Chapter 1

Rust

Forgive me, I am a tad Rusty

Rustaceans

First, let me begin by saying that these notes are only a condensed and high level account of what is found in The Rust Book: https://doc.rust-lang.org/book/. I would definitely recommend you take a look there as nothing I could do would be as good. The topics covered per semester change rapidly depending on time and who is teaching. Historically we focus on what makes Rust unique (Ownership, Lifetimes and Smart pointers, chapters 4, 10, and 15 respectively).

This is a programming language chapter so it has two (2) main things: talk about some properties that the Rust programming language has and the syntax the language has. If you want to code along, all you need is a working version of Rust and a text editor. You can check to see if you have Rust installed by running `rustc -version`. At the time of writing, I am using Ruby 1.65.0.

Unlike other programming chapters, there will be a lot about the properties of the language itself, since Rust has a few new things not found in other languages. Before reading this chapter, please refer to the Garbage Collection Notes.

1.1 Introduction

Rust was made around 2016 from the folks over at Mozilla (the firefox[1] people). They built the first Rust Compiler in Ocaml, which means they really liked that language, so you may see some Ocaml-ness in the Rust Language. Rust's goal is to be a **safe** language, but at the same time, maintain the speed and find grain control of the machine that C gives you. Before we get into all that though, let's write our very first Rust program:

```rust
// hello_world.rs
fn main() {
    println!("Hello, world!");
}
```

Despite this being very simple, we've already learned several things. We can also notice that we are going back to more "traditional" style languages.

- Single-line comments are started with the backslash
- Semicolons!
- `println!` is used to print things out to stdout
- functions use parenthesis (weird we need to say this)
- Rust file entension is `.rs`

[1]Firefox over Chrome-based browsers, always. Ad-blockers ftw

- Strings exist in the language (most languages have them, but some do not)
- Need a `main` function.

Can you think of more?

Now to compile our program we can just use

```
rustc hello_world.rs
./hello_world
```

Congrats, you have just made your first program in Rust! There are some things to note however:

- Rust is a compiled Language
- `rustc` is the rustc compiler (some compilers like to take the name of the language and add 'c' to the end: javac, ocamlc, rustc).
- If you run an `ls` you will notice that the executable `hello_world` was created. There were no other files made (like headers or object files).
- `rustc` is wrapped in a nice program called `cargo`. `Cargo` will allow you to create, compile, run, and test your Rust programs without much overhead. We use `cargo` to help manage your projects.

1.2 Memory and Security

Again, I would recommend that you go read the Garbage Collection Notes before this section.

C is considered a low level language because it doesn't really abstract things. We could think of C as an API for assembly. This makes C a very fast language because it doesn't have to deal much with garbage collection, or type checking during runtime. This also makes C a very unsafe language, because it is up to the programmer to manage memory and check types. Higher level languages abstract this away making for safer languages, but also makes the languages slower by comparison. Rust wants the best of both worlds here. It wants to be safe, but also be fast. For Rust to do this, it would need to forgo some of these costly overheads like garbage collection and type checking at runtime and instead drop them completely OR lean more on the compiler. Forgoing these options would just be C, so Rust is designed around how it's compiler can do a lot of work to get fast and safe programs.

1.2.1 Safety

What is safety? Consider the following:

- A server that sends out restricted information like bank passwords to anyone is unsafe.
- An air traffic control application that can be taken down via a DOS (denial of service) attack leading to plane crashes is unsafe.
- A grades server where you can overwrite everyone's grades with an F is unsafe.

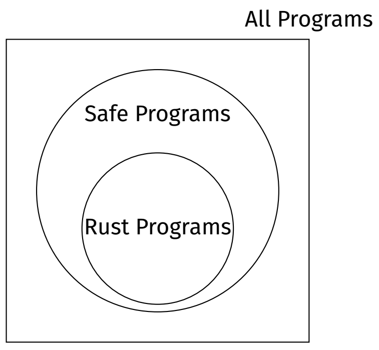
These are examples of unsafe programs and they all have at least one thing in common: a program having unwanted behavior. Thus safety could be thought of as a part of correctness of a program. We made proofs about correctness of a program when we did Operational Semantics. However, OpSem is not the only way to describe meaning of a program, and it cannot capture or measure all the semantics or behavior of a program.

In the above examples and what Rust hopes to mitigate, is the insecurity of memory. Human insecurity (like falling for a phishing attack, or being tortured, not insecurity like self esteem (however this can also be leveraged)), is not something Rust or any language can really prevent. Memory safety has a few types of incorrect behavior, for example:

- Giving read access when it shouldn't (sending plaintext passwords)
- Gives write access when it shouldn't (overwriting people's grades)

- Denies read access when it shouldn't (DOS)

There are others but these are 3 big ones. If Rust can prevent unwanted behavior, then it has succeed in it's job. However, making sure a program is secure is very hard. Thus Rust will take a very conservative approach: It will refuse to compile programs it is unsure if they are safe or not. This is sometimes known as whitelisting: denying everything except what you know. This is opposed to blacklisting: allowing everything except what you don't want. Whitelisting is more secure. What this means: there are a set of programs that are safe, but Rust's compiler cannot verify it so Rust will not compile your safe program. Visually:



1.2.2 Stack and Safety

As we saw in the Garbage Collection Notes, memory safety comes down the idea that memory is not being managed correctly. We also said the garbage collector makes languages slow. So we need some fast way of determining when and how to manage memory. To do so, let's look at another place in memory: the stack.

