Draft - Rust Book Notes - Not a Cheat Sheet

by seannarr via cheatography.com/61019/cs/15819/

Installation

```
curl https://sh.rustup.rs -sSf | sh
export
PATH="$HOME/.cargo/bin:$PATH"
```

Hello World

```
fn main() { println!("Hello,
world!"); }
```

compile: rustc main.rs	prinln! = macro!

create variables with let

`let foo =5;` // immutable (default)

`let mut foo=5;` //mutable

`let foo=5; let foo="hello";`
Shadowing allows reuse of variable. useful in type conversions

`'let foo: u32 =5;` //annotating with type using :

`let foo=5; foo = 6;` error[E0384]: cannot assign twice to immutable variable `foo`

const

```
const MAX_POINTS: u32 = 100_000;
// no mut allowed on const.
// it should be annotated with
datatype.
// It is visible with in the scope
it's declared.
// only constant expression
assignment.. not return value from
function or expression evaluated at
runtime
```

Prelude

Rust inserts
`extern crate std;`
into the crate root of every crate, and
`use std::prelude::v1::*;`

Prelude (cont)

into every module.
`std::prelude::v1`

Prelude is set of types Rust imports.

Import external crate (library)

```
extern crate rand; //external
dependency
use rand::Rng; //bring Rng trait
which defines the methods into
scope
those in prelude, need not to
extern the crate.
just use std::io; //brings io trait
into scope
```

&mut and stdin() and io::Result

```
io::stdin() // stdin() =
std::io::Stdin instance a handle to
standard input
    .read_line(&mut guess) // &mut
- pass by reference and make it
mutable
    .expect("Failed to read line")
// io::Result -> If returns Err, it
crashes displaying the message
```

io::Result -> Result, Enumerations Ok, Err. If ok, returns the value, if Err, it crashes program. Without expect, compiler warning - Unused io::Result which must be used.

Floating-Point Types

f32 single precision	f64 - double precision (default)

Math Operators

+, -, *, /

bool

true, false

`let t = true; let f: bool = false;`

Character Type

`let c='z'; let k: char = 'a';`

char is a unicode scalar value.

Tuple Type

`let x:(u32,f64,u8) = (6,3.2,1); x.0 => 6, x.1 =>3.2`

destructuring - `let (a,b,c) =x;` a=> 6

first index in tuple is 0. ex: x.0 => 6

Array Types

Fixed Size vs Vector's size can change

let a =[1,2,3,4]

Elements of same type

access by index: a[0], a[1]

Invalid Access a[10] :Runtime error: Index out of bounds

match => arms; arm : pattern => code

```
match guess.cmp(&secret_number) {
=> arms
        Ordering::Less => println!
("Too small!"), //pattern => code
        Ordering::Greater =>
println!("Too big!"),
        Ordering::Equal => { //code
block
                println!("You
win!");
```

match => arms; arm : pattern => code (cont)

```
            }
    }
//similar to switch case? but
concise !
```

Integer Types

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
arch	isize	usize

----

arch 32/64 bit system - isize =i32/64, usize=u32/u64

Signed	-(2^{n-1}) to 2^{n-1} - 1
Unsigned	0 to 2^n-1

arch (useful in indexing collections)

type suffix	57u8
visual separator	1_000

default type is u32 even on 64bit arch

Functions fn

snake case - lowercase words separated by (_) underscore	`fn one_two() { }`

Functions fn (cont)

parameters: name **:** type	`fn foo_bar(x: i32, y:u32) { }`

statement vs expression

statement ends with ; and does not evaluate to a value;

expression doesn't end with ; and evaluates to a value

code block expression {} x+1 is expression which is returned	`let y = { let x = 3; x + 1 }` value of y will be 4;
Error: expected expression, found statement (`let`)	`let a = (let b =2);`
functions with return value -> type	`fn five() -> i32 { 5 };` 5 is expression as no colon, and return value as it's last expression
`fn plus_one(x: i32) -> i32 { x + 1; }`	; turns into statement, and empty tuple () will returned, will be a compiler error as () is not i32

enums - methods

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
impl Message {
    fn call(&self) {
        // method body would be
defined here
    }
}
let m =
Message::Write(String::from("hello"
));
m.call();
```

options<T> alternate Null implementation

