Installation

```
curl https://sh.rustup.rs -sSf | sh
export PATH="$ HOM E/.c ar go/ -bin :$P ATH "
```

Hello World

```
fn main() {  printl n!( " 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="he llo ";` 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 `f oo`

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 assign ment.. 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::p rel ude ::v 1::*;`

into every module.

`std::p rel ude::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
       .re ad_ lin e(&mut guess) // &mut - pass by reference and make it mutable
       .ex pec t("F ailed 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 ,f6 4,u8) = (6,3.2,1);
.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

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

match => arms; arm : pattern => code

```
match guess.cmp(&secret_number)
{ => arms
                Ord eri -
ng: :Less => printl n!( "Too
small! "), //pattern => code
                Ord eri -
ng: :Gr eater => printl n!
( "Too big!"),
                Ord eri -
ng: :Equal => { //code block
                                -
pri ntl n!( "You win!");


                        }
       }
//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	

Integer Types (cont)

default type is u32 even on 64bit arch

Functions fn

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

Functions fn (cont)

```
fn plus_o ne(x: i32) -> i32 {
```

enums - methods

```
enum Message {
        Quit,
       Move { x: i32, y: i32 },
        Wri te( Str ing),
        Cha nge Col or(i32, i32,
i32),       x + 1      }
}
impl Message {
       fn call(&self) {
                // method body
would be defined here
       }
}
let m = Messag e:: Wri te( Str -
ing ::f rom ("he llo "));
m.call();
```

options<T> alternate Null implementation

```
enum Option<T> {
        Som e(T),
        None,
}
let some_n umber = Some(5);
let some_s tring = Some("a
string ");
```


Sponsored by Readable.com
Measure your website readability!
https://readable.com

options<T> alternate Null implementation (cont)

> 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::o ps: :Ad d< s td: :op tio n:: Opt ion <i8 >>` is not satisfied

match - _ placeholder

```
let some_u8_value = 0u8;
match some_u 8_value {
        1 => printl n!( " - one "),
        3 => printl n!( " thr - ee"),
        5 => printl n!( " fiv - e"),
        7 => printl n!( " sev - en"),
        _ => (), 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

Control flow - if { } else if { } else { } (cont)

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 {}

```
printl n!( "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

Ownership (cont)

variable assign-ment, Passing to as function parameter or returning from function, it is moved.	`let s1 = String ::f rom` s2 is the owner

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

Reference doesn't have ownership	when it goes out of scope, nothi
Pass by reference	it is not moved. It's just borrowe

References are immutable by default

Mutable References	`fn main() { let mut s = String :: fn change (so me_ strin rld"); }`

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

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

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 = " hel lo_ car go" #name of the executable
version = " 0.1.0" #version
authors = ["Your Name <yo u@e - xam ple.co m>"] #Cargo gets name and email from the enviro nment
[depen den cies] #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 fi n target /de bug /he llo _ cargo
```

```
cargo run  # build and run
```

```
cargo check #compi lation check, no building executable
```

cargo build , cargo run , cargo check (cont)

```
cargo build --release
```

error[E0308]: mismatched types

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


          match guess.c mp (& - sec ret _nu mber)
                ^^^^^^ expected struct std::s tri ng: :String, found integral variable
   = note: expected type &s td: - :st rin g:: String
     = note: found type &{ int e ger}
```

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() {
                 ass ert _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	integr- ation tests

struct

```
struct User {
        use rname: String, field => name :type
        email: String,
        sig n_i n_c ount: u64,
        active: bool,
}
// Instan tiating
let user1 = User { //struct name
```


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

struct (cont)

```
>    email: String::from("someone@exampl-
e.com"), //and
   username: String::from("someusernam-
e123"),
   active: true,
   sign_in_count: 1,
};
//dot notation
let mut user2 = User { };
user2.email = String::from("someone@exa-
mple.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);
```

struct - instantiating options

```
// .. shorthand
let user2 = User {
        email: String ::f rom -
("an oth er@ exa mpl e.c om"),
        use rname: String ::f -
rom ("an oth eru ser nam e56 7"),
       ..user1 // .. remaining
fields should be from user1
instance
```

struct - instantiating options (cont)

```
> };
// 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
                sel f.width *
self.h eight
       }
       fn square (size: u32) ->
Rectangle { //asso ciated
function Rectan gle ::s quare
                Rec tangle {
width: size, height: size }
       }
}
```

