# Ownership & Lifetimes

## CIS 198 Lecture 1

# Ownership & Borrowing

- Explicit ownership is the biggest new feature that Rust brings to the table!
- Ownership is all[1] checked at compile time!
- Newcomers to Rust often find themselves "fighting with the borrow checker" trying to get their code to compile

[1]*mostly*

# Ownership

- A variable binding *takes ownership* of its data.
  - A piece of data can only have one owner at a time.
- When a binding goes out of scope, the bound data is released automatically.
  - For heap-allocated data, this means de-allocation.
- Data *must be guaranteed* to outlive its references.

```
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];

    v1.pop();
    v1.push(4);

    // At the end of the scope, v1 goes out of scope.
    // v1 still owns the Vec object, so it can be cleaned up.
}
```

# Move Semantics

```
let v1 = vec![1, 2, 3];

// Ownership of the Vec object moves to v2.
let v2 = v1;

println!("{}", v1[2]); // error: use of moved value `v1`
```

- `let v2 = v1;`
  - We don't want to copy the data, since that's expensive.
  - The data cannot have multiple owners.
  - Solution: move the Vec's ownership into `v2`, and declare `v1` invalid.
- `println!("{}", v1[2]);`
  - We know that `v1` is no longer a valid variable binding, so this is an error.
- Rust can reason about this at compile time, so it throws a compiler error.

# Move Semantics

- Moving ownership is a compile-time semantic; it doesn't involve moving data during your program.
- Moves are automatic (via assignments); no need to use something like C++'s `std::move`.
  - However, there are functions like `std::mem::replace` in Rust to provide advanced ownership management.

# Ownership

- Ownership does not always have to be moved.
- What would happen if it did? Rust would get very tedious to write:

```rust
fn vector_length(v: Vec<i32>) -> Vec<i32> {
    // Do whatever here,
    // then return ownership of `v` back to the caller
}
```

- You could imagine that this does not scale well either.
    - The more variables you had to hand back, the longer your return type would be!
    - Imagine having to pass ownership around for 5+ variables at a time :(

# Borrowing

- Obviously, this is not the case.
- Instead of transferring ownership, we can *borrow* data.
- A variable's data can be borrowed by taking a reference to the variable; ownership doesn't change.
  - When a reference goes out of scope, the borrow is over.
  - The original variable retains ownership throughout.

```rust
let v = vec![1, 2, 3];

// v_ref is a reference to v.
let v_ref = &v;

// use v_ref to access the data in the vector v.
assert_eq!(v[1], v_ref[1]);
```

# Borrowing

- Caveat: this adds restrictions to the original variable.
- Ownership cannot be transferred from a variable while references to it exist.
  - That would invalidate the reference.

```rust
let v = vec![1, 2, 3];

// v_ref is a reference to v.
let v_ref = &v;

// Moving ownership to v_new would invalidate v_ref.
// error: cannot move out of `v` because it is borrowed
let v_new = v;
```

# Borrowing

```rust
/// `length` only needs `vector` temporarily, so it is borrowe
fn length(vec_ref: &Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on i
    vec_ref.len()
    // you can also explicitly dereference.
    // (*vec_ref).len()
}

fn main() {
    let vector = vec![];
    length(&vector);
    println!("{:?}", vector); // this is fine
}
```

- Note the type of `length`: `vec_ref` is passed by reference, so it's now an `&Vec<i32>`.
- References, like bindings, are *immutable* by default.
- The borrow is over after the reference goes out of scope (at the end of `length`).

# Borrowing

```rust
/// `push` needs to modify `vector` so it is borrowed mutably.
fn push(vec_ref: &mut Vec<i32>, x: i32) {
    vec_ref.push(x);
}

fn main() {
    let mut vector: Vec<i32> = vec![];
    let vector_ref: &mut Vec<i32> = &mut vector;
    push(vector_ref, 4);
}
```

- Variables can be borrowed by *mutable* reference: `&mut vec_ref`.
    - `vec_ref` is a reference to a mutable `Vec`.
    - The type is `&mut Vec<i32>`, not `&Vec<i32>`.
- Different from a reference which is variable.