```
enum Option<T> {
    Some(T),
    None,
}
let some_number = Some(5);
let some_string = Some("a
string");
let absent_number: Option<i32> =
None;
//error
let x: i8 = 5;
let y: Option<i8> = Some(5);
let sum = x + y; //error as x and y
are two different types
error[E0277]: the trait bound i8:
std::ops::Add<std::option::Option<i
8>> is not satisfied
```

match - _ placeholder

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (), unit value
}
```

Control flow - if { } else if { } else { }

if number == 3 { }	arm
condition should evaluate to bool type	or mismatched types error
if condition { } else if condition { } else { }	blocks of code called arms
let a = if a == 3 { 2 } else { 5}	expressions in all arms should evaluate to same type

match handling Result

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num, //Ok receives num from return Result, which is returned by match
    Err(_) => continue, // _ underscore catches all values
};
```

println!

No argument indices.. just simple brace {}

```
println!("x = {} and y = {}", x, y);
```

Control flows - while and for

```
while number != 3 { number = number +1; }
```

```
let a =[1,2]; for element in a.iter() { }
```

`for number in (1..4).rev() {`	alternate approach to while

Ownership

My first reaction to the concept of Ownership	WOW!
Value has a variable	Owner
When owner goes out of scope	Value is dropped
variable assignment, Passing to as function parameter or returning from function, it is moved.	`let s1 = String::from("hello"); let s2 = s1;` s1 is no longer valid. It is moved. Only s2 is the owner

error[E0382]: use of moved value: `s1`

Reference doesn't have ownership	when it goes out of scope, nothing happens.
Pass by reference	it is not moved. It's just borrowed.

References are immutable by default

Ownership (cont)

Mutable References	`fn main() { let mut s = String::from("hello"); change(&mut s); } fn change(some_string: &mut String) { some_string.push_str(", world"); }`

Memory is managed through a system of ownership with a set of rules that the compiler checks at compile time. At compile time!!!

Cargo

`cargo --version`

`cargo new hello_cargo --bin` creates `Cargo.toml` and main.rs

--bin=bin(ary) or library.

source control: default git --vcs=

Cargo is Rust's build system and package manager.

Cargo.toml

```
[package]
name = "hello_cargo" #name of the executable
version = "0.1.0" #version
authors = ["Your Name <you@example.com>"] #Cargo gets name and email from the environment
[dependencies] #packages aka crates
```

TOML: Tom's Obvious, Minimal Language

import crate toml

```
[dependencies]
  rand = "0.3.14" # SimVer ~ ^0.3.14 any version that is compatible with 0.3.14
```

Adding crates to toml file.

cargo build , cargo run , cargo check

```
cargo build # creates an executable file in target/debug/hello_cargo
cargo run # build and run
cargo check #compilation check, no building executable
cargo build --release
```

error[E0308]: mismatched types

```
error[E0308]: mismatched types --> src/main.rs:23:21

     match guess.cmp(&secret_number)
               ^^^^^^ expected struct std::string::String, found integral variable
  = note: expected type &std::string::String
   = note: found type &{integer}
```

cargo.lock

Cargo maintains versions in cargo.lock file

cargo update

cargo updates all versions upto next symver

registry

Cargo fetches external dependencies and their dependencies from registry, a copy from the crates.io.

crates.io is a public repo

loop

loop { }

`break;` exits the loop

cargo test

```
#[cfg(test)]
mod tests {
    #[test] --> test
    fn it_works() {
        assert_eq!(2 + 2, 4);
    }
}
```

Doc-tests adder documentation tests? to have examples
assert! false value, assetseq! == assertive! !=
[should_panic] to expect panic!
[ignore]
cargo test -- --test-threads=1 stop parallel run
cargo test --nocapture , no print output
cargo test add //runs tests containing add

tests organization

`#[cfg(test)] mod tests { }`	compile only cfg is test
tests folder	integration tests

struct