When we need to allocate memory on the stack, we simply push the value onto the stack. When we need to deallocate memory, we simply pop it off. In order to allocate something on the stack we need to know 2 major things. The first is more obvious: we need to know how much memory we need to allocate. This is why things that have a fixed size link an `int` will be stored on the stack, but things of changing size are stored on the heap. The second is less obvious: we need to know how long that item needs to live in memory. Consider function calls. The parameters and local variables of the function have a scope. We know when the item enters memory (when the function is called) and we know when it needs to leave (when the function returns). This is why when we have items that need to exist for longer than a certain scope need to return a pointer, rather than the value.

Both of these points allow for the automatic memory management of the stack: we know how much to `malloc` since size is known, and we know exactly when to `free` due to knowing when the scope ends. Now items on the stack are not `malloc`'d or `free`'d but the point still stands, to automatically mangage memeory, we need to know these 2 things. Rust will introduce concepts built into the design of the language to allow for this behavior.

1.3 Statements vs Expressions and Codeblocks

Before we can begin to talk about how rust manages memory without a garbage collector, we need to know some syntax for code examples later used.

To begin, in Rust, there is a explicit distinction between expressions and statements which we need to consider. As we saw in OCaml, an expression is something that evaluates down to a value. A statement on the other hand does not evaluate to a value but may perform some sort of action.

For example, something like `let x = 5` is a statement, not an expression. It does not evaluate to a value. You could not say something like `3 + (let x = 5)`. However things like 3 or 1+2 are expressions, and they evaluate to the value of 3. Expressions can be part of other expressions: `3 + (2 + 3)`, and they can be part of expressions: `let x = 3 = 4`.

This becomes important because of the idea of a codeblock which Rust allows. Rust allows for this idea of a codeblock which is a collection of statements followed by zero or more expressions surrounded by curly braces. For example:

```
1 let x = 3;
2 {
3   let y = 5;
4   let z = 7;
5   println!("{}",y+z);
6 };
```

Lines 2-6 are the codeblock. This codeblock has three statements and zero expressions. Statements are ended with a semicolon while expressions are not.

```
1 let x = 3;
2 {
3   1 + 2;
4   3 + 5
5 }
```

Lines 2-5 are the codeblock. This codeblock has one statement: `1 + 2`, and one expression: `3 + 5`. The codeblock itself is treated as an expression.

Again, codeblocks are zero or more statements followed by at most one expression. If we were to describe the structure with some regex syntax, we could say that a codeblock looks like

`{ stmt;* expr? }`

Some more example of codeblocks:

```
1 {
2     1 + 2;
3     let a = 4 - 3;
4     a
5 }
```

two statements, one expression

```
1 {
2     1 + 2;
3     let a = 4 - 3;
4     a;
5 }
```

three statements, zero expressions

```
1 {
2     1 - 3
3 }
```

Zero Statements, one expression

Codeblocks are expressions themselves which means we can capture the value of the last statement in a codeblock:

```
1 let a = {
2   let b = 3;
3   let c = 4;
4   b + c
5 };
6 println!("{}",a);
```

Here we can see a codeblock on lines 2-4 which has two statements and one expression, where the result of the expression is being bound to the a variable. This however raises the question: what happens when there is no expression?

```
1 let a = {let b = 3; b + 5;}
2 println!("{}",a);
```

In this case, Rust will use the default return value, which is called `unit`. Like Ocaml, `unit` is an empty tuple: `()`. Thus the above code will fail (since Rust does not not know how to print the `unit` type). We could instead however do the following:

```
1 let a = {let b = 3; b + 5;}
2 if a == (){
3     println!("a is unit type");
4 }
```

This is a perfect segue to using codeblocks as expressions in the `if` expression.

1.4 If Expression

The `if` expression is exactly that: an expression. Thus it evaluates to a value. Much like Ocaml, each branch of the `if` expression must return the same type. You may be thinking, but what about that very last example in the above section? We will get to that.

The `if` expression takes the form of `if guard_expr {true_block} else {false_block}`. The `true_block` and `false_block` are both codeblocks whereas the `guard_expr` is an expression (which could also be a codeblock). For example:

```
if true {false} else {true}
// analogous to Ocaml’s expression: if true then false esle true

if false {
    let a = 3;
    let b = 4;
    println!("{} is not {}",a,b)
} else {
    println!("I am false")
}
// First no such thing as multiline comment
// Second: this shows the true codeblock having two statements and 1 expression
//         whereas the false codeblock has 1 expression
//         (technically println! is a macro that exapnds to an expression)

if {let a = 1;
    let b = 3;
    a < b}
    {0}
    else
    {1}
// example where the guard is a codeblock (because a codeblock is an expression)
// the true and false block are just one expresssion
// analagous to Ocaml’s: if
//                          (let a = 1 in
//                          let b = 3 in
//                          a < b)
//                          then 0 else 1
```

In each of these examples, the type of true block is the same as the type of the false block. Breaking this type check will fail to compile:

```
if true {3} else {"hello"}

if true {3}
```

This second example is a case of the `if` expression without an else block that we saw as the last example in the previous section. That example does, work, but this one does not. Why?

We know that a codeblock with no expression will evaluate to the `unit` type by default. The same is the case here. When the `else` block is left out, then the default return type of the else branch is type `unit`. Thus, if we want to leave out the `else` block and still be able to compile, we need to make sure the `true_block` will also evaluate to type `unit`. This can be done by in a few ways: we can make the last expression a statement by adding a semicolon, or we use an expression that returns type `unit`.

```
if true {3;}
// here the true block has one statement and zero expressions
// and hence evaluates to unit

if true {3; println!("println! is an function that evaluates to unit")}
// println! is an function that evaluates to unit

if true {3; ()}
// We could explictly return unit
```

1.5 A bit of data types and Functions

Functions in Rust are expressions, they evaluate (return) a value (which includes unit). Functions are a named collection of commands which are dependent on an input (an empty input is included here). They can also be unnamed (anonymous functions) which Rust call closures. To define a function however we first need to talk about data types.

