## 

## 

version 1.0.7

# 

# Table of Contents

[Basic Types & Variables](#_4r7qhbze8gk5)

[Control Flow](#_tyjcwt)

[References, Ownership, and Borrowing](#_3dy6vkm)

[Pattern Matching](#_ny39k0mrv251)

[Iterators](#_4d34og8)

[Error Handling](#_2s8eyo1)

[Combinators](#_3rdcrjn)

[Multiple error types](#_26in1rg)

[Iterating over errors](#_lnxbz9)

[Generics, Traits, and Lifetimes](#_35nkun2)

[Functions, Function Pointers & Closures](#_1ksv4uv)

[Pointers](#_44sinio)

[Smart pointers](#_2jxsxqh)

[Packages, Crates, and Modules](#_z337ya)

**YouTube Channel:** [**https://www.youtube.com/c/LetsGetRusty**](https://www.youtube.com/c/LetsGetRusty)

## Basic Types & Variables

bool - Boolean

**Unsigned integers**

u8, u16, u32, u64, u128

**Signed integers**

i8, i16, i32, i64, i128

**Floating point numbers**

f32, f64

**Platform specific integers**

usize - Unsigned integer. Same number of bits as the platform's pointer type.

isize - Signed integer. Same number of bits as the platform's pointer type.

char - [Unicode scalar value](http://www.unicode.org/glossary/#unicode_scalar_value)

&str - String slice

String - Owned string

**Tuple**

| **let** coordinates = (82, 64); **let** score = ("Team A", 12) |
| --- |

**Array & Slice**

| *// Arrays must have a known length and all elements must be initialized* ***let*** *array = [1, 2, 3, 4, 5];* ***let*** *array2 = [0; 3]; // [0, 0, 0]*  *// Unlike arrays the length of a slice is determined at runtime* ***let*** *slice = &array[1 .. 3];* |
| --- |

**HashMap**

| **use** std::collections::HashMap;  **let** **mut** subs = HashMap::new(); subs.insert(String::from("LGR"), 100000); *// Insert key if it doesn't have a value* subs.entry("Golang Dojo".to\_owned()) .or\_insert(3); |
| --- |

**Struct**

| *// Definition* **struct** **User** {  username: String,  active: bool, }  *// Instantiation* **let** user1 = User {  username: String::from("bogdan"),  active: true, };  *// Tuple struct* **struct** **Color**(i32, i32, i32); **let** black = Color(0, 0, 0); |
| --- |

**Enum**

| *// Definition* **enum** **Command** {  Quit,  Move { x: i32, y: i32 },  Speak(String),  ChangeBGColor(i32, i32, i32), }  *// Instantiation* **let** msg1 = Command::Quit; **let** msg2 = Command::Move{ x: 1, y: 2 }; **let** msg3 = Command::Speak("Hi".to\_owned()); **let** msg4 = Command::ChangeBGColor(0, 0, 0); |
| --- |

**Constant**

| **const** MAX\_POINTS: u32 = 100\_000; |
| --- |

**Static variable**

| *// Unlike constants static variables are* *// stored in a dedicated memory location* *// and can be mutated.* **static** MAJOR\_VERSION: u32 = 1; **static** **mut** COUNTER: u32 = 0; |
| --- |

**Mutability**

| **let** **mut** x = 5; x = 6; |
| --- |

**Shadowing**

| **let** x = 5; **let** x = x \* 2; |
| --- |

**Type alias**

| *// `NanoSecond` is a new name for `u64`.* **type** **NanoSecond** = u64; |
| --- |

## Control Flow

**if and if let**

| **let** num = Some(22);  **if** num.is\_some() {  println!("number is: {}", num.unwrap());  }  *// match pattern and assign variable* **if** **let** Some(i) = num {  println!("number is: {}", i); } |
| --- |

**loop**

| **let** **mut** count = 0; **loop** {  count += 1;  **if** count == 5 {  **break**; *// Exit loop*  } } |
| --- |

**Nested loops & labels**

| 'outer: **loop** {  'inner: **loop** {  *// This breaks the inner loop*  **break**;  *// This breaks the outer loop*  **break** 'outer;  } } |
| --- |

**Returning from loops**

| **let** **mut** counter = 0;  **let** result = **loop** {  counter += 1;   **if** counter == 10 {  **break** counter;  } }; |
| --- |

