

Type State

CS242

Lecture 9

APIs

Consider a typical use of a file API:

```
File f = open("MyFile");
```

```
Char c = f.read();
```

```
f.close()
```

APIs

File operations mutate state and can change what operations are legal

```
File f = open("MyFile"); // opening the file enables reading
```

```
Char c = f.read();
```

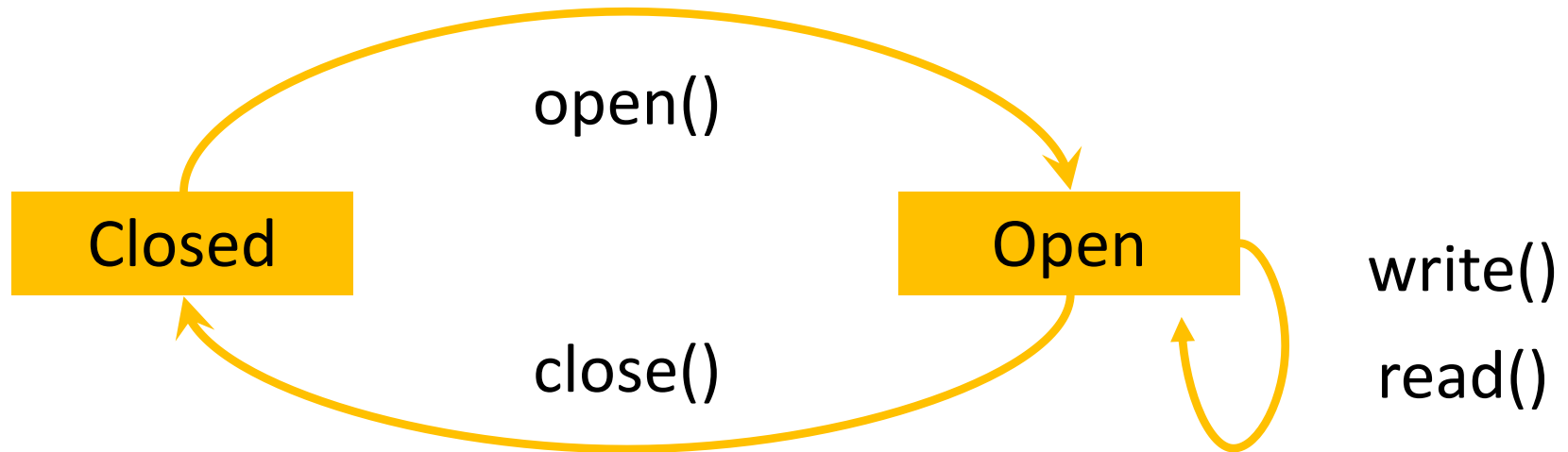
```
f.close() // but once the file is closed, reading is no  
// legal even though f is still available
```

Stateful APIs

- Many APIs are stateful in this way
- Correct usage depends on the state of the system
 - Which changes over time as API calls are made
- States and state changes are commonly modeled using finite state machines

Files

- Files can be open or closed
- A closed file can only be opened
- An open file can be read, written or closed



Discussion

- Many similar examples of this two state machine in real APIs:
- File open/close
- Lock acquire/release
- Message encrypted/decrypted
- User logged in/logged out
- Resource initialized/uninitialized

