NOTES: Rust

Rust

Typing

- **Strong typing** vs **weak typing**
  - Strong typing is when the type of a variable is enforced by the compiler (think TypeScript)
  - Weak typing is when types can automatically convert (think JavaScript)
- **Static typing** vs **dynamic typing**
  - Static typing is when the type of a variable is known at compile time and is based on the syntax
    - Essentially, if you include the type in the variable declaration, it is statically typed
  - Dynamic typing is when the type of a variable is known at runtime and is based on the program behavior
- **Rust** is strongly, statically typed
- **C** is weakly typed, even though it is statically typed, since you can treat a character as an integer
- **Python** is strongly typed, even though it is dynamically typed, because for example, you can't add a string to an int

Error Handling of Type Mismatches

- Rust will generate errors when you try to do operations of variables of different types, i.e. comparing an unisgned int to a signed int
- You can cast types using `as` keyword, and Rust will throw an error if there is an invalid promotion

Bounds Checking

- Rust will throw an error if you try to access an array out of bounds
- However, bounds checking is slow, so you have to be careful how you iterate

Slow:

```rust
let nums = vec![1, 2, 3, 4, 5];
for x in 0...10 {
    println!("{}", nums[x]);
}
```

Fast:

```rust
let nums = vec![1, 2, 3, 4, 5];
for num in &nums {
    println!("{}", num);
}
```

Functional Programming

- There is functional syntactic sugar in Rust

> Ex.
>
> ```
> struct Circle {
>     x: f64,
>     y: f64,
>     radius: f64,
> }
> trait HasArea {
>     fn area(&self) -> f64;
> }
> impl HasArea for Circle {
>     fn area(&self) -> f64 {
>         3.14 * (self.radius > * self.radius)
>     }
> }
> ```

- You can have closures (anonymous functions) in Rust

> Ex.
>
> ```
> let num = 5;
> let plus_num = |x: i32| x + > num;
> print!("{}", plus_num(5));
> ```

- There are functions like `map` and `filter`

> Ex.
>
> ```
> fn fundSumOfOddSquares() {
>     let sum: u32 =
>         (0..) // Range of > all integers, from 0 > up
>         .map(|x| x * x) // > Square each number
>         .take_while(|&x| x < > 1000) // Only take > below 1000
>         .filter(|&x| x % 2 > == 1) // Only take > odd numbers
>         .fold(0, |sum, x| > sum + x); // Sum > them all up
>     println!("{}", sum);
> }
> ```

Variables

- Variables are statically typed, although you can leave out the type and the compiler will infer it
- Variables are immutable by default, but you can make them mutable with the `mut` keyword

```
fn main() {
    let x = 5; // Inferred to be int 32 (i32)
    let y: u32 = 6; // Explicit typing as an unsigned int 32
    let mut z: f32 = 3.14; // Mutable variable
    println!("x is {}", x); // ! means the function is a macro
}
```

Constants

- `const` keyword is most like `#define` in C, where it does a find-and-replace

```
const PI: f32 = 3.14;
```

- `&'static` is like const, but it is instead a reference to memory. It is never inlined. It is used for string literals and other static data

```
let x: &'static str = "hello";
```

> **Note: Shadowing**
>
> You can shadow variables in Rust, even within the same scope.
>
> ```
> let x = get_num_from_user();
> let x = x.parse().unwrap(); // x is an entirely new variable
> ```

Strings

- Strings defined by `" "` are string literals, and cannot be mutated
- You can use the `String` type to create a mutable string

```
let mut x = String::from("hello");
x.push_str(", world!");
```

- If you know you'll be pushing a large number of bytes, you can pre-reserve memory

```
let mut x = String::with_capacity(10);
```

- Rust supports UTF-8, which means you cannot index characters (`str[0]`) because of multi-byte characters 🇺🇸
- You can use a string slice instead

```
let x = String::from("hello");
println!("x is {}", &x[0..1]);
// [x..y], x inclusive, y exclusive
```