&mut self to modify struct

struct #[derive(Debug)]

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

error[E0277]: the trait bound `Rectangle: std::f mt: :Di splay` is not satisfied

{:?} ? to use Debug Trait

```
#[deri ve( 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,
                use rname:
username,
                active: true,
                sig n_i n_c -
ount: 1,
       }
}
```

modules

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


Sponsored by Readable.com
Measure your website readability!
https://readable.com

modules (cont)

```
> }
```

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

module - referencing a submodule

```
mod client; => mod client {
//client.rs contents here }
mod network {
  //s nippet
}
//contents of clents.rs
fn connect { // No need to add
mod declar ation
}
```

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
declar ations in foo::bar)
       └── mod.rs (contains the
declar ations 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 _mo dules() {}
               }
       }
}
```

use (cont)

```
> 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 ::f rom -
("he llo ");
let r1 = &mut s;
let r2 = &mut s; // error[ -
E0499]: cannot borrow s as
mutable more than once at a time
```

Combin ation of mutable and immutable references are not allowed. to guarantee immuta - bility.

```
let mut s = String ::f rom -
("he llo ");
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.

Sponsored by Readable.com
Measure your website readability!
https://readable.com

Dangling References

```
fn main() {
       let refere nce _to _no -
thing = dangle();
}
fn dangle() -> &S tring {
       let s = String ::f rom -
("he llo ");
        &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 ::f -
rom ("hello world");
       let word = first_ wor -
d(&s);
        s.c lear(); // Error!
}
fn first_ word(s: &S tring) ->
&str //immu table {
       let bytes = s.as_b -
ytes();
       for (i, &item) in
bytes.i te r().en ume rate() {
               if item == b' ' {
                       return
&s [0..i]; //borrowed as
immutable
               }
       }
        &s[..]
```

Slices (cont)

```
> }
```

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: IpAddr Kind, //
type
        add ress: String,
}
let home = IpAddr {
        kind: IpAddr Kin d::V4,
        add ress: String ::f -
rom ("12 7.0.0.1 "),
};
let loopback = IpAddr {
        kind: IpAddr Kin d::V6,
        add ress: String ::f -
rom (":: 1"),
};
```

enums - variations

```
enum IpAddr {
        V4(u8, u8, u8, u8),
        V6( Str ing),
}
```

enums - variations (cont)

```
> 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 Traffi cLi ght ::{Red,
Yellow};
fn main() {
        let red = Red;
        let yellow = Yellow;
        let green = Traffi cLi -
ght ::G reen;
```


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

enum - bringing some variants into scope (cont)

```
>}
or
use TrafficLight::*;
```

glob operator * to bring all items in a namespace.

match - enum

```
enum Coin {
        Penny,
        Nickel,
        Dime,
        Qua rte r(U sSt ate),
}
fn value_ in_ cen ts( coin:
Coin) -> u32 {
        match coin {
                Coi n:: Penny =>
1,
                Coi n:: Nickel
=> 5,
                Coi n::Dime =>
10,
                Coi n:: Qua -
rte r(s tate) => { // state
value is bind to the variable
                        pri -
ntl n!( " State quarter from
{:?}!", state);
                        25
                },
       }
}
```

if let

```
let some_u8_value = Some(0u8);
match some_u 8_value {
        Some(3) => printl n!
( " thr ee"),
       _ => (),
}
or
//if let concise for one pattern
if let Some(3) = some_u 8_value
{
        pri ntl n!( " thr ee");
}
```

vectors Vec<T>, vec! macro

```
let v: Vec<i3 2> = Vec::n ew();
```

```
let v = vec![1, 2, 3];
```

```
 let mut v = Vec::n ew();  v.push(5); v.push(6);
```

Inferring datatype from push?

fn function pointer