More Complex Machines

- File example can be refined, revealing more states specific to reading/writing

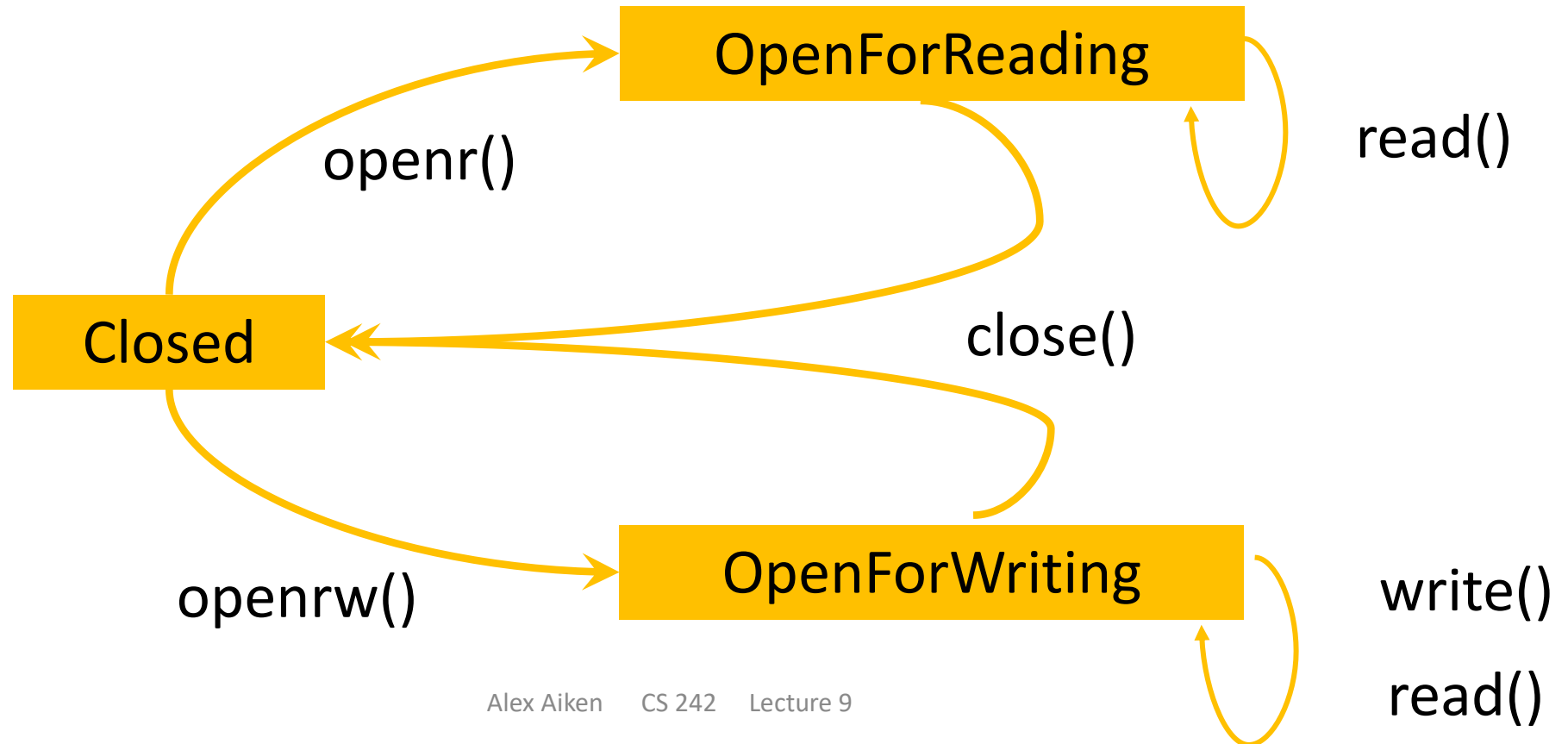
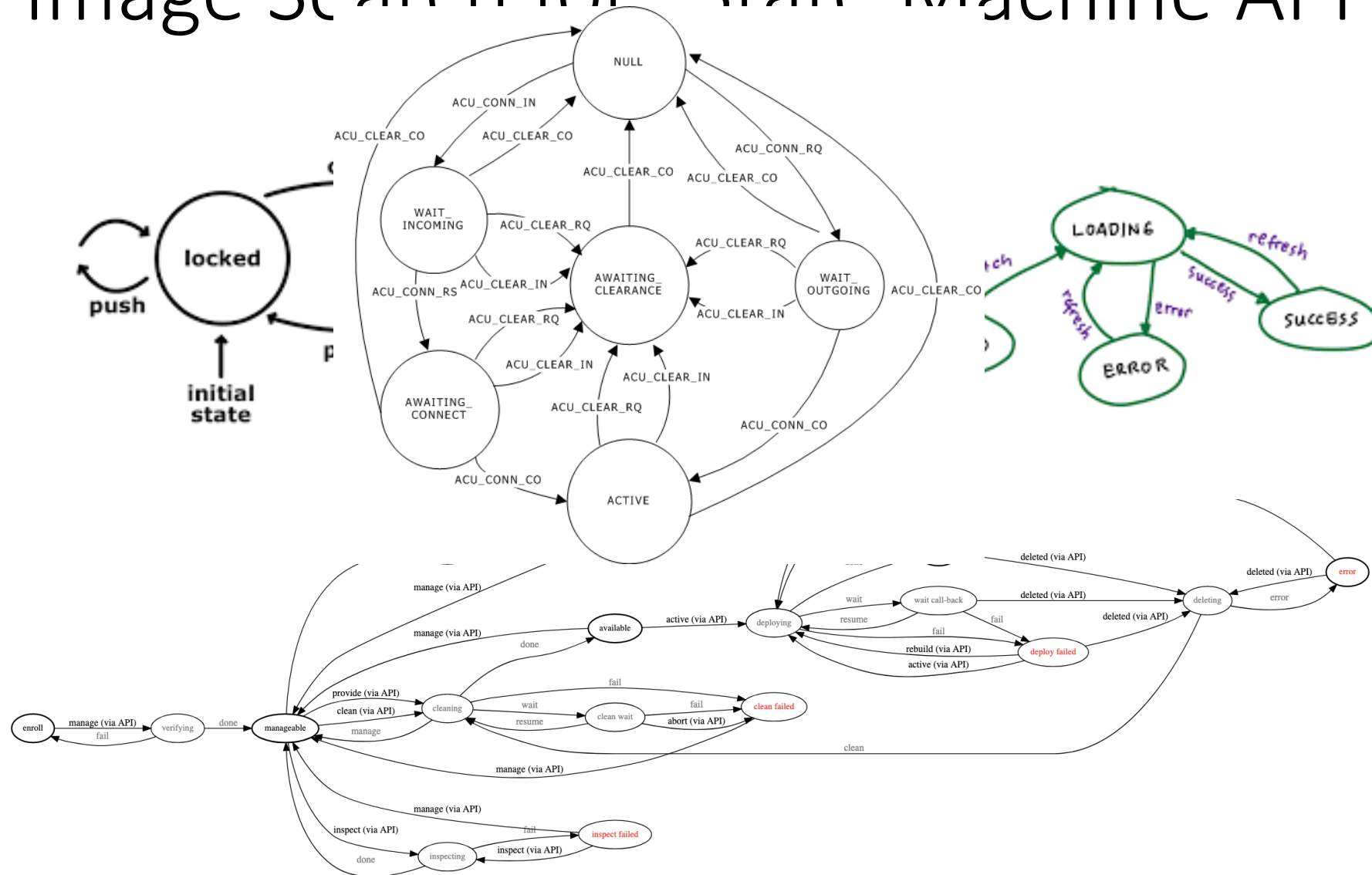