```
struct User {
    username: String, field => name :type
    email: String,
    sign_in_count: u64,
    active: bool,
}
// Instantiating
let user1 = User { //struct name
    email: String::from("someone@example.com"), //and
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
//dot notation
let mut user2 = User { };
user2.email = String::from("someone@example.com");
```

unit-like structs without fileds () .. Used to implement traits with out data on the type.

ownership - lifetimes?

Comments

//	/* */
//! //	

tuple structs

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);
let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);
```

Draft - Rust Book Notes - Not a Cheat Sheet

by seannarr via cheatography.com/61019/cs/15819/

struct - instantiating options

```
// .. shorthand
let user2 = User {
    email:
String::from("another@example.com")
,
    username:
String::from("anotherusername567"),
    ..user1 // .. remaining fields
should be from user1 instance
};
// shorthand - when variables and
fields have same names
let email ="";
let user2 = User {
  email //shortHand
}
```

struct methods

```
struct Rectangle {
    width: u32,
    height: u32,
}
impl Rectangle {
    fn area(&self) -> u32 { //first
parameter should be self , instance
of the struct
        self.width * self.height
    }
    fn square(size: u32) ->
Rectangle { //associated function
Rectangle::square
        Rectangle { width: size,
height: size }
    }
}
```

&mut self to modify struct

struct #[derive(Debug)]

println!("rect1 is {}", rect1);

error[E0277]: the trait bound `Rectangle: std::fmt::Display` is not satisfied

{:?} ? to use Debug Trait

```
#[derive(Debug)]
struct Rectangle {
width: u32, height: u32
}
```

{:#?} to pretty print

Derived Traits

struct as expression

```
fn build_user(email: String,
username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

modules

```
mod network {
    fn connect() {
    }
    mod client { //nested module
        fn connect() {
        }
    }
}
```

network::client::connect();
network:connect();

module - referencing a submodule

```
mod client; => mod client {
//client.rs contents here }
mod network {
 //snippet
}
//contents of clents.rs
fn connect { // No need to add mod
declaration
}
```

modules-tree

```
communicator
 ├── client
 └── network
     └── server
└── src
    ├── client.rs
    ├── lib.rs // mod client; mod
network;
    └── network
        ├── mod.rs mod server;
        └── server.rs
```

modules- rules

```
└── foo
    ├── bar.rs (contains the
declarations in foo::bar)
    └── mod.rs (contains the
declarations in foo, including mod
bar)
```

If a module named foo has no submodules, you should put the declarations for foo in a file named foo.rs.

If a module named foo does have submodules, you should put the declarations for foo in a file named foo/mod.rs.

pub - privacy rules

If an item is public

> it can be accessed through any of its parent modules

If an item is private

> it can be accessed only by its immediate parent module and **any of the parent's child modules**

use

```
bring modules into scope.
pub mod a {
    pub mod series {
        pub mod of {
            pub fn
nested_modules() {}
        }
    }
}
fn main() {
    a::series::of::nested_modules()
;
}
use a::series::of;
fn main() {
    of::nested_modules();
}
```

In use statement, paths are relative to the crate root by default
super:: confusing? if the module privacy rules state that parent and its immediate children of the parent can access private items, then why we need Super?

Ownership - References Rules

`Only one mutable reference in a particular scope. Prevents datarace`

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s; //  error[E0499]:
cannot borrow s as mutable more
than once at a time
```

`Combination of mutable and immutable references are not allowed. to guarantee immutability.`

```
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // error[E0502]:
cannot borrow s as mutable because
it is also borrowed as immutable
```

One mutable reference restriction prevents data race.
Only All Readers or just One Writer are allowed.

Dangling References

```
fn main() {
    let reference_to_nothing =
dangle();
}
fn dangle() -> &String {
    let s =
String::from("hello");
    &s // It is returning
reference, borrowed value..
requiring s to be live outside this
scope
} //Compiler check
```

error[E0106]: missing lifetime specifier
= help: this function's return type contains a borrowed value, but there is
no value for it to be borrowed from
= help: consider giving it a 'static lifetime

Slices

```
fn main() {
    let mut s =
String::from("hello world");
    let word = first_word(&s);
    s.clear(); // Error!
}
fn first_word(s: &String) -> &str
//immutable {
    let bytes = s.as_bytes();
    for (i, &item) in
bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
//borrowed as immutable
        }
    }
    &s[..]
}
```

error[E0502]: cannot borrow **s** as mutable because it is also borrowed as immutable.
Slicing : [..1] to start at 0, [2..] to the end.