```
Use existing functions in place
of closure :
fn add_one(x: i32) -> i32 {
       x + 1
}
fn do_twi ce(f: fn(i32) -> i32,
arg: i32) -> i32 {
        f(arg) + f(arg)
}
fn main() {
       let answer = do_twi ce( -
add _one, 5);
        pri ntl n!( "The answer
is: {}", answer);
}
//return the closure
fn return s_c losure() ->
Box<Fn (i32) -> i32> {
        Box ::new(|x| x + 1)
}
```

annotate type when not initialized with macro vec! to create instance and hold data mut to update the vector. Rush infers the datatype from push.

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


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Vector iterator

```
let v = vec![100, 32, 57];
for i in &v {
        pri ntl n!( " {}", i);
}
//mutable vector and i to
derefe rence 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),
        Flo at( f64),
        Tex t(S tring),
}
let row = vec![
        Spr ead she etC -
ell ::I nt(3),
        Spr ead she etC -
ell ::T ext (St rin g:: fro -
m("b lue ")),
        Spr ead she etC -
ell ::F loa t(1 0.12),
];
```

String, str

str: string literals are stored in the binary program.
format: UTF8
OsString, OsStr, CString, and CStr are string variant libraries.

String, str (cont)

> //creating strings
let mut s = String::new();
let s = "initial contents".to_string(); //to_s-tring and from are matter of style
let s = String::from("initial contents");
let hello = String::from("नमस्ते"); //utf-8

String update and concatenation +/ format!

```
let mut s = String::from("foo");
s.push _st r("b ar"); // foobar
append , it takes slice so no
ownership transfer
let mut s = String ::f rom -
("lo ");
s.push ('l'); //char acter , lol
// concat enation +
let s1 = String ::f rom ("Hello,
");
let s2 = String ::f rom ("wo -
rld !");
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 => &s 2[..]
//more than two string, use
let s1 = String ::f rom ("ti -
c");
let s2 = String ::f rom ("ta -
c");
let s3 = String ::f rom ("to -
e");
let s = format !("{ }-{ }-{ }",
s1, s2, s3);
```

strings indexing

```
let s1 = String ::f rom ("he llo "
1[0];
```

error[E0277]: the trait bound `std::s tri -ng: :St ring: std::o ps: :In dex <{i nte ger }>` 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_B ACK TRACE=1 cargo run
```

Err(ref error) if error.kind() == ErrorKind::NotF
=>

unwrap

expect

```
let mut f = File:: ope n("h ell o.
t")?;
```

```
File:: ope n("h ell o.t xt") ?.r e
g(&mut s)?;
```

? for return type Result

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

Sponsored by Readable.com
Measure your website readability!
https://readable.com

Cargo in depth

```
//Cargo profiles
[profi le.dev]
opt-level = 0 //over riding defaults
[profi le.r el ease]
opt-level = 3
//re exporting API
pub use kinds: :Pr ima ryC olor;
pub use kinds: :Se con dar - yColor;
pub use utils: :mix;
cargo.io login: cargo login abcdef ghi jkl mno pqr stu vwx - yz0 12345
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 = HashMa p:: - new();
scores.in ser t(S tri ng: :fr - om( " Blu e"), 10);
scores.in ser t(S tri ng: :fr - om( " Yel low "), 50);
//creating a hash map from two vectors
let teams = vec![S tri ng: :fr - om( " Blu e"), String ::f rom - ("Ye llo w")];
let initia l_s cores = vec![10, 50];
let scores: HashMa p<_, _> = teams.i te r().zi p(i nit ial - _sc ore s.i ter ()).co lle ct();
//HashMap to get the desired type from collect .. strange way of specifying return value
```

HashMap<K, V> (cont)

```
> // _, _ 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> {
```

Generic data types <T> (cont)

```
>     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> {
    x: T,
    y: U,
}
//mixup
impl<T, U> Point<T, U> {
```


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Generic data types <T> (cont)

```
>     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 summar ize _au tho -
r(&self) -> String;
        fn summar ize (& self) ->
String {
                for mat !("(Read
more from {}...) ", self.s umm -
ari ze_ aut hor()) //default can
call other methods in the trait.
        }
}
pub struct NewsAr ticle {
        pub headline: String,
        pub location: String,
        pub author: String,
        pub content: String,
}
impl Summary for NewsAr ticle {
        fn summar ize (& self) ->
String {
```

trait (cont)

```
>         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.c-
ontent)
    }
}
//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_f unc tio n<T: Display +
Clone, U: Clone + Debug>(t: T,
u: U) -> i32 {
//where clause: with mutilple
trait bounds
fn some_f unc tio n<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;
        }
         pri ntl n!( "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
```