1.5.1 Data Types

Built-in data in Rust has type which can be thought of as scalar (flat) or compound. These are pretentious words that describe types based on what is needed to describe the data. Scalar types include things like numbers, booleans, and characters. Since Rust wants to have some of the advantages of C which include fine grain control of memory, Rust breaks it's numeric types of integers and floats into subtypes. That is to say, there is no `integer` type, but rather "integer of 8 bits", "integer of 16 bits", "integer of 32 bits", etc. See below the table of data types:

Integers		
Size	Signed	Unsigned
8 bits	i8	u8
16 bits	i16	u16
32 bits	i32	u32
64 bits	i64	u64
128 bits	i128	u128
Machine Dependent	isize	usize

Machine dependent sizes are dependent on the hardware since different machines have different architectures (32-bit vs 64-bit for example). They are typically the size of a pointer. For Floats, there are just `f32` and `f64`.

Last of the scalar types are `bool` and `char`. It is important to note, that characters are 4 bytes long so they include more than ascii, including utf-8 characters (which includes emoji and characters in other languages).

Compound types on the other hand, are pieces of data that need at least 2 values to define its type. In Rust, the built in compound types are tuples and arrays. Like OCaml, tuples in Rust are fixed size, hetergenous and defined by the types it holds. A `(u8,u16)` tuple is different from a `(u16,u8)` tuple and both are different from a `(u8,u16,u32)` tuple. Again, the empty tuple is called `unit` and used to say it the value is nothing meaningful.

Arrays on the other hand are different from arrays in other languages or lists in OCaml. They must be both homogenous and a fixed length. An array's type is defined by the type of data it holds and its length.

```
let a = [1,2,3,4]
// this has type [i32;4]. Rust will default to i32 for numbers in that range
let a = [3;5]
// this tells Rust to make an arrayof size 5 with each element being the value 3
// this has type [i32;5]
```

When describing a type of a value or variable, the colon notation is used.

```
3: i32
```

```
'h': char
true: bool
[1,2,3]: [i32;3]
(1,1.0,false): (i32, f64, bool) //default float is f64
```

1.5.2 Functions

Now that we know some data type notation, we can now talk about functions! Functions can have zero or more parameters, and will return a single value (could be `unit`). In Rust we need to annotate types of our inputs and output if they are not `unit`.

```
fn main(){
    println!("Hello")
}
// this function takes in input and returns unit. Notice I don't need a semicolon here

fn this(x: i32){
    println!("{}", x + 4);
}
// this function takes in a single argument of type i32 and returns unit. Do not need to
    denote the return type but do need to specify the input type

fn that(x:u8, y:u8) -> bool{
    x > y
}
// this function takes in two arguments and returns a boolean. Need to denote the return and
     input types since they are not unit

fn the_other_thing() -> char{
    'a'
}
// this function takes in zero arguments and returns a char. Need to denote the return since
     it is not unit
```

1.5.3 Closures

We can make anonymous functions called closures. Their notation looks like a Ruby Codeblock.

```
|x| x + 1
// parameters are surrounded by |
// body of closure is an expression

|x, y| x + y
|x, y| {let z = x + y; z}
// expression can be a codeblock
|x:i32| -> i32 {x}
```

There are a few important things to note

- Much like Python and OCaml, we can bind a closure to a variable (`let x = |y| y`).
- Rust will perform type inference on a closure so type annotations are not needed (but they are prefered!)
- when using a return type annotation, you need to put the expression in a codeblock (for the parser)
- Closures cannot use generics. So `|x| x` cannot be called on both a int and string. It will default to the type of the first call

1.6 Ownership

We are now finally ready to start talking about the thing that makes Rust safe: ownership. Ownership describes a set of rules to determine when a value is going to be drop'd (Rust's equivalent to free[2]). It can also influence who has read/write access. The rules of ownership is as follows:

- Every value in Rust has an Owner
- There can only be one owner at a time
- When the owner goes out of scope, the value will be dropped

The last rule here is very important: it tells you when a value should be free'd which was one of the rules to help bring automatic memory management to Rust without a garbage collector. What exactly is a an owner though? An owner is the one responsible for freeing and is the one who can access the value. This can be better seen here:

```
1 let x = 3;
2 {
3   let a = 4;
4 }
5 println!("{}",x);
```

Here, 3 is a value and x is the owner. The owner goes out of scope after line 5 so it is dropped then. 4 is also a value, and it's owner is a. a goes out of scope at line 4 so it is dropped then. However, this is ultimately a tad misleading and doesn't really showcase anything since they are stored on the stack to begin with. They don't really have complicated interactions with ownership. Let us talk about values stored on the heap. For example: a String.