enums

```
enum IpAddrKind {
    V4,
    V6,
}
struct IpAddr {
    kind: IpAddrKind, // type
    address: String,
}
let home = IpAddr {
    kind: IpAddrKind::V4,
```

enums (cont)

```
    address:
String::from("127.0.0.1"),
};
let loopback = IpAddr {
    kind: IpAddrKind::V6,
    address: String::from("::1"),
};
```

enums - variations

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
struct Ipv4Addr {
    // --snip--
}
struct Ipv6Addr {
    // --snip--
}
enum IpAddr {
    V4(Ipv4Addr),
    V6(Ipv6Addr),
}
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

enum - bringing some variants into scope

```
enum TrafficLight {
    Red,
    Yellow,
    Green,
}
use TrafficLight::{Red, Yellow};
fn main() {
    let red = Red;
    let yellow = Yellow;
    let green =
TrafficLight::Green;
}
or
use TrafficLight::*;
```

glob operator * to bring all items in a namespace.

match - enum

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
fn value_in_cents(coin: Coin) ->
u32 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
// state value is bind to the
variable
```

match - enum (cont)

```
            println!("State quarter
from {:?}!", state);
            25
        },
    }
}
```

if let

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
or
//if let concise for one pattern
if let Some(3) = some_u8_value {
    println!("three");
}
```

vectors Vec<T>, vec! macro

`let v: Vec<i32> = Vec::new();`	annotate type when not initialized with data
`let v = vec![1, 2, 3];`	macro vec! to create instance and hold data

vectors Vec<T>, vec! macro (cont)

`let mut v = Vec::new(); v.push(5); v.push(6);`	mut to update the vector. Rush infers the datatype from push.
Inferring datatype from push?	

fn function pointer

```
Use existing functions in place of closure :
fn add_one(x: i32) -> i32 {
    x + 1
}
fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}
fn main() {
    let answer = do_twice(add_one, 5);
    println!("The answer is: {}", answer);
}
//return the closure
fn returns_closure() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Vector access elements

```
let v = vec![1, 2, 3, 4, 5];
let third: &i32 = &v[2]; => access with reference
let third: Option<&i32> = v.get(2); //return None
let hundredth: &i32 = &v[100]; //panic , use get
//
```

Vector access elements (cont)

```
let mut v = vec![1, 2, 3, 4, 5];
let first = &v[0]; // immutable borrowing
v.push(6); //immutable borrow above line, it's error to borrow mutable reference again
```

Vector iterator

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
//mutable vector and i to dereference using
let mut v = vec![100, 32, 57];
for i in &mut v {
    *i += 50;
}
```

Vector: enum to store different types

```
enum SpreadsheetCell {
    Int(i32),
    Float(f64),
    Text(String),
}
let row = vec![
    SpreadsheetCell::Int(3),
    SpreadsheetCell::Text(String::from("blue")),
    SpreadsheetCell::Float(10.12),
];
```

String, str

```
str: string literals are stored in the binary program.
format: UTF8
OsString, OsStr, CString, and CStr are string variant libraries.
//creating strings
let mut s = String::new();
let s = "initial contents".to_string();
//to_string and from are matter of style
let s = String::from("initial contents");
let hello = String::from("■■■■■■■");
//utf-8
```

String update and concatenation +/ format!

```
let mut s = String::from("foo");
s.push_str("bar"); // foobar append , it takes slice so no ownership transfer
let mut s = String::from("lo");
s.push('l'); //character , lol
// concatenation +
let s1 = String::from("Hello, ");
let s2 = String::from("world!");
let s3 = s1 + &s2; // Note s1 has been moved here and can no longer be used
fn add(self, s: &str) -> String {
// s2 string => str deref coersion &s2 => &s2[..]
//more than two string, use
let s1 = String::from("tic");
let s2 = String::from("tac");
```

String update and concatenation +/ format! (cont)

```
let s3 = String::from("toe");
let s = format!("{}-{}-{}", s1, s2, s3);
```

strings indexing

```
let s1 = String::from("hello"); let h = s1[0];
```

error[E0277]: the trait bound `std::string::String: std::ops::Index<{integer}>` is not satisfied
A String is a wrapper over a Vec<u8>. UTF8, 2 bytes for some characters, 1byte for some

let len = String::from("Hola").len(); => 4

let len = String::from("Здравствуйте").len(); => 24

Error - Recoverable Result, Unrecoverable panic!