Image Search for “State Machine API”



Enforcing APIs

- We would like to enforce that API usage is correct according to a state machine specification
- Guarantees
 - We start in a correct state
 - The only calls that are made are valid in the current state
 - Can also enforce that we end in a correct state
 - e.g., Don't leave a resource locked forever

Type State

- Idea: Use the type system
- Each state machine state corresponds to a distinct type
- Method invocations cause transitions in the state diagram
 - `open(): ClosedFile -> OpenFile`
 - `close(): OpenFile -> ClosedFile`
 - `read(): OpenFile -> OpenFile`

A Possible Implementation

```
Class FileRecord {  
    File f;  
    Str name; }  

```

```
Class ClosedFile {  
    FileRecord r;  
    new ClosedFile(r: FileRecord);  
    open(): OpenFile; }  

```

```
Class OpenFile {  
    FileRecord r;  
    new OpenFile(name: Str);  
    read(): Char;  
    close(): ClosedFile }  

```

Example

```
f = new OpenFile("foo")    // f is of type OpenFile  
c = f.read()  
x = f.close()              // x is of type ClosedFile
```

// but ...

```
d = f.read()  // f has been closed!
```

Discussion

- The `OpenFile` and `ClosedFile` types allow us to express the transition from an open file to a closed file in the type system
 - And even back again
- The types only permit valid operations for the state of the file
- But ...
 - Nothing gets rid of the `OpenFile` objects when the file is closed
 - We could null out the `FileRecord`, but the `OpenFile` object would still exist

What Do We Want?

```
f = new OpenFile("foo")
```

```
c = f.read()
```

```
x = f.close()
```

```
// ideally at this point the f object would be deallocated
```

```
// and it would also be impossible to invoke operations on f
```

```
// So...
```

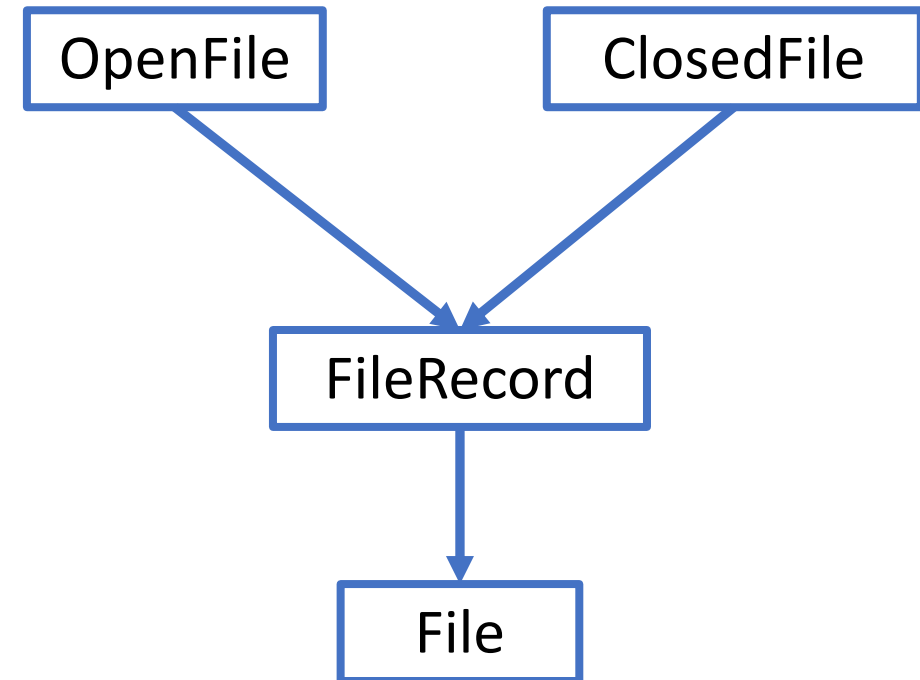
```
d = f.read() // this should be a type error!
```

The Real Problem

- Types are not the problem
 - We can express the state transitions as functions that map one type to another
- Keeping track of object references is the problem
 - In particular, tracking *aliasing* between names

Aliasing

- OpenFile and ClosedFile methods both modify the state of the File object in the FileRecord
- An example of *aliasing*
 - Having multiple names (ways to access) the same value
- Either object can change the state of the file without the knowledge of the other



Aliasing Control

A Classic Example

```
copy(char *x, char *y) {  
    ...  
}
```

But what about `copy(a,a)`?

A Classic Example

```
copy(restrict char *x, restrict char *y) {  
    ...  
}
```

Semantics: In C, a restricted pointer cannot be aliased to any other pointer in scope.

A Point of View

- Aliasing is bad
- State can be modified through one name and those changes are visible through a different name
 - Leads to subtle and difficult bugs
- But aliasing is very common in real programs
 - Impossible to avoid
 - E.g., references passed as arguments to functions
 - Object-oriented code is particularly prone to generating aliasing

Idea #1

- Maybe aliasing is not the problem ...
- Problems arise only when aliasing is combined with mutation
- So, disallow mutation!
 - The pure functional programming viewpoint

Could This Really Work?