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

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 <Se lf: :It em>;
        // methods with default implem ent ations 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 {
                        pri - ntl n!( "hi number {} from the spawned thread !", i);
                        - std ::t hre ad: :sl eep (st - d:: tim e:: Dur ati on: :fr - om_ mil lis (1));
                }
        });
        han dle.jo in( ).u nwr - ap(); //wait to finish
let v = vec![1, 2, 3];
        let handle = thread ::s - paw n(move || { => to let capture take the ownership
                pri ntl n!( " - 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:: cha - nnel(); tx: transm itter, rx: receiver
  thr ead ::s paw n(move || {
                let val = String ::f rom ("hi ");
```


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

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::c lon e(&co unter); //clone
               let handle =
thread ::s paw n(move || {
                       let mut
num = counte r.l ock ().u nw -
rap();
                       *num +=
1;
               });
                han dle s.p -
ush (ha ndle);
       }
```

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 {
                pri ntl n!( " -
cal cul ating slowly..." );
                thr ead ::s -
lee p(D ura tio n:: fro m_s -
ecs (2));
               num
       };
// type inference
let exampl e_c losure = |x| x;
let s = exampl e_c los ure (St -
rin g:: fro m("h ell o"));
let n = exampl e_c los ure(5);
//error type inference only one
type
// memoiz ation or lazy evalua -
tions
impl<T> Cacher <T>
        where T: Fn(u32) -> u32
{
```

closures (cont)

```
>    fn new(calculation: T) -> Cacher<T> {
      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::n-
ew(|num| {
      println!("calculating slowly...");
      thread::sleep(Duration::from_secs(2));
      num
   });
//memoization
```


Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

unsafe

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

mutating static is unsafe

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_m angle] pub extern " C" fn call_from_c() { printl n!( "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;
let r2 = &mut num as *mut i32;
unsafe { //dere fer encing
        pri ntl n!( "r1 is: {}",
*r1);
        pri ntl n!( "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 refere nces. With raw pointers, can create mutable pointer and an immutable pointer to the same location, and change data through the mutable pointer, potent ially 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> {
        con text: &'c Contex -
t<' s>,
}
// lifetime bounds on references
to Generic Types
struct Static Ref <T: 'stati c>
( &' static T);
struct Ref<'a, T: 'a> (&'a T);
//Infe rence of Tait life times
```

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 + 'stati c>

Just as with the other bounds, this means that any implem entor of the Red trait that has references inside must have the same lifetime specified in the trait object bounds as those references

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com

Advanced Traits

```
pub trait Iterator {
       type Item; // place
holder
       fn next(&mut self) ->
Option <Se lf: :It em>;
}
//With Generic, needs to
annotate the type
//Default generic type
trait Add<RH S=S elf> { // RHS
is self type a = a + a
       type Output;
       fn add(self, rhs: RHS) ->
Self:: Output;
}
fn main() {
       let person = Human;
        Pil ot: :fl y(&pe rson);
//quality to avoid ambiguity
        Wiz ard ::f ly( &p -
erson);
        per son.fly();
}
Assosc iated functions:
//As no self, it can infer
<Dog as Animal >:: bab y_n ame()
//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::n-ew(|| 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 "नमस्ते".chars() {
        pri ntl n!( " {}", c);
}
न
म
स
ु
त
े
for b in "नमसु्ते".bytes() {
        pri ntl n!( " {}", b);
}
224
164
// --snip--
165
135
```

Cargo workspaces

```
[workspace]
members = [
        " add er",
]
├── Cargo.lock
├── Cargo.toml
├── add-one
│ ├── Cargo.toml
│ └── src
│ └── lib.rs
├── adder
│ ├── Cargo.toml
│ └── src
│ └── main.rs
└── target
[depen den cies]
add-one = { path = "../ add -
on e" } -- explicit
cargo run -p adder //to run
cargo install $HOME/.ca rgo/bin
cargo- som ething => cargo
something
```

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

Sponsored by **Readable.com**
Measure your website readability!
https://readable.com