More Rust

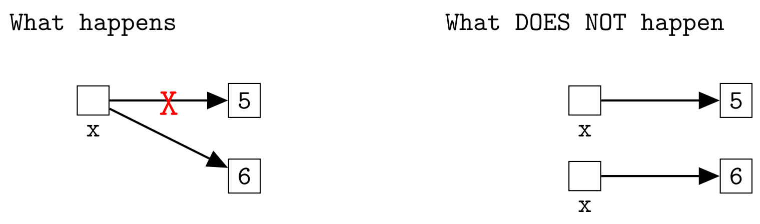
By CNMPeralta, Revised 1st Semester, A.Y. 2020-2021

1 More on Mutability

As stated before, Rust variables are immutable by default. Mutability can be introduced using the `mut` keyword.

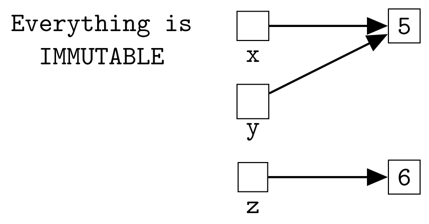
```
let x = 5;     //Immutable
x = 6;         //ERROR

let mut x = 5;//Mutable
x = 6;         //VALID
```

Mutability in Rust applies to the *variable binding*. A mutable binding means that the identifier can be used to point to another instance of the same type (in this case, `i32`).



A reference can be created to a variable using the `&` symbol. Consider the example below. The reference variable `y` will be immutable, that is, you cannot point the reference variable anywhere else. Moreover, you can't change the value it is pointing to, because `x` is also immutable.

```
let x = 5;
let y = &x;
let z = 6;
y = &z; //error
*y = 7; //error
```



To make things mutable, just use the `mut` keyword where appropriate. Note that for a reference variable to be able to change a value it is pointing to, it must have a mutable reference binding, **AND** the variable it is borrowing must be mutable as well.

EXAMPLE	**DESCRIPTION**

`let mut x = 5;`
`let y = &x;`
`*y = 7;`
	`x` is mutable, but the reference binding of `y` to `x`'s value is not. This will result in an error.

`let x = 5;`
`let y = &mut x;`
`*y = 7;`
	`x` is not mutable, but `y` is attempting to make a mutable reference binding to `x`. This will also result in an error.

`let mut x = 5;`
`let y = &mut x;`
`*y = 7;`
	Both `x` and the reference binding of `y` to `x`'s value are mutable. This will <u>not</u> result in an error.

2 Matching

Rust also provides a construct similar to C's `switch` statement: `match`. To use it:

```
let x = 5;
match x {
   0 => println!("zero"),
   1 => println!("one"),
   2 => println!("two"),
   3 => println!("three"),
   4 => println!("four"),
   5 => println!("five"),
   _ => println!("something else"),
}
```

If you wish to have multiple statements in a branch, use curly braces as delimiters:

```
let x = 5;
match x {
   1 => {
      println!("one");
      println!("1");
   },
   _ => println!("not one"),
}
```

The last branch, using the *exhaustive check* symbol (_), is **required**. Omitting the _ branch will result in an error. The _ is similar to the `default` branch of a C `switch` statement: it catches all possible values that are not in a previous arm of the `match` statement.

3 File Input and Output

To open a file for reading (no writing), do the following:

```
use std::error::Error; //Error class
use std::fs::File;     //File class
use std::io::prelude::*; //for file reading (io)
use std::path::Path;   //Path class

fn main(){
   let path = Path::new("files/in.txt");
   let display = path.display();
   let mut file = match File::open(&path){
      //Description method of the error returned by open (if it fails)
      //This clause will be used if an error occurs.
      Err(why) => panic!("couldn't open {}: {}", display, why.description()),
      Ok(file) => file,
      //If there are no problems opening the file (Ok), then it is opened
      //and accessible using the variable file
   };

   let mut s = String::new();
   match file.read_to_string(&mut s){
      Err(why) => panic!("couldn't read {}: {}", display, why.description()),
      Ok(_) => print!("{} contains:\n{}", display, s),
      //s will print whole content of file
   }

   //file variable goes out of scope
   //file automatically gets closed
}
```