Copying vs. Borrowing

Copying

```
let x = 5;
let y = x;
// This compiles...
let s1 = String::from("hello");
let s2 = s1;
println!("{}", s1);
println!("{}", s2);
// ...but this doesn't! "Borrow of moved value"
```

- For data types that are easy to copy (like `i32`), Rust will copy the value
- For more complex, heap-stored types, you have to explicitly clone (deep-copy) the value

```
let s1 = String::from("hello");
let s2 = s1.clone();
println!("{}", s1);
println!("{}", s2);
```

Borrowing

- Borrowed values are similar to pointers in C, but:
    - There are no null pointers
    - There are restrictions on simultaneous borrows
- Borrows are immutable by default, even if the original value is mutable

```
let mut x = String::from("hello");
let y = &x; // Borrow
let z = &x;
// This works...

let mut x = String::from("hello");
let y = &mut x; // Borrow
let z = &x;
// This does not work
```

- You cannot have a mutable borrow and an immutable borrow at the same time (to prevent data races)

Moving

```
fn main() {
    let x = 5; // Immutable
    foo(x);
}
fn foo(mut y: i32) {
    y = 6; // This works; y is a copy of x
}
```

In the following example, `x` is moved to `foo` and is no longer valid in `main`:

```
fn main() {
    let x = String::from("hello"); // Mutable
    foo(x);
    println!("{}", x); // This does not work; x has been moved
}
```

In this example, `x` is borrowed by `foo` and is still valid in `main`:

```
fn main() {
    let x = String::from("hello"); // Mutable
    foo(&x);
    println!("{}", x);
}
fn foo(x: &String) {
    println!("x is {}", x);
}
```

You can even mutate borrowed values it you use the `mut` keyword:

```
fn main() {
    let mut x = String::from("hello"); // Mutable
    foo(&mut x);
    println!("{}", x);
}
fn foo(x: &mut String) {
    println!("x is {}", x);
}
```

Functions

> **Note: Returning**

> You can return a value by using it as a statement, as long as you don't put a semicolon afterwards.
>
> ```
> fn add(x: i32, y: i32) -> i32 {
>     x + y
> }
> ```

Structs

- Rust structs are split into structure and functionality.

```
struct Car {
    owner: String,
    model: String,
    year: u32,
}
impl Car {
    fn drive() { /* ... */ }
}
```

Ownership

- Every value has a single, statically-known owner
- References (similar to pointers) to owned values have a limited lifetime
- All references to values are known statically

Ownership Transfer

- You can transfer ownership of a variable to another function

```
fn doAnything(v: Vec<i32>) {
    // This function now owns v
    // After this function scope ends, v will be freed
}
fn main() {
    let v = vec![1, 2, 3];
    doAnything(v);
    // v is no longer valid here
}
```

Borrowing

- You can borrow a reference to a value, but you can't modify the value
- Owner cannot free, move, or mutate its underlying value while it is borrowed
- This makes it so in multi-threaded programs, there are no race condition problems
- To mark a variable as borrowed, use the `&` symbol

```rust
fn foo(a: &i32, b: &mut i32) {
    // Do stuff with a (immutable) and b (mutable)
    // a and b are not owned by foo, but are borrowed
}
fn main() {
    let x = 3;
    let mut y = 5;
    foo(&x, &mut y);
    // We can still use x and y here since we still own them
}
```

Mutability

- By default, variables are immutable (like JS `const`)
- To make a variable mutable, use the `mut` keyword

Error Handling and Null Values

- `Option<T>` can be used if you want to return a value that may be `None` (like `?` in TS)

```rust
let mut x: Option<i32> = None;
x = Some(5);
```

- Many functions that can fail return a type with an optional error field (`Result<T>`). To extract the value (and crash if there is an error), use the `unwrap` method

```rust
let f = File::open("hello.txt").unwrap();
```

- You can choose to throw a nice error with the `expect` function:

```rust
let f = File::open("hello.txt").expect("Failed to open hello.txt");
```

- Since `Result<T>` is an enum of either OK or Error, you can use a `match` statement:

```rust
let f = match File::open("hello.txt") {
    Ok(file) => file,
    Err(error) => panic!("Failed to open hello.txt: {}", error),
};
```