The String data type is part of the standard library, not something built-in to Rust and can change size, hence it must be stored on the heap. String literals however are hardcoded in memory and are technically owned by the Rust program. Regardless, let's take a look at this heap allocated String and how ownership affects it.

```
1 let s = String::from("Hello");
2 //this is calling the 'from' function from the 'String' library. Namespacing
```

Here there is the value "hello" stored on the heap and its owner is 's'. This is uninteresting so let's see something fun:

```
1 let s = String::from("Hello");
2 let x = s;
3 println!("{}",s); //fails to compile
```

This fails to compile. This is because of the second rule of ownership. Only 1 owner is allowed at a time. When we execute line 2, ownership of "Hello" is **moved** to the variable 'x'. Think of it this way: I own a jar of dirt. If I give it to you, then I cannot use the jar of dirt again since you now have ownership of it. Thus, if we wanted to print "Hello", we would have to do the following:

```
1 let s = String::from("Hello");
2 let x = s;
3 println!("{}",x);
```

Why does Rust make you do this? Consider the following:

```
1 char* s = malloc(sizeof(char) * 6);
2 char* x = s;
```

When we no longer need that char pointer, who should free that segment of memory? 'x' or 's'? We don't want them both to free, that would be a double free. Rust gets around this by making sure it knows who exactly can free and knows when. This also prevents the use after free since 's' becomes invalid once ownership is moved to 'x' (so s cannot use that memory address), and the value is dropped once 'x' goes out of scope (hence it becomes impossible for x to use that memory address due to scoping constraints).

[2]Drop is actually a function that is called when 'freeing' so it's more like a deconstructor in C++

This process of passing ownership is called moving. The owner of a value can be moved around using let bindings, function calls, closure creation, etc. With closures you need to explicitly tell rust to move ownership and I would say you probably won't come across this in this course. For let bindings and function calls, let's consider the following:

```
1 let x = String::from("Hello"); //x owns value
2 let y = x; // y now owns, x becomes invalid
3 let z = y; // z now owns, y becomes invalid
4 let a = String::from("Hello"); // a now owns it's own copy of hello
5 let b = a; // b now owns the second hello, a is invalid
6 println!("{} is {}", z, b); // this is fine
```

In this scenario, 'x' initially points to a value of "Hello". After passing ownership to 'y' which then passes ownership to 'z', a new segment of memory is allocated with the value of "Hello" and 'a' becomes the owner of this second instance of "Hello". 'z' still maintains ownership of the first instance of "Hello" since it's a separate segment of memory. Thus, this code compiles because the rules are followed, each value has its own owner.

We can see this more in action when we add codeblocks, which change the scope of variables.

```
1 let x = String::from("Hello"); //x owns value
2 {
3   let y = x;
4   println!("y is the owner of {}",y);
5 };
6 println!("x cannot be used here");
```

in this case, 'x' is the inital owner of "Hello", but then 'y' takes ownership in line 3. When the owner 'y' goes out of scope at line 5, the value is dropped. Ownership is **NOT** passed back to 'x'. Thus, we cannot use 'x' in line 6.

For functions, we pass ownership into functions via parameters and pass ownership back using return values. Consider:

```
1 fn main(){
2   let x = String::from("Hello");
3   let y = this(x);
4   println!("y is the owner of {}",y);
5 }
6
7 fn this(a:String) -> String{
8   a
9 }
```

In this case, if we consider a code trace of the above, we could say that the lines of execution would be something like

- Line 1: main is called
- Line 2: 'x' becomes owner of the newly made "Hello" String
- Line 3a: the `this` function is called in line 3
- Line 7: function is called and stack frame is made
- Line 8: 'a' is evaluated and returned
- Line 9: stack frame is popped off
- Line 3b: return value from `this` is bound to 'y'
- Line 4: print line is run
- Line 5: main is returned, program ends

This is a rough estimation of the order in which lines are executed. We can trace this order to figure out how ownership of "Hello" is passed around.

- Line 2: 'x' is the owner of "Hello"
- Line 3a/7: ownership of "Hello" is moved from 'x' to 'a' during this function call. 'x' becomes invalid
- Line 8,9,3b: ownership of "Hello" is moved from 'a' to the variable capturing the return value; 'y'. 'a' becomes invalid but is dropped anyway because the function ends.
- Line 4: Since 'y' is the owner, this is valid
- Line 5: the owner 'y' goes out of scope so "Hello" is freed now

This is important to note because the following would not work because "Hello" gets dropped when the function ends.

```
1  fn main(){
2    let x = String::from("Hello");
3    that(x);
4    println!("{}",x); // will not compile
5  }
6
7  fn that(a:String) -> String{
8    a
9  }
```

Here since 'a' is the owner and goes out of scope with nothing capturing the return value, "Hello" is dropped around line 9, after line 3 executes.

1.6.1 No Heap Values, Copy trait

This whole process of ownership and passing ownership around really only affects values on the heap. Values on the stack are just immediately copied. See the following:

```
1  let x = 3;
2  let y = x;
3  println!("{} == {}", x,y);
4
5  let a = String::from("Hello");
6  let b = a;
```

In the above example, line 3 is valid since many built in data types support the `Copy` Trait. A `Trait` is analagous to an interface in Java. More on these later. For now, just know that instead of 'y' getting ownership of the value that 'x' owns, a copy of 'x' is made and given to 'y'. Thus, 'x' and 'y' are both owners of their own instance of '3'. This is analogous to a previous example where we had two variables that pointed to their own segment in memory even if the value was the same.