`RUST_BACKTRACE=1 cargo run`	stack trace ? (enable debug symbols)
Err(ref error) if error.kind() == ErrorKind::NotFound =>	error.kind()
unwrap	error => panic, ok=> returns value
expect	message when error occurs
`let mut f = File::open("hello.txt")?;`	? to propagate error. From trait. From::from, error type should implement from

```
File::open("hello.txt")?.read_to_string(&mut s)?;
```

Error - Recoverable Result, Unrecoverable panic! (cont)

? for return type Result

Stack unwinding -
[profile.release]
panic = 'abort'

Cargo in depth

```
//Cargo profiles
[profile.dev]
opt-level = 0 //overriding defaults
[profile.release]
opt-level = 3
//re exporting API
pub use kinds::PrimaryColor;
pub use kinds::SecondaryColor;
pub use utils::mix;
cargo.io login: cargo login
abcdefghijklmnopqrstuvwxyz012345
unique package name to publish and
cargo publish
cargo yank --vers 1.0.1
```

HashMap<K, V>

```
use std::collections::HashMap; keys of the same type and values of the same type.
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
//creating a hash map from two vectors
let teams = vec![String::from("Blue"), String::from("Yellow")];
```

HashMap<K, V> (cont)

```
let initial_scores = vec![10, 50];
let scores: HashMap<_, _> = teams.iter().zip(initial_scores.iter()).collect();
//HashMap to get the desired type from collect .. strange way of specifying return value
// _, _ Rust infers the data types of Key and Value
//access value
let mut scores = HashMap::new();
scores.insert(String::from("Blue"), 10);
scores.insert(String::from("Yellow"), 50);
let team_name = String::from("Blue");
let score = scores.get(&team_name); //get
//iterating
for (key, value) in &scores { }
//To only insert if key doesn't have a value
scores.entry(String::from("Yellow")).or_insert(50); //returns mutable reference
// let count = map.entry(word).or_insert(0); //to insert 0, for first time key insertion
```

BuildHasher type: default is cryptographically secure hashing, can be slow
Values are moved, and Hashmap takes the ownership of keys and values.

Hasmap insert doesn't take the ownership.
insert of a existing key overrides the value

Generic data types <T>

```
fn largest<T>(list: &[T]) -> T
struct Point<T> {
    x: T,
    y: T,
}
struct Point<T, U> {
    x: T,
    y: U,
}
enum Option<T> {
    Some(T),
    None,
}
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
impl Point<f32> { //specific type
    fn distance_from_origin(&self)
-> f32 {
        (self.x.powi(2) +
self.y.powi(2)).sqrt()
    }
}
struct Point<T, U> {
```

Generic data types <T> (cont)

```
    x: T,
    y: U,
}
//mixup
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other:
Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
```

Monomorphization to specify concrete code at compile time

trait

```
pub trait Summary {
    fn summarize_author(&self) ->
String;
    fn summarize(&self) -> String
{
        format!("(Read more from
{}...)", self.summarize_author())
//default can call other methods in
the trait.
    }
}
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
```

trait (cont)

```
    pub content: String,
}
impl Summary for NewsArticle {
    fn summarize(&self) -> String
{
        format!("{}, by {} ({})",
self.headline, self.author,
self.location)
    }
}
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
impl Summary for Tweet { //for
    fn summarize(&self) -> String
{
        format!("{}: {}",
self.username, self.content)
    }
}
//default implementation
fn summarize(&self) -> String {
        String::from("(Read
more...)")
    }
//to use default implementation,
impl Summary for NewsArticle {}
//empty block
```

we can implement a trait on a type only if either the trait or the type is local to your crate. coherence/orphan rule:people’s code can’t break your code and vice versa