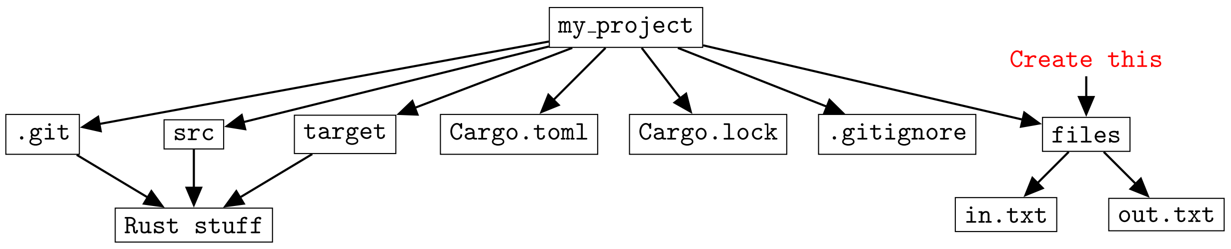
To open a file for writing (no reading), do the following:

```
use std::error::Error;
use std::fs::File;
use std::io::prelude::*;
use std::path::Path;

fn main(){
   let path = Path::new("files/out.txt");
```

```
    let display=path.display();
    //creates a new file
    let mut file = match File::create(&path){
       Err(why) => panic!("couldn't create {}: {}",display,why.description()),
       Ok(file) => file,
    };

    let mut s = "We love Rust!";
    //writes in the file
    match file.write_all(s.as_bytes()){
       Err(why) => panic!("couldn't write {} to {}",
          why.description(),display),
       Ok(_) => print!("successfully wrote to {}", display),
    };

    //file variable goes out of scope
    //file automatically gets closed
}
```

Project Directory Structure

For files to viewable for Rust, they must be placed in the root directory of your project:



4 Regular Expressions

To use regular expressions, a dependency must be added to your Rust project's Cargo.toml:

```
[package]
name = "project name"
version = "0.1.0"
authors = ["your name"]

[dependencies]
regex = "0.1"
```

Moreover, the following must be added to your crate root (usually `main.rs`):

```
extern crate regex;
use regex::Regex;
```

When that is done, you can use Rust's implementation of regular expressions:

```
let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
assert!(re.is_match("2014-01-01"));
```

Note the presence of an `r` before the opening double quote. This marks the strings as a raw string, which are just like regular strings but do not process escape sequences. Thus, you do not need to write `\\d` (to escape backslash) as you would have to if you were using a regular string. `"\\d"` is the same as `r"\d"`.

Iterating Over Capture Groups

Iterating over capture groups allows Rust to test a pattern repeatedly on a search string to find non-overlapping matches. For example:

```
let re = Regex::new(r"(\d{4})-(\d{2})-(\d{2})").unwrap();
let text = "2012-03-14, 2013-01-01 and 2014-07-05";
for cap in re.captures_iter(text) {
   println!("Month: {} Day: {} Year: {}", cap.at(2).unwrap_or(""),
                                          cap.at(3).unwrap_or(""),
                                          cap.at(1).unwrap_or(""));
}
```

The regular expression is able to capture multiple instances of dates from the text, and is also able to isolate month, day, and year values by introducing **parentheses** to the regular expression:

```
(\d{4})-(\d{2})-(\d{2})
   1       2       3
```

These groupings are accessed using the `.at()` function of the `cap` loop variable. The year is `.at(1)`, the month is `.at(2)`, and the day is `.at(3)`.

Naming Capture Groups

Capture groups can also be named by using `?P<group_name>` inside its grouping. Modifying the above regex, we have:

```
(?P<y>\d{4})-(?P<m>\d{2})-(?P<d>\d{2})
   1 or y       2 or m       3 or d
```

Using these names in our example, we have:

```
let re = Regex::new(r"(?P<y>\d{4})-(?P<m>\d{2})-(?P<d>\d{2})").unwrap();
let text = "2012-03-14, 2013-01-01 and 2014-07-05";
for cap in re.captures_iter(text) {
   println!("Month: {} Day: {} Year: {}", cap.name("m").unwrap_or(""),
                                          cap.name("d").unwrap_or(""),
                                          cap.name("y").unwrap_or(""));
```

References

[1] Rust File I/O. https://doc.rust-lang.org/book/second-edition/ch01-00-getting-started.html.

[2] Rust Match. https://doc.rust-lang.org/book/second-edition/ch01-00-getting-started.html.

[3] Rust Regex, 2018. https://doc.rust-lang.org/book/2018-edition/.