**while and while let**

| **while** n < 101 {  n += 1; }  **let** **mut** optional = Some(0);  **while** **let** Some(i) = optional {  print!("{}", i); } |
| --- |

**for loop**

| **for** n **in** 1..101 {  println!("{}", n); }  **let** names = vec!["Bogdan", "Wallace"];  **for** name **in** names.iter() {  println!("{}", name); } |
| --- |

**match**

| **let** optional = Some(0);  **match** optional {  Some(i) => println!("{}", i),  None => println!("No value.") } |
| --- |

## References, Ownership, and Borrowing

**Ownership rules**

1. Each value in Rust has a variable that’s called its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

**Borrowing rules**

1. At any given time, you can have *either* one mutable reference *or* any number of immutable references.
2. References must always be valid.

**Creating references**

| **let** s1 = String::from("hello world!"); **let** s1\_ref = &s1; *// immutable reference*  **let** **mut** s2 = String::from("hello"); **let** s2\_ref = &**mut** s2; *// mutable reference*  s2\_ref.push\_str(" world!"); |
| --- |

**Copy, Move, and Clone**

| *// Simple values which implement the Copy trait are copied by value* **let** x = 5; **let** y = x;  println!("{}", x); *// x is still valid*  *// The string is moved to s2 and s1 is invalidated*  **let** s1 = String::from("Let's Get Rusty!"); **let** s2 = s1; *// Shallow copy a.k.a move*  println!("{}", s1); *// Error: s1 is invalid*  **let** s1 = String::from("Let's Get Rusty!"); **let** s2 = s1.clone(); *// Deep copy*  *// Valid because s1 isn't moved* println!("{}", s1); |
| --- |

**Ownership and functions**

| **fn** **main**() {  **let** x = 5;  takes\_copy(x); *// x is copied by value*   **let** s = String::from("Let’s Get Rusty!");  *// s is moved into the function*  takes\_ownership(s);    *// return value is moved into s1*  **let** s1 = gives\_ownership();  **let** s2 = String::from("LGR");  **let** s3 = takes\_and\_gives\_back(s2); }  **fn** **takes\_copy**(some\_integer: i32) {  println!("{}", some\_integer); }  **fn** **takes\_ownership**(some\_string: String) {  println!("{}", some\_string); } *// some\_string goes out of scope and drop is called. The backing memory is freed.*  **fn** **gives\_ownership**() -> String {  **let** some\_string = String::from("LGR");  some\_string }  **fn** **takes\_and\_gives\_back**(some\_string: String) -> String {  some\_string } |
| --- |

## Pattern Matching

**Basics**

| **let** x = 5;   **match** x {  *// matching literals*  1 => println!("one"),  *// matching multiple patterns*  2 | 3 => println!("two or three"),  *// matching ranges*  4..=9 => println!("within range"),  *// matching named variables*  x => println!("{}", x),  *// default case (ignores value)*  \_ => println!("default Case") } |
| --- |

**Destructuring**

| **struct** **Point** {  x: i32,  y: i32, }  **let** p = Point { x: 0, y: 7 };  **match** p {  Point { x, y: 0 } => {  println!("{}" , x);  },  Point { x, y } => {  println!("{} {}" , x, y);  }, }  **enum** **Shape** {  Rectangle { width: i32, height: i32 },  Circle(i32), }  **let** shape = Shape::Circle(10);  **match** shape {  Shape::Rectangle { x, y } => *//...*  Shape::Circle(radius) => *//...* } |
| --- |

**Ignoring values**

| **struct** **SemVer**(i32, i32, i32);  **let** version = SemVer(1, 32, 2);  **match** version {  SemVer(major, \_, \_) => {  println!("{}", major);  } }  **let** numbers = (2, 4, 8, 16, 32);  **match** numbers {  (first, .., last) => {  println!("{}, {}", first, last);  } } |
| --- |

**Match guards**

| **let** num = Some(4);  **match** num {  Some(x) **if** x < 5 => println!("less than five: {}", x),  Some(x) => println!("{}", x),  None => (), } |
| --- |

**@ bindings**

| **struct** **User** {  id: i32 }   **let** user = User { id: 5 };   **match** user {  User {  id: id\_variable @ 3..=7,  } => println!("id: {}", id\_variable),  User { id: 10..=12 } => {  println!("within range");  },  User { id } => println!("id: {}", id), } |
| --- |