Generic constraints

```
// +
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
//where clause: with mutilple trait bounds
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

Conditionally implement on bounds: impl<T: Display + PartialOrd> Pair<T> {

lifetimes

```
{
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
error[E0597]: x does not live long enough
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
error[E0106]: missing lifetime specifier
```

lifetimes (cont)

```
//lifetime annotations with generics -- means all x, y are has the same lifetime
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

smaller lifetime is chosen.
static - let s: &'static str = "I have a static lifetime.";

Iterator

```
trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
    // methods with default implementations elided
}
```

1.The iter method produces an iterator over immutable references.
2. into_iter to take over ownership of the parent and returns owned values
3. iter_mut - iterate over mutable references
4. consuming adaptors -> uses up iterator such as sum()
5. chain of iterator adaptors following a consumer adaptor gets you the results (ex: collect())
6.

Box<T>, RC<T>, RefCell<T>

RC<T>	Box<T>	RefCell<T>	
multiple	Single	Single	Data Ownership
immutable	immutable or mutable	immutable or mutable	Borrowing
compile time	compile time	runtime	checked at

Because RefCell<T> allows mutable borrows checked at runtime, we can mutate the value inside the RefCell<T> even when the RefCell<T> is immutable

let y = &mut x;? let mut y = x; confusion?

threads

```
let handle = std::thread::spawn(|| { //spawn new thread
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            std::thread::sleep(std::time::Duration::from_millis(1));
        }
    });
    handle.join().unwrap(); //wait to finish
let v = vec![1, 2, 3];
    let handle = thread::spawn(move || { => to let capture take the ownership
        println!("Here's a vector: {:?}", v);
    });
```

threads (cont)

```
error[E0373]: closure may outlive the current function, but it borrows v,
which is owned by the current function
 after move drop(v) ^ error value used here after move
```

JoinHandle is an owned value ?

Closures - Environment capture - Traits

Taking ownership **FnOnce**	as it takes ownership of variables it uses from the environment, closure can be called once
Borrowing mutably **FnMut**	It borrows mutably, so it can change the environment

Borrowing immutably "Fn"

Can FnMut be called multiple times? (which tries borrow mutably in every call) Yes, as call finishes, the variables are available for borrowing.
let equal_to_x = move |z| z == x; to move the ownership of x to the closure.

threads - channel mpsc

```
use std::sync::mpsc;
let (tx, rx) = mpsc::channel(); tx: transmitter, rx: receiver
 thread::spawn(move || {
        let val = String::from("hi");
```

threads - channel mpsc (cont)

```
        tx.send(val).unwrap();//send takes the ownership of val
    });
let tx1 = mpsc::Sender::clone(&tx); //clone a transmitter
```

mpsc : multiple producer, single consumer => multiple senders and one receiver

Mutex<T> , Arc<T>

```
let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];
    for _ in 0..10 {
        let counter = Arc::clone(&counter); //clone
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }
```

Sync and Send

concurrency is part of the standard library not the language.

two concurrency concepts embedded in the language: the std::marker traits Sync and Send

Sync and Send (cont)

The Send marker trait indicates that ownership of the type implementing Send can be transferred between threads	except Rc<T> multiple references but can't be shared between threads

The Sync marker trait indicates that it is safe for the type implementing Sync to be referenced from multiple threads

In other words, any type T is Sync if &T (a reference to T) is Send, meaning the reference can be sent safely to another thread

closures

```
let expensive_closure = |num: u32| -> u32 {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    };
// type inference
let example_closure = |x| x;
let s = example_closure(String::from("hello"));
let n = example_closure(5); //error type inference only one type
// memoization or lazy evaluations
impl<T> Cacher<T>
    where T: Fn(u32) -> u32
{
    fn new(calculation: T) -> Cacher<T> {
```

closures (cont)

```
        Cacher {
            calculation,
            value: None,
        }
    }
    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            },
        }
    }
}
let mut expensive_result = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });
//memoization
```

unsafe

Implementing unsafe trait	`unsafe trait Foo { }`
	`unsafe impl Foo for i32 { }`

mutating static is unsafe

unsafe (cont)

extern "C" { } Foreign Function Interface (FFI)	call in unsafe block
C : application binary interface (ABI)	`extern "C" { fn abs(input: i32) -> i32; }`
Calling Rust from other languages	`#[no_mangle] pub extern "C" fn call_from_c() { println!("Just called a Rust function from C!"); }`

unsafe block to call unsafe functions

1. Dereference a raw pointer
2. Call an unsafe function or method
3. Access or modify a mutable static variable
4. Implement an unsafe trait