Any struct (object) in Rust that has the `Copy trait` (implements the copy interface) will instead copy the value (using a memcpy) so there will be two instances of the value, each with their own owner. Copy is not overloadable and happens implicitly, if you wanted finer control of how your data is copied (like when making your own data structure), use the `clone` trait). More on this later

1.7 Borrowing

In most cases it makes no sense to pass ownership to a function, especially if you want ownership back. Consider the following:

```
fn main(){
  let x = String::from("hello");
  let(y,z) = get_len(x);
  println!("{} has len {}",z,y);
}

fn get_len(a:String)-> (usize,String){
  let l = a.len();
  (l,a)
}
```

In this case, if I wanted to get the length of a value, I don't need to give full ownership of the String, because then I have to then get ownership back. It should be sufficient to *borrow* the data. That is, if I have a jar of dirt, you can take a look at it, but it's still mine. I could even lend it to you to look at and but afterwards, I want it back. Rust allows this occur with the idea of references and borrowing.

However if we are lending things out, we need to make sure we maintain the ability to know when to drop things, and know what can use pieces of data so there isn't insecurities. To help us out, there are two(ish) rules of references:

- Rule 1: Every reference must be valid (we need to say this. We will see later)
- One but not both (XOR) of the following must will be true:
  - Rule 2a: You can have any number of immutable references
  - Rule 2b: You can have one mutable reference

Before we begin, we need to talk about mutability. In Rust, every variable is immutable. That means once you bind a value to a variable, you cannot change that value. A let binding is a new binding every single time, so like OCaml, the variable is shadowing any previously made variable of the same name. To make a variable mutable, you can use the `mut` keyword.

```
let x = 3;
let x = 4; // new binding, shadows the previous line

let mut y = 5;
y = 6;
```

The above is an example of the shadowing and `mut` keyword use. Just because you use the `mut` keyword, does not mean you have to use it mutably. This will become important when talking about references.

So back to references. A reference is like a pointer that has certain access rights. Ownership is like a special type of reference that will invalidate any previous references if possible. Otherwise we can make references that don't take ownership and have certain access rights to the value in question. Let's see what this means:

```
1 let x = String::from("Hello");
2 {
3     let y = &x; // & tells rust to have y borrow from x, not take ownership
4     println!("I can use {} and {}",x,y)
5 };
6 println!("I can still use {}",x);
```

In this case, in Line 1, 'x' is the owner of "Hello". In line 3, 'y' borrows from 'x', making rule 2a be true: there are some number (two in this case, x and y) of immutable references to "Hello". When we have immutable references, we have read access to that piece of data so we can use both 'x' and 'y' to read "Hello". When 'y' goes out of scope on line 5, the value of "Hello" is not dropped since 'y' was not the owner, allowing us to still use 'x' on line 6. Using this idea, we can better make use of our `get_len` function:

```
fn main(){
  let x = String::from("hello");
3   let y = get_len(&x);
4   println!("{} has len {}",x,y);
5 }
6
7 fn get_len(a:&String)-> usize{
8   a.len()
9 }
```

In this case, the variable 'a' in `get_len` does not take ownership of "Hello" but instead receives a read-only reference to it. Thus, we can still use 'x' on line 4 even when the function that uses the reference ends and 'a' goes out of scope.

So back to the rules. Rule 1 is pretty straightforward, we cannot have dangling pointers. Rule 1 prevents this from occuring. Consider the dangling reference in the C code:

```
int* x = this();

int* this(){
    int i = 5;
    return &i;
}
```

in this case, 'x' is dangling since the place it points to was dropped when the `this` function ended. This is prevented in Rust by ensuring that all references are valid. Rust will not compile the following:

```
fn main(){
  let s = that();
}
fn that()->&String{
    let s = String::from("Hello");
    &s
}
```

This also doesn't really make sense in Rust, just give ownership, no need to pass a reference.

In regards to rule 2: it is important to note, that we can have any number of immutable references at a time (as long as there are no mutable references):

```
let x = String::from("Hello");
let y = &x;
let z = &x;
```

it is also important to note, that Rust will try its best to dereference automatically through deref coercion. This is important when talking about smart pointers (we will get to this), but for now it means we can do the following:

```
1 let x = String::from("Hello");
2 let y = &x;
3 let z = &x;
4 let a = &z;
5 let b = z;
6 println!("I can use all these values to read {},{},{},{},{}",x,y,z,a,b);
```

Here, rust is automatically dereferencing lines 4 and 5 so they point to the "Hello" value in memory. This has to deal with the `Deref` trait for those interested.

Back on track, what about mutable references? When we have a mutable variable, we can make either immutable references or mutable references to it. We cannot make a mutable references to an initially immutable variable. We also have to make sure we keep rule 2 in mind as we do this. Consider:

```
let mut x = String::from("Hello");
x.push_str(" World"); // concats " World" to "Hello"
{
```

```
4     let y = &x;
5     println!("{} and {} can only read, no write",x,y);
6 };
7 x.push_str("!");
8 println!("{} is still valid",x);
```

At line 1: ’x’ is the mutable owner of the "Hello" value so it can read and write to "Hello". We can see it write in line 2 with the `push_str` function. Then at line 4, an immutable reference is created, which makes us take a look at rule 2. We cannot have both one immutable and one mutable reference to a value so what happens? In this case, Rust makes the ’x’ mutable reference immutable (revokes write access) for the entire duration of ’y’s lifetime (not scope!). This means, we cannot mutate the now "Hello World" string until ’y’s lifetime ends (at line 5). Afterwards, ’x’ gains write access again and is allowed to mutate "Hello World" to "Hello World!" in line 7.

The purpose behind this rule is to prevent dangling pointers, and also prevent data races. Data races are a good place for undesired behaviour to occur, and temporal attacks, while difficult to pull off, can be devastating. To mitagate data races, the rules of references come into play. Let us rephrase rule 2 into read and write access terms:

- You can have many readers to a piece of data and no writers XOR
- You can have 1 writer and no readers to a piece of data at a time.