# Borrowing

```
/// `push` needs to modify `vector` so it is borrowed mutably.
fn push2(vec_ref: &mut Vec<i32>, x: i32) {
    // error: cannot move out of borrowed content.
    let vector = *vec_ref;
    vector.push(x);
}

fn main() {
    let mut vector = vec![];
    push2(&mut vector, 4);
}
```

- Error! You can't dereference `vec_ref` into a variable binding because that would change the ownership of the data.

# Borrowing

- Rust will auto-dereference variables...
  - When making method calls on a reference.
  - When passing a reference as a function argument.

```rust
/// `length` only needs `vector` temporarily, so it is borrowe
fn length(vec_ref: &&Vec<i32>) -> usize {
    // vec_ref is auto-dereferenced when you call methods on i
    vec_ref.len()
}

fn main() {
    let vector = vec![];
    length(&&&&&&&&&&&&vector);
}
```

# Borrowing

- You will have to dereference variables...
  - When writing into them.
  - And other times that usage may be ambiguous.

```
let mut a = 5;
let ref_a = &mut a;
*ref_a = 4;
println!("{}", *ref_a + 4);
// ==> 8
```

# ref

```
let mut vector = vec![0];

{
      // These are equivalent
      let ref1 = &vector;
      let ref ref2 = vector;
      assert_eq!(ref1, ref2);
}

let ref mut ref3 = vector;
ref3.push(1);
```

- When binding a variable, `ref` can be applied to make the variable a reference to the assigned value.
  - Take a mutable reference with `ref mut`.
- This is most useful in `match` statements when destructuring patterns.

# `ref`

```rust
let mut vectors = (vec![0], vec![1]);
match vectors {
    (ref v1, ref mut v2) => {
        v1.len();
        v2.push(2);
    }
}
```

- Use `ref` and `ref mut` when binding variables inside match statements.

# Copy Types

- Rust defines a trait[1] named `Copy` that signifies that a type may be copied instead whenever it would be moved.
- Most primitive types are `Copy` (`i32`, `f64`, `char`, `bool`, etc.)
- Types that contain references may not be `Copy` (e.g. `Vec`, `String`).

```
let x: i32 = 12;
let y = x; // `i32` is `Copy`, so it's not moved :D
println!("x still works: {}, and so does y: {}", x, y);
```

[1] Like a Java interface or Haskell typeclass

# Borrowing Rules

***The Holy Grail of Rust***

Learn these rules, and they will serve you well.

- You can't keep borrowing something after it stops existing.
- One object may have many immutable references to it (`&T`).
- **OR** *exactly one* mutable reference (`&mut T`) (not both).
- That's it!

# Borrowing Prevents...

- Iterator invalidation due to mutating a collection you're iterating over.
- This pattern can be written in C, C++, Java, Python, Javascript...
  - But may result in, e.g, `ConcurrentModificationException` (at runtime!)

```
let mut vs = vec![1,2,3,4];
for v in &vs {
    vs.pop();
    // ERROR: cannot borrow `vs` as mutable because
    // it is also borrowed as immutable
}
```

- `pop` needs to borrow `vs` as mutable in order to modify the data.
- But `vs` is being borrowed as immutable by the loop!

# Borrowing Prevents...

- Use-after-free
- Valid in C, C++...

```
let y: &i32;
{
    let x = 5;
    y = &x; // error: `x` does not live long enough
}
println!("{}", *y);
```

- The full error message:

```
error: `x` does not live long enough
note: reference must be valid for the block suffix following statement
    0 at 1:16
...but borrowed value is only valid for the block suffix
    following statement 0 at 4:18
```

- This eliminates a *huge* number of memory safety bugs *at compile time*.

# Example: Vectors

- You can iterate over `Vec`s in three different ways:

```
let mut vs = vec![0,1,2,3,4,5,6];

// Borrow immutably
for v in &vs { // Can also write `for v in vs.iter()`
    println!("I'm borrowing {}.", v);
}

// Borrow mutably
for v in &mut vs { // Can also write `for v in vs.iter_mut()`
    *v = *v + 1;
    println!("I'm mutably borrowing {}.", v);
}

// Take ownership of the whole vector
for v in vs { // Can also write `for v in vs.into_iter()`
    println!("I now own {}! AHAHAHAHA!", v);
}

// `vs` is no longer valid
```