- People have studied pure functional languages for decades
 - No mutation, whenever a data structure is changed a copy is made
- A surprising number of standard algorithms are just as efficient without mutation of state
 - Sometimes just amortized bounds, but that is still quite good!
- But there are some operations that seem to fundamentally require mutation to be efficient
 - Update in place of an array is $O(1)$
 - The best known functional update is $O(\log N)$ in the size of the array

A Practical Approach

- Split the world into mutable and immutable values
- Rust
 - `let x = 5` `// immutable`
 - `let mut x = 5` `// mutable`
 - `x = 3` `// only allowed if x is mutable`
- ML
 - `let x = 5` `// immutable`
 - `let x = ref 5` `// mutable`
 - `x := 3`

Separating Mutable & Immutable

- Not entirely a new idea
 - E.g., `const` in C
- Gaining in popularity
 - More languages are making this distinction
 - With immutability being the default
- Now accepted as a good idea
 - Limit the possibility of mutation to places it is really needed
 - Make these points obvious in the syntax & types

Idea #2

- Control aliasing in the type system
 - Track it, restrict it, or even disallow it
- Ownership types
 - Track aliases using types
- There is a large literature on ownership types
 - Some quite elaborate ...

Ownership in Rust

- Rust has a simple ownership model
- There is always a single *owner* variable of every object
 - Owning = responsible for the resources of the object
- Implications
 - An object with no owner is deallocated
 - When an owner goes out of scope, the owned object is deallocated
 - Copies transfer ownership
 - $x = y$ removes ownership from y and transfers it to x
 - y can no longer be used after the assignment

Ownership Example

```
fn main(){
    let v = vec[1,2,3];    // v owns the vector
    let v2 = v;            // moves ownership to v2
    // println(v) compile-time error!
    display(v2);           // ownership is moved to display
    // println(v2); compile-time error!
}

fn display(v:Vec<i32>){
    println(v);
    // v goes out of scope here and the vector is deallocated
}
```

Another Ownership Example

```
A() {  
  X = new Foo(); // X is the owner  
  Y = Bar(X)      // ownership is transferred to the argument of Bar  
                  // and then back to Y  
  // Y goes out of scope and the vector is deallocated  
}
```

```
Bar(Z) {  
  return Z // ownership is transferred back to the caller  
}
```

Lifetimes

- Rust reasons about aliasing/ownership by using *lifetimes*
- The lifetime of a variable is the span between
 - The definition (first use)
 - The last use
- Roughly: Lifetimes of aliases cannot overlap
 - Thus it is important to minimize lifetimes
 - We will refine this rule shortly

Lifetimes

```
fn main(){
```

```
    let v = vec[1,2,3]; // vector v owns the object
```

```
    let v2 = v;          // moves ownership to v2
```

```
    // println(v) compile-time error!
```

```
    display(v2);         // ownership is moved to display
```

```
    // println(v2); compile-time error!
```

```
}
```

```
fn display(v:Vec<i32>){
```

```
    println(v);
```

```
    // v goes out of scope here and the vector is deallocated
```

```
}
```

Lifetimes: A Compile Time Error

```
fn main(){  
    let v = vec[1,2,3]; // vector v owns the object  
    let v2 = v;          // moves ownership to v2  
    println(v)           // compile-time error!  
    display(v2);         // ownership is moved to display  
    // println(v2); compile-time error!  
}
```

```
fn display(v:Vec<i32>){  
    println(v);  
    // v goes out of scope here and the vector is deallocated  
}
```

Lifetimes: A Fix

```
fn main(){
```

```
    let v = vec[1,2,3];    // vector v owns the object 2
```

```
    println(v)            // compile-time error!
```

```
    let v2 = v;            // moves ownership to v2
```

```
    display(v2);           // ownership is moved to display
```

```
    // println(v2); compile-time error!
```

```
}
```

```
fn display(v:Vec<i32>){
```

```
    println(v);
```

```
    // v goes out of scope here and the vector is deallocated
```

```
}
```


Aliasing Control in Rust

- Having only a single owner is painful in many situations
 - Can never have another name for an object or even a piece of an object
 - E.g., makes it impossible to name a subarray
 - And we often don't need to take ownership anyway
- Rust allows the creation of explicit aliases
 - called *references* or *borrows*
- There are two kinds of references:
 - mutable
 - immutable