This means nothing could read a piece of data as it is being updated, and no more than one thing could write at a time. Thus, many data races are mitigated in Rust which make for secure programs. There are ways to do concurrency correctly with `Mutex` which we will get to (//TODO).

1.7.1 A thing on `mut`

The `mut` keyword has multiple uses and it can get confusing when we talk about mutable variables and mutable references. A variable can be mutable, and you can borrow a value mutably. These are different.

When a variable is bound to a *value*, then we can talk about the variable’s ability to modify the value.

```
let mut x = 42;               // x is mutable and can change its value.
println!("{}",x);             // 42
x = 32;
println!("{}",x);             // 32

let y = 84;                   // y is immutable and cannot change its value
let mut z = y;                // z is mutable and can change its value
                              // additionally y is copied to z here
z = 42;
println!("{},{}",y,z);        // 84,42

let a = String::from("hello"); // a is immutable and cannot change what it is bound to
let mut b = a;                // b is mutable and can change value
                              // additionally a is moved to b here
println!("{}",b);             // hello
b.push_str(" world");
println!("{}",b);             // hello world
```

Things get tricky when a variable is bound to a reference. In this case, we can say the variable is mutable, or we can say the reference the variable is bound to is mutable. Or we can say both. Or neither. Confusing. This impacts the variable’s ability to mutate what it is bound to, or the data it points to.

```
let x = String::from("hello");     // immutable reference from x to hello
let y = &x;                        // immutable refenence from y to hello
println!("{},{}",x,y);             // hello,hello

let mut a = String::from("hello"); // mutable reference from a to hello
```

```
println!("{}",a);                  // hello
a.push_str(" world");              // a can mutate what it points to
println!("{}",a);                  // hello world

let mut b = String::from("hello"); // mutable reference from b to hello
let c = String::from("bye");       // mutable reference from c to bye
println!("{}",b);                  // hello
let mut d = &b;                    // the variable d is mutable, but borrows immutably
println!("{}",d);                  // hello
//d.push_str("world");             // d cannot change the value it points to
d = &c;                            // but d can change the value it is bound to
//d = 42;                          // as long as its the same type
println!("{}",d);                  // bye


let mut e = String::from("hello"); // mutable reference from d to hello
println!("{}",e);                  // hello
let f = &mut e;                    // the variable f is immutable, but borrows mutably
//println!("{}",e);                // this makes e invalid during f's lifetime
f.push_str(" world");              // f can change the value it points to
//f = &mut String::from("bye");    // but f cannot change the value it is bound to
                                   // this also doesn't work because &mut String::from
                                       doesnt make sense
println!("{}",f);                  // hello world; also f's lifetime ends here
println!("{}",e);                  // hello world; allowing e to valid again

let mut g = String::from("hello"); // mutable reference from b to hello
let mut h = String::from("bye");   // mutable reference from z to bye
let mut i = &mut g;                // the variable i is mutable and also borrows mutably
//println!("{}",g);                // this makes g invalid during i's lifetime
i.push_str(" world");              // i can change the value it points to
println!("{}",i);                  // hello world
i = &mut h;                        // i can also change the value it is bound to
                                   // making there only be one mutable ref to "hello"
                                   // meaning g is now valid again
println!("{},{}",i,g);             // bye,hello world
```

1.8 Lifetimes

As we just saw, I made a distinction about lifetimes and scope. Rust does the same thing. Scope in Rust is no different than any other language, it determines where a variable could be used. Rust however goes a step further and makes a distinction about a reference's lifetime and a variable's scope. In many cases, the lifetime of the reference and scope of a variable are the same, but there are also many cases when they are different. Consider:

```
1  { let x = String::from("hello");
2    { let y = String::from("hi");
3      println!("{}",y);
4    }
5    println!("Hello World");
6    let z = String::from("World");
7  }
```

In this example, 'x's scope is lines 1-7, 'y's scope is lines 2-4, and 'z's scope is line 6-7. However, the reference that 'x' points to is only used on line 1, making it's lifetime Line 1. The reference 'y' is bound to has a lifetime that is the same as 'y's scope: it is used first on line 2 and last used on line 3/4, and the reference bound to 'z's lifetime is also the same as 'z's scope: line 6/7.

Let's see this in terms of mutable and immutable references:

```
1 let mut x = String::from("Hello");
2 let y = &x;
3 println!("{}",y);
4 x.push_str(" World");
```

In this case: 'x' starts off as a mutable reference [3], but the reference bound to y is created and to keep the rules of references valid, x becomes immutable. However once the reference bound to 'y's lifetime ends, the reference that is bound to 'x' becomes mutable again. The reference bound to 'y's has a lifetime from lines 2 and 3 so after line 3, 'x' becomes mutable again. If we swap lines 3 and 4, this will not compile since 'x' only becomes mutable after 'y's lifetime ends.

```
1 let mut x = String::from("Hello");
2 let y = &x;
3 x.push_str(" World");
4 println!("{}",y); // this does not compile
```

It is important to note that variables are not references. References have a lifetime property (something that is checked by the borrow checker to ensure when a reference is valid). Variables are valid to use during their scope. A variable's scope ends when the variable is popped of stack (typically a function call ending, but could be ending a code block, etc). A reference's lifetime is from when it is created to when it is are last used.

