`std`: Pointer Types

CIS 198 Lecture 6

Reference: TRPL 5.8

`&T` and `&mut T`

- Your basic, economy-class references.
- Zero runtime cost; all checks are done at compile time.
- Not allowed to outlive their associated lifetime.
  - Can introduce serious lifetime complexity if you're not careful!
- Use these unless you *actually* need something more complicated.

`Box<T>`

- A `Box<T>` is one of Rust's ways of allocating data on the heap.
- A `Box<T>` owns a `T`, so its pointer is unique - it can't be aliased (only referenced).
- `Box`es are automatically freed when they go out of scope.
- Almost the same as unboxed values, but dynamically allocated.
- Create a `Box` with `Box::new()`.

```
let boxed_five = Box::new(5);
```

Box<T>

- Pros:
  - Easiest way to put something on the heap.
  - Zero-cost abstraction for dynamic allocation.
  - Shares typical borrowing and move semantics.
  - Automatic destruction.
- Cons (ish):
  - The `T` is strictly owned by the `Box` - only one owner.
    - This means the particular variable holding the box can't go away until all references are gone - sometimes this won't work out!

Aside: Box Syntax & Patterns

- In homework 2 & 3, you may have noticed patterns like this:

```
let opt_box: Option<Box<i32>> = Some(Box::new(5));

match opt_box {
    Some(boxed) => {
        let unboxed = *boxed;
        println!("Some {}", unboxed);
    }
    None => println!("None :("),
}
```

Aside: Box Syntax & Patterns

- It's currently not possible to destructure the `Box` inside the `Option`. :(
- In Nightly Rust, it is, thanks to `box` syntax!

```
#![feature(box_syntax, box_patterns)]

let opt_box = Some(box 5);

match opt_box {
    Some(box unboxed) => println!("Some {}", unboxed),
    None => println!("None :("),
}
```

- This may change before it reaches Stable.

std::rc::Rc<T>

- Want to share a pointer with your friends? Use an Rc<T>!
- A "**R**eference **C**ounted" pointer.
  - Keeps track of how many aliases exist for the pointer.
- Call clone() on an Rc to get a reference.
  - Increments its reference count.
  - No data gets copied!
- When the ref count drops to 0, the value is freed.
- The T can only be mutated when the reference count is 1 😕.
  - Same as the borrowing rules - there must be only one owner.

```
let mut shared = Rc::new(6);
{
    println!("{:?}", Rc::get_mut(&mut shared)); // ==> Some(6)
}
let mut cloned = shared.clone(); // ==> Another reference to same dat
{
    println!("{:?}", Rc::get_mut(&mut shared)); // ==> None
    println!("{:?}", Rc::get_mut(&mut cloned)); // ==> None
}
```

`std::rc::Weak<T>`

- Reference counting has weaknesses: if a cycle is created:
  - A has an `Rc` to B, B has an `Rc` to A - both have count = 1.
  - They'll never be freed! ~~Eternal imprisonment~~ a memory leak!
- This can be avoided with *weak references*.
  - These don't increment the *strong reference* count.
  - But that means they aren't always valid!
- An `Rc` can be downgraded into a `Weak` using `Rc::downgrade()`.
  - To access it, turn it back into `Rc`: `weak.upgrade() -> Option<Rc<T>>`
  - Nothing else can be done with `Weak` - upgrading prevents the value from becoming invalid mid-use.

Strong Pointers vs. Weak Pointers

- When do you use an `Rc` vs. a `Weak`?
  - Generally, you probably want a strong reference via `Rc`.
- If your ownership semantics need to convey a notion of possible access to data but no ownership, you might want to use a `Weak`.
  - Such a structure would also need to be okay with the `Weak` coming up as `None` when upgraded.
- Any structure with reference cycles may also need `Weak`, to avoid the leak.
  - Note: `Rc` cycles are difficult to create in Rust, because of mutability rules.

`std::rc::Rc<T>`

- Pros:
  - Allows sharing ownership of data.
- Cons:
  - Has a (small) runtime cost.
    - Holds two reference counts (strong and weak).
    - Must update and check reference counts dynamically.
  - Reference cycles can leak memory. This can only be resolved by:
    - Avoiding creating dangling cycles.
    - Garbage collection (which Rust doesn't have).

Cells

- A way to wrap data to allow *interior mutability*.
- An *immutable* reference allows modifying the contained value!
- There are two types of cell: `Cell<T>` and `RefCell<T>`.

```
struct Foo {
    x: Cell<i32>,
    y: RefCell<u32>,
}
```

`std::cell::Cell<T>`

- A wrapper type providing interior mutability for `Copy` types.
    - `Cell<T>`s cannot contain references.
- Get values from a `Cell` with `get()`.
- Update the value inside a `Cell` with `set()`.
    - Can't mutate the `T`, only replace it.
- Generally pretty limited, but safe & cheap.

```rust
let c = Cell::new(10);
c.set(20);
println!("{}", c.get()); // 20
```

`std::cell::Cell<T>`

- Pros:
  - Interior mutability.
  - No runtime cost!
  - Small allocation cost.
- Cons:
  - Very limited - only works on `Copy` types.

std::cell::RefCell<T>

- A wrapper type providing interior mutability for any type.
- Uses dynamic borrow checking rules (performed at runtime).
  - This may cause a panic at runtime.
- Borrow inner data via `borrow()` or `borrow_mut()`.
  - These may panic if the `RefCell` is already borrowed!

```
use std::cell::RefCell;

let refc = RefCell::new(vec![12]);
let mut inner = refc.borrow_mut();
inner.push(24);
println!("{:?}", *inner); // [12, 24]

let inner2 = refc.borrow();
// ==> Panics since refc is already borrow_mut'd!
```

`std::cell::RefCell<T>`

- A common paradigm is putting a `RefCell` inside an `Rc` to allow shared mutability.
- Not thread-safe! `borrow()` et al don't prevent race conditions.
- There is no way (in stable Rust) to check if a borrow will panic before executing it.
  - `borrow_state(&self)` is an unstable way to do this.

`std::cell::RefCell<T>`

- Pros:
  - Interior mutability for any type.
- Cons:
  - Stores an additional borrow state variable.
  - Must check borrow state to dynamically allow borrows.
  - May panic at runtime.
  - Not thread-safe.

`std::cell::Ref<T>` & `RefMut<T>`

- When you invoke `borrow()` on a `RefCell<T>`, you actually get `Ref<T>`, not `&T`.
    - Similarly, `borrow_mut()` gives you a `RefMut<T>`.
- These are pretty simple wrapper over `&T`, but define some extra methods.
    - Sadly, all of them are unstable pending the `cell_extras` feature 😞.

`*const T` & `*mut T`

- C-like raw pointers: they just point... somewhere in memory.
- No ownership rules.
- No lifetime rules.
- Zero-cost abstraction... because there is no abstraction.
- Requires `unsafe` to be dereferenced.
  - May eat your laundry if you're not careful.
- Use these if you're building a low-level structure like `Vec<T>`, but not in typical code.
  - Can be useful for manually avoiding runtime costs.
- We won't get to unsafe Rust for a while, but for now:
  - Unsafe Rust is basically C with Rust syntax.
  - Unsafe means having to manually maintain Rust's assumptions (borrowing, non-nullability, non-undefined memory, etc.)