Rust Notes: Ownership, Lifetimes, and Borrowing

January 30, 2015

Ownership in C++

What does the local declaration

```
vector<int> counts;
```

mean in C++? Variable `counts` denotes an object in memory. For how long? Until it goes out of scope. So:

```
void countSheep() {
    vector<int> counts; /* counts starts here */

    ...
} /* counts ends here */
```

What if we want to pass `counts` to another function? Well, we can do that:

```
void displayCounts(vector<int>);

void countSheep() {
    vector<int> counts;
    ...
    displayCounts(counts);
    ...
}
```

In C++, vectors are passed by value, which results in two potential problems:

- Semantic problem: a function like `updateCounts(vector<int>)` cannot modify `counts` so that `countSheep` can observe the modification.

- Performance problem: passing a vector by value can mean a lot of copying!

We can fix the first problem by having `updateCounts` return the modified vector:

```
vector<int> updateCounts(vector<int>);

void displayCounts(vector<int>);

void countSheep() {
    vector<int> counts;
    ...
    counts = updateCounts(counts);
    displayCounts(counts);
    ...
}
```

But now we're copying twice. The standard solution is rather than pass the vector by value, we pass it by reference using a pointer:

```
void updateCounts(vector<int>*);

void displayCounts(const vector<int>*);

void countSheep() {
    vector<int> counts;
    ...
    updateCounts(&counts);
    displayCounts(&counts);
    ...
}
```

What exactly does `&counts` mean? It expresses the address of the variable `counts`. So how long is that address valid for? The lifetime of `counts`. This means we can get into trouble, because it's possible for the address of `counts` to *escape*—that is, to outlive the storage associated with `counts`:

```
vector<int>* countSheep() {
    vector<int> counts;
    ...
    updateCounts(&counts);
    ...
    return &counts;
}
```

If the address of `counts` is accessed after `countSheep` returns, that's an unchecked runtime error, and hence the behavior is undefined.

If we want a more flexible lifetime, we can allocate our object on the heap:

```
vector<int>* countSheep() {
    vector<int>* counts = new vector<int>;
    ...
    updateCounts(counts);
    ...
    return counts;
}
```

This makes it *possible* for `countSheep` to legitimately return the pointer, but now we have another problem, because now we need to explicitly recover the storage somewhere. Whose responsibility is it? A common pattern for reasoning about this sort of thing is *ownership*, where the owner is responsible for deleting the object. We need to document ownership protocols as parts of APIs and follow them carefully, because C++ doesn't understand (except for C++11 moves) or enforce them, and if we get them wrong, we get undefined behavior.

In the code above, the caller of `countSheep` takes ownership of `counts`, so it is responsible either to delete it or transfer ownership somewhere else, and so on. How this can mess us up:

- Forget to free: we leak memory and possibly eventually run out. NBD.
- Access after free / double free: *anything* might live at that address after the allocator recycles it, and that means nasal demons.

And we haven't even talked about concurrency yet.

Ownership in Rust

Rust eliminates these problems by ensuring that every object has *exactly one* owner at any given time, which gives up ownership either by transferring it (by passing it to a function, say) or allowing it to go out of scope (which runs its destructor). Furthermore, it tracks the lifetime of each object, and ensures that references to that object, which are *borrowed* from its owner, never outlast the object's lifetime.

If we never hand-off ownership, objects are deterministically destructed when they go out of scope:

```
fn count_sheep() {
    let mut counts = Vec::new();    // counts starts here

    ...
} /* assuming we don't transfer ownership, counts ends here and its
   * destructor is run (deterministically!) */
```

If we do hand-off ownership, we lose access to the object at that point:

```
fn display_counts(Vec<int> someCounts);
fn something_else_with(Vec<int> someCounts);

fn count_sheep() {
    let mut counts = Vec::new();
    ...
    display_counts(counts);         // gives up counts (moves it!)
    ...
    something_else_with(counts);    // type error because we no
    ...                             // longer own `counts`
}
```

Borrowing

If we want to invoke a function with temporary access to an object, we can borrow a reference, mutably or immutably:

```
fn update_counts(&mut Vec<int> someCounts);

fn display_counts(&Vec<int> someCounts);

fn count_cheep() {
    let mut counts = Vec::new();
    ...
    update_counts(&mut counts);
    display_counts(&counts);
    ...
}
```

As in C++ this passes a pointer, but unlike C++ the borrowing is checked to ensure memory safety. The borrower (the callee) has to "return" (statically) the borrowed reference upon return. It can sub-lend it to other functions, but it can't, say, send it over a channel (or do explicit lifetimes make this possible??).

Furthermore, in order to avoid data races, borrowing is restricted to ensure that mutable objects are not aliased. In particular, the rules (cf. reader-writer locks) are:

- A reference to a mutable object needs to be unique, so borrowing mutably suspends access to the original owned object:

```
{
  let mut cref = &mut counts;        // borrow here
  ...
  cref[0] = 5;                       // okay!
  use(counts[0]);                    // type error here
  ...
}                                    // return here
```

- Immutable references need not be unique, but the original owned object needs to be immutable as well so long as the borrowed references exist:

```
{
  let mut cref1 = &counts;           // borrow here
  let mut cref2 = &counts;           // and borrow here
  ...
  use(counts[0]);                    // okay
  counts.push(0);                    // type error
  ...
}
```

Explicit lifetimes

Borrowing as we've seen it so far works provided that borrowed references are downward-only—passed to functions, but never returned. Is there a way to return a borrowed reference? For example, in C++, we might want a function to return a pointer to a member of a data structure:

```
int* last_ref(vector<int>* v) {
    return &v->at(v->size() - 1);
}
```

What is the lifetime of the returned pointed? It's valid until the vector is either freed or *moved*, which can happen if it needs to grow.

Can we do this in Rust? Yes, but we need to track the lifetime of the borrowed reference in such a way as to associate its lifetime with the lifetime of the result. We do this with *lifetime variables*:

```
fn last_ref<'a, T>(vec: &'a mut Vec<T>) -> &'a mut T {
    let len = vec.len();
    assert!(len > 0);
    &mut vec[len - 1]
}
```

The function is polymorphic over lifetimes `'a` (and the element type `T`). It says that it takes a mutable reference to a vector whose lifetime is `'a`, and returns a mutable reference to an element whose lifetime is `'a`. We don't care and never get to know what `'a` is concretely—what we care about is that the result has the same lifetime as the argument. (Note that `'a` is the lifetime of the borrowed reference, not the lifetime of the object from which it was borrowed.)

Explicit lifetimes are also useful when talking about data structures and their contents. We need to make sure that data structures don't outlive their contents. Thus we have two possibilities for elements of containers (for example):

- They are owned by the container.
- They are borrowed, with uniform lifetimes.

The `'static` lifetime

Typically we work with lifetime variables but not concrete lifetimes. There's one exception to this, though: `'static` is the lifetime corresponding to the whole run of the program. It's used for static data such as strings and global constants:

```
static GREETING: &'static str = "Hello, CS4620";
```