What makes things confusing is that a variable may be bound to a reference which may be invalid. So while a variable is valid to use, the data it is bound to is not.

```
let mut x = String::from("Hello");
let mut b = String::from("other");
let mut y = &mut x;         // mutable borrow, all past references become invalid;
                            // x cannot be used until this reference's lifetime ends.
// println!("{x},{y}");     // fails because above reason. using x while that references's
                            // lifetime is still valid
println!("{y}");            // can use y though

let a = &x;                 // immutable borrow. All previous references to x become
                            // immutable. since y was last used in previous line, it's
                            // lifetime ended. Since y's lifetime ended, we can use x again.
                            // y still in scope, but the reference it's bound to is invalid
// println!("{y}");         // fails because the reference y is bound to is invalid.
y = &mut b;                 //y is bound to new reference. can now use y.
println!("this works: {y}");

let b = &x;                 // can have infinitely many immutable borrows to x. this is fine
println!("{x},{a},{b}");    // all good.

let z = &mut x;             // mutable borrow, all past references become invalid
                            // a,b still in scope, but the references they were bound to
                            // were last used in previous line so those references'
                            // lifetimes end. Cannot use x until the reference created
                            // here's lifetime ends

z.push_str(" World");       // the reference that is a mutable borrow to "hello" is last
                            // used here. So it's lifetime ends.
println!("{x}");            // can now use x again
```

[3]technically owner reference here is different, but different in a way that is not important right now

Lifetimes are actually part of a reference's type in Rust. Thus, when we use references in functions, we need to make sure we know their lifetimes. A function with the type signature `fn this(x:&'a i32)` would not accept a piece of data that has a lifetime of `'b`. Now we can't explicitly define a lifetime in Rust, but we can compare lifetimes to each other. We can say that two variables 'a' and 'b' have either different lifetimes, or the same lifetime. We use a generic modifier like we did in OCaml: 'a, 'b, etc. For the most part we no longer have to manually annotate lifetimes on references since Rust has some handy rules for determining lifetimes through type inference. This is done with an extension to the type checker called the Borrow checker. Rust will run this checker before compilation and try to figure out lifetimes. If Rust cannot determine the lifetimes through the borrow checker, you need to help the compiler out by annotating lifetimes for references. If the borrow checker sees a contradiction of lifetimes, then Rust will not compile your program. Let's first look at the rules of lifetimes and then we can see examples:

- Rule 1: For every parameter that is a reference: we assign a new lifetime generic
- Rule 2: if there is exactly one input reference, if the output is a reference, we assign it to the same lifetime as the parameter
- Rule 3: If there are more than one input reference but one of them is `&self` or `&mut self`, then if the output is a reference, it recieves the same lifetime as the `self` parameter.

Let's consider the following function headers:

```
fn this(x: &i32, y: &i32, z: &i32) -> i32
// Rust will use rule 1 here to give each parameter a different lifetime
// -> fn this<'a,'b,'c>(x: &'a i32, y: &'b i32, z: &'c i32) -> i32
// <> after a function means it will use a generic. Much like Arraylist<T> uses a generic
// More on generics later

fn that(x: &i32,y:i32) -> &i32
// Rust will use rule 1 to give every input reference a different life time.
// -> fn that<'a>(x: &'a i32,y:i32) -> &i32
// Notice that 'y' doesn't get a lifetime. That is because it's not a reference
// Rust will then use rule 2 to give the output reference a lifetime
// -> fn that<'a>(x: &'a i32,y:i32) -> &'a i32

fn otherthing(&self, x: &i32, y:i32) -> &i32)
// Here &self is a reference to an object. Like in puython where you have self to refer to
// the current object, Rust also uses self. This means we have things like Objects and
// structs in Rust.
// Using rule 1 we give every input reference a lifetime
// -> fn otherthing<'a,'b>(&'a self, x: &'b i32, y:i32) -> &i32)
// Rule 2 does not apply here since there are more than one input refernce
// Rules 3 does apply however so we can assign the output reference a lifetime
// -> fn otherthing<'a,'b>(&'a self, x: &'b i32, y:i32) -> &'a i32)
```

However, there are cases when these rules cannot cover or determine the lifetimes of all references. What happens if we have multiple input references, but none are `self`? In these cases, we need to manually give lifetimes to lifetimes to the parameters.

```
fn this(x: &i32, y: &i32) -> &i32
// Rust will attempt to use rule 1 and give a lifetime to every input
// -> fn this<'a, 'b>(x: &'a i32, y: &'b i32) -> &i32
// Rule 2 does not apply, nor does rule 3.
// We cannot determine the output's lifetime so Rust Fails and makes use
// put explicit lifetimes on the parameters.
// either of the following would work
fn this<'a>(x: &'a i32, y: &'a i32) -> &'a i32
fn this<'a,'b>(x: &'a i32, y: &'b i32) -> &'a i32
```

```
// The following would not work since there would be no way for the
// function to make a new lifetime without breaking rule 1 of references
fn this<'a,'b,'c>(x: &'a i32, y: &'b i32) -> &'c i32
```

1.8.1 Dangling Pointers

So now that we have an idea about what a lifetime is, and how Rust infers a lifetime for a reference, we should probably see why Rust decided to make rules for this.

