Dref Trait

Dereference Operator

- The dereference operator is `*`
- This is used to get the value from a pointer
- This operator is used by referneces and smart pointers
- The dereference `Deref` trait allows you to customize the behavior of the dereference operator
- Example:

```
fn main() {
    let x = 5;
    let y = &x;
    let z = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
    assert_eq!(5, *z);
}
```

  - Notice that the immutable reference of x (`y`) is dereferenced the exact same way as a smart pointer
  - There is a difference here, the `Box::new()` takes a copy of `x` and `z` points to the copy and not `x` itself
  - `y` actually points to `x`

Making our own Box Smart Pointer

- In ouer example case we will make a box smart pointer but it won't be fully functional
- It will only have the Dref trait so we can just give the Dref operator some functionality
- Example:

```
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}
```

  - This is a Struct that has a generic `T`
  - Then it has a single field which is a tuple `(T)`
  - And we make a new method for it
- To make it a more like a smart pointer we need to implment the `Dref` trait and its `associate type` (as discussed in the Iterators chapter) and the `deref()` method

- Example:

```
use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}

let y = MyBox::new(5);

assert_eq!(x, *y);
```

  - We have to put the `Deref` trait into scope first
  - Then we implement the `associate type` for `Target` as a generic `T`
  - Then we implement the `deref()` method
    - Remember that the Struct is a Tuple Struct which means it only contains a Tuple as a field
    - So We return an immutable **reference** to the **first** element of the tuple `&self.0`
    - This Element is going to be a `Target` type which means `T`
  - We can see that at the end of this we can use the `*` operator and derefence to the value of our tuple element.
  - Internally Rust does the following whenever we dereference using the `*` operator
    - `assert_eq!(x, *(y.deref()));`
    - It will auto call the `deref()` method to get a reference to the value
    - Then we can use `*` operator to get the value itself
- Note that `Deref` is not returning the value directly, but rather returns a reference in which we have to `dereference` using the `*` operator
- This is because we don't want to transfer ownership of the value directly

Deref Coercion

- Deref Coercion will convert a reference to one type to a reference to another type
- Example:

```
fn hello(name: &str) {
    println!("Hello, {name}!");
}

let m = MyBox::new(String::from("Rust"));
hello(&m);
```

  - Let's analyze this code example
  - First we have a function that expect a string literal or string slice

- Then we make a `MyBox` pointer and put in a `Owned String` into `m`
- And we pass a reference of `&m` which would be `&MyBox<String>` to `&str`
- How does this happen?
  - Well `MyBox` implements the `Deref` trait
  - So Dereferencing `&MyBox<String>` would give us `&String`
  - `String` also implements the `Deref` trait
  - So dereferencing `&String` would give us `&str`
- Rust automatically performs these chain of Dereferencing methods if it sees that the type doesn't match
- If the type exists in the dereferencing chain, then the compiler is smart enough to convert the types **implicitly**
- If you wanted to do the example above by using explicit dereferencing, you would have to do the following
  - `hello(&((*m)[..]));`
  - Which dereferences `m` and converts it into a string slice and then takes the reference of the slice

Deref Mutability and Coercion

- Note that we have used the `Deref` trait which only handles **immutable** references
- But we can also implement the `DerefMut` trait which handles **mutable** references
- Rust Performs `Deref` Coercion in the 3 following cases:
  - From `&T` to `&U` when `T: Deref<Target=U>`
  - From `&mut T` to `&mut U` when `T: DerefMut<Target=U>`
  - From `&mut T` to `&U` when `T: Deref<Target=U>`
- Note it does not work with `&T` to `&mut U`, it only can do mutable -> immutable or same mutability
  - This is because it goes against one of the Borrowing rules
  - Converting a `immutable` reference to `mutable` would require that the initial `&T` must be the only mutable reference to that data which the borrow checker doesn't check for.