design patterns
- State machines
- Ownership