It all stems from the first rule of references: that all references must be valid. In order to determine if a reference is valid, Rust needed some way to determine if a reference was actually pointing to live data, or data that is invalid. Consider the following C Code:

```
int main(){
  char* s1 = malloc(sizeof(char)*6);
  strcpy(s1,"hello");
  char* l;
  {
    char* s2 = malloc(sizeof(char)*4);
    strcpy(s2,"bye");
    l= longest(s1,s2);
    free(s2);
  }
  printf("%s is longer",l);
}

char* longest(char* x, char* y){
  if (strlen(x) > strlen(y)){
    return x;
  }else{
    return y;
  }
}
```

In this program, all pointers will be valid and you will not have any memory safety issues. I could however, change s2 to make this program unsafe:

```
int main(){
  char* s1 = malloc(sizeof(char)*6);
  strcpy(s1,"hello");
  char* l;
  {
    char* s2 = malloc(sizeof(char)*10);
    strcpy(s2,"byebyebye");
    l= longest(s1,s2);
    free(s2);
  }
  printf("%s is longer",l);
}

char* longest(char* x, char* y){
  if (strlen(x) > strlen(y)){
    return x;
  }else{
    return y;
  }
}
```

Here, `l` is a pointer trying to point to the value of `’byebyebye"` but that memory area has already been freed. If we are lucky, we will get a segfault.

This small change can make a program safe or unsafe. Rust will fix this problem and enforce the former using lifetimes. Let’s convert this to Rust without explicit lifetimes real quick:

```
fn main(){
  let s1 = String::from("hello");
  let long;
  {
    let s2 = String::from("bye");
    long = longest(&s1,&s2);
  }
  println!("{} is longer",long);
}

fn longest(x:&str, y:&str) -> &str{
  if x.len() > y.len() {x} else {y}
}
```

During compilation, Rust’s borrow checker will make sure that all references are valid and try to infer the lifetime (type) of all the references. Let’s take a look at `longest`:

```
fn longest(x:&str, y:&str) -> &str
// Rule 1: give each input reference a different lifetime:
-> fn longest<’a,’b>(x:&’a str, y:&’b str) -> &str
// Rule 2 does not apply: there is more than one input reference
// Rule 3 does not apply because there is no self.
```

If we try to apply the above rules, Rust will be confused as to what the return type of the `longest` function is so it will not compile this program.

So we need to manually add the annotations of the lifetimes. We could try a few ways:

```
fn longest<’a,’b,’c>(x:&’a str, y:&’b str) -> &’c str
// If we try to do this, we will not compile because a lifetime of ’c could only be created
// in the function leading to a dangling pointer

fn longest<’a,’b>(x:&’a str, y:&’b str) -> &’a str
// this is valid, but recall that a lifetime is part of type. This means we could not
// return y, because y (&’b str) has a different type than the the return value (&’a str)
fn longest<’a>(x:&’a str, y:&’a str) -> &’a str
// this is valid and would allow us to return either x or y
```

As noted in the comments, only real way to add lifetimes for a generic longest string function would be the last way (`fn longest<’a>(x:&’a str, y:&’a str) -> &’a str`). However, how does that impact our program?

```
fn main(){
  let s1 = String::from("hello");
  let long;
  {
    let s2 = String::from("bye");
    long = longest(&s1,&s2);
  }
  println!("{} is longer",long);
}

fn longest<'a>(x:&'a str, y:&'a str) -> &'a str{
  if x.len() > y.len() {x} else {y}
}
```

This means that both inputs to `longest` have to live at least as long as the return value. This is not true: while `s1` lives at least as long as the return variable `long`, `s2` does not live this long. So Rust will not compile this program (even though we know this particular program is safe). This is an example of why Rust has a steep learning curve and why you will be fighting the compiler on some fronts. It also showcases that Rust uses a conservative approach to safety.

1.8.2 Structs and Traits

1.8.3 Smart Pointers

When we used pointers in C, they were pretty straightforward: a pointer was just something that contained a memory address. This meant that we could a whole bunch of unsafe things like read outside the bounds of an array. We know that in higher level languages like Java, this is abstracted away and we get something like an `IndexOutOfBoundException`. Rust wants to be safe so it needs to figure out an answer to this problem.

To solve this problem, Rust does what any good language does and takes from something else. C++ was the first language to introduce the idea of **smart pointers**. A smart pointer is a wrapper for a pointer that also includes metadata about what is being pointed to [4]This is typically done via a `struct`.

We actually have seen multiple smart pointers already: the String and Vec types. In Rust, smart pointers also allow you to own the data the point to. A typical reference can only borrow. In Rust, smart pointers implement both the `deref` and `drop` traits. The `deref` trait allows you to treat the smart pointer like any other pointer rather than a pointer to a pointer. The `drop` trait allows you to write a deconstructor and tell rust how to drop what the smart pointer is pointing to.

trit objects allow us to say something of a particular tye. Box<dyn Draw> is different than Box<T>. Consider Vecs. they need to be the same type. I could get around this with an enum, but also gross. Vec<Box<dyn Draw» allows me to say vector of boxes that point to things that have a trait. useful!

recall onwership rules. there may be times this is not good. a linked list that we want two things to point to a already existing thing. also maybe a tree or graph? why did ownership exist? to prevent double freeing. if we actualyl make a garbage collector (ref count) then we don;t need to worry about this. so we can throw out ownership rules and use a garbage colelctor (though Rust's rc is static not dynamic in terms of lifetime determination so is it really a gc? who knows

let a = Cons(5,Box::new(Cons(10,Box::new(Nil)))); let b= Cons(3,Box:::new(a)) // new takes ownership let c = Cons(4,Box::new(a)). could fix this with clone... but then list would need to implement clone we can geta round this with RC or reference counter. Rc::clone(a) Rc::clone(a). this is fine. drop will decrement count, clone will increase it

[4]Some language make a distinction between fat and smart pointers. I am not sure if Rust does. A Fat pointer typically has the pointer and metadata. A smart pointer is a pointer which can have metadata, but also has additional functionality