Example: Immutable Reference

```
A() {  
  X = new Foo(); // X is the owner  
  Y = &X         // An immutable reference to X; X is still the owner.  
  Bar(Y)         // pass an immutable reference to Bar  
}
```

```
Bar(&Z) {  
  ... = .. Z ... // can read from Z in Bar as many times as we like  
  // Global.x = Z but if we try to store Z somewhere that outlives Bar we get an error  
}
```

Example: Immutable Reference

```
A() {  
  X = new Foo(); // X is the owner  
  Y = &X         // An immutable reference to X; X is still the owner.  
  Bar(Y, Y)      // pass two immutable references to Bar  
}
```

```
Bar(&A, &B) {  
  ... = .. A ... // can read from A and B in Bar as many times as we like  
  ... = ... B ...  
  
}
```

Example: Mutable Reference

```
A() {  
  X = new Foo();    // X is the owner  
  Y = &mut X;       // Y is a mutable reference to X  
  Bar(Y)            // pass a mutable reference to Bar  
}
```

```
Bar(&mut Z) {  
  Z.f = ... // can mutate Z  
}
```

Example: Mutable Reference

```
A() {  
  X = new Foo();    // X is the owner  
  Y = &mut X;       // Y is a mutable reference to X  
  Bar(Y, Y)         // Error: Cannot have two mutable references to X  
}
```

```
Bar(&mut A, &mut B) { // since A and B are mutable, they cannot alias  
  A.f = ... // can mutate A  
  B.f = ... // can mutate B  
}
```

Reference Rules

- A reference cannot outlive its referent
 - No dangling references ...
- A mutable reference cannot be aliased
- Meaning:
 - There can be one mutable reference to an object in scope
 - There can be any number of immutable references
 - The owner's lifetime must contain the lifetime of all references

Example: Immutable Reference

```
A() {
```

```
  X = new Foo(); // X is the owner
```

```
  Y = &X          // An immutable reference to X; X is still the owner.
```

```
  Bar(Y)          // pass an immutable reference to Bar; the reference's lifetime is the lifetime of Bar
```

```
}
```

```
Bar(&Z) {
```

```
  ... = .. Z ...    // can read from Z in Bar as many times as we like
```

```
  // Global.x = Z    but if we try to store Z somewhere that outlives Bar we'll get an error
```

```
}
```

Back to Type State

- Recall type state allowed us to express state changes in the type system:

```
// f: OpenFile
```

```
x = f.close() // x is of type ClosedFile
```

- Ownership types can be used to make objects corresponding to the previous API state inaccessible

The Type State Example

```
f = new OpenFile("foo")    // f is the owner
```

```
c = f.read()
```

```
x = f.close()  // x is of type ClosedFile, transfers file ownership f -> x
```

```
// then ...
```

```
d = f.read()  // generates a type error: f is not the owner
```

Type State Abstraction

- Objects of a given state support only the methods available in that state
- Methods that change the internal API state
 - Return a new object of the appropriate type
 - Transfer ownership to the new object so that the references to the previous state become inaccessible
- Thus only objects of the correct type are ever accessible by the program

Discussion

- Ownership rules are very restrictive
 - Program must be *linear* in owned objects
 - Exactly one owner at all times
 - Implies no copies can be made
- Three techniques help:
 - Using immutable data wherever possible
 - Deep copies are OK (*cloning*)
 - Borrowing creates a reference that can be used
 - Does not transfer ownership
 - Implies a borrowed reference cannot deallocate an object
 - The owner cannot deallocate an object until all borrowed references are returned
 - Borrowed references have a different syntax and type

Ownership in Practice

- Ownership has been studied for > 20 years
- Rust is the first full language to support ownership types
 - The major new feature
- Experience is that Rust's ownership system helps
 - Enables manually managed memory without the bugs
 - Makes it possible to write efficient and correct code
 - Ownership types are the key
 - Which is not to say ownership is always easy to use ... it's not!