Raw pointers

```
Different from references and smart pointers, keep in mind that raw pointers:
1. Are allowed to ignore the borrowing rules and have both immutable and mutable pointers, or multiple mutable pointers to the same location
2. Aren’t guaranteed to point to valid memory
3. Are allowed to be null
4. Don’t implement any automatic clean-up
let mut num = 5;
let r1 = &num as *const i32;
```

Raw pointers (cont)

```
let r2 = &mut num as *mut i32;
unsafe { //dereferencing
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```
const i32 and `mut i32 raw pointers that both pointed to the same memory location, that of num. If instead we’d tried to create an immutable and a mutable reference to num, this would not have compiled because Rust’s ownership rules don’t allow a mutable reference at the same time as any immutable references. With raw pointers, can create mutable pointer and an immutable pointer to the same location, and change data through the mutable pointer, potentially creating a data race.`

Lifetimes adv

```
In our definition of Parser, in order to say that 's (the lifetime of the string slice) is guaranteed to live at least as long as 'c (the lifetime of the reference to Context), we change the lifetime declarations to look like this:
struct Parser<'c, 's: 'c> {
    context: &'c Context<'s>,
}
// lifetime bounds on references to Generic Types
struct StaticRef<T: 'static>(&'static T);
struct Ref<'a, T: 'a>(&'a T);
//Inference of Tait life times
```

Lifetimes adv (cont)

```
The default lifetime of a trait
object is 'static.
With &'a Trait or &'a mut Trait,
the default lifetime is 'a.
With a single T: 'a clause, the
default lifetime is 'a.
With multiple T: 'a-like clauses,
there is no default; we must be
explicit.
Box<Red + 'a> or Box<Red +
'static>
 Just as with the other bounds,
this means that any implementor of
the Red trait that has references
inside must have the same lifetime
specified in the trait object
bounds as those references
```

Advanced Traits

```
pub trait Iterator {
    type Item; // place holder
    fn next(&mut self) ->
Option<Self::Item>;
}
//With Generic, needs to annotate
the type
//Default generic type
trait Add<RHS=Self> { // RHS is
self type a = a + a
    type Output;
    fn add(self, rhs: RHS) ->
Self::Output;
}
fn main() {
    let person = Human;
    Pilot::fly(&person); //quality
to avoid ambiguity
    Wizard::fly(&person);
```

Advanced Traits (cont)

```
    person.fly();
}
Assosciated functions:
//As no self, it can infer
<Dog as Animal>::baby_name()
//NewType
use tuple to creat traits on
external types
```

Implemented directly on the type has precedence over trait impls

Advanced Types

type Kilometers = i32;	Type aliases
let f: Box<Fn() + Send + 'static> = Box::new(|| println!("hi"));	
type Thunk = Box<Fn() + Send + 'static>;	
type Result<T> = Result<T, std::io::Error>;	
! never type = void	

iterating over string - bytes() and chars()

```
for c in "■■■■■■".chars() {
    println!("{}", c);
}
■
■
■
■■
■
■■

for b in "■■■■■■■".bytes() {
    println!("{}", b);
```

iterating over string - bytes() and chars() (cont)

```
}
224
164
// --snip--
165
135
```

Cargo workspaces

```
[workspace]
members = [
    "adder",
]
├── Cargo.lock
├── Cargo.toml
├── add-one
│  ├── Cargo.toml
│  └── src
│  └── lib.rs
├── adder
│  ├── Cargo.toml
│  └── src
│  └── main.rs
└── target
[dependencies]
add-one = { path = "../add-one" }
-- explicit
cargo run -p adder //to run
cargo install $HOME/.cargo/bin
cargo-something => cargo something
```

workspaces: all related crates share Cargo.lock and output directory.
dependencies should be added to cargo.toml files to extern crate