## Iterators

**Usage**

| *// Methods that consume iterators* **let** v1 = vec![1, 2, 3]; **let** v1\_iter = v1.iter(); **let** total: i32 = v1\_iter.sum();  *// Methods that produce new iterators* **let** v1: Vec<i32> = vec![1, 2, 3]; **let** iter = v1.iter().map(|x| x + 1);  *// Turning iterators into a collection* **let** v1: Vec<i32> = vec![1, 2, 3]; **let** v2: Vec<\_> = v1.iter().map(|x| x + 1).collect(); |
| --- |

**Implementing the Iterator trait**

| **struct** **Counter** {  count: u32, }   **impl** Counter {  **fn** **new**() -> Counter {  Counter { count: 0 }  } }  **impl** Iterator **for** Counter {  **type** **Item** = u32;    **fn** **next**(&**mut** **self**) -> Option<Self::Item>  {  **if** **self**.count < 5 {  **self**.count += 1;  Some(**self**.count)  } **else** {  None  }  } } |
| --- |

## Error Handling

**Throw unrecoverable error**

| panic!("Critical error! Exiting!"); |
| --- |

**Option enum**

| **fn** **get\_user\_id**(name: &str) -> Option<u32> {  **if** database.user\_exists(name) {  **return** Some(database.get\_id(name))  }    None } |
| --- |

**Result enum**

| **fn** **get\_user**(id: u32) -> Result<User, Error> {  **if** is\_logged\_in\_as(id) {  **return** Ok(get\_user\_object(id))  }    Err(Error { msg: "not logged in" }) } |
| --- |

**? operator**

| **fn** **get\_salary**(db: Database, id: i32) -> Option<u32> {  Some(db.get\_user(id)?.get\_job()?.salary) }  **fn** **connect**(db: Database) -> Result<Connection, Error> {  **let** conn =  db.get\_active\_instance()?.connect()?;  Ok(conn) } |
| --- |

### 

### Combinators

**.map**

| **let** some\_string = Some("LGR".to\_owned());  **let** some\_len = some\_string.map(|s| s.len());  **struct** **Error** { msg: String } **struct** **User** { name: String }  **let** string\_result: Result<String, Error> = Ok("Bogdan".to\_owned());  **let** user\_result: Result<User, Error> =   string\_result.map(|name| {  User { name }  }); |
| --- |

**.and\_then**

| **let** vec = Some(vec![1, 2, 3]); **let** first\_element = vec.and\_then(  |vec| vec.into\_iter().next() );  **let** string\_result: Result<&'static str, \_> = Ok("5"); **let** number\_result =   string\_result  .and\_then(|s| s.parse::<u32>()); |
| --- |

### Multiple error types

**Define custom error type**

| **type** **Result**<T> = std::result::Result<T, CustomError>;  **#[derive(Debug, Clone)]** **struct** **CustomError**;  **impl** fmt::Display **for** CustomError {  **fn** **fmt**(&**self**, f: &**mut** fmt::Formatter) -> fmt::Result {  write!(f, "custom error message")  } } |
| --- |

**Boxing errors**

| **use** std::error;  **type** **Result**<T> = std::result::Result<T, Box<dyn error::Error>>; |
| --- |

### Iterating over errors

**Ignore failed items with filter\_map()**

| **let** strings = vec!["LGR", "22", "7"]; **let** numbers: Vec<\_> = strings  .into\_iter()  .filter\_map(|s| s.parse::<i32>().ok())  .collect(); |
| --- |

**Fail the entire operation with collect()**

| **let** strings = vec!["LGR", "22", "7"];  **let** numbers: Result<Vec<\_>, \_> = strings  .into\_iter()  .map(|s| s.parse::<i32>())  .collect(); |
| --- |

**Collect all valid values and failures with partition()**

| **let** strings = vec!["LGR", "22", "7"];  **let** (numbers, errors): (Vec<\_>, Vec<\_>) = strings  .into\_iter()  .map(|s| s.parse::<i32>())  .partition(Result::is\_ok);  **let** numbers: Vec<\_> = numbers  .into\_iter()  .map(Result::unwrap)  .collect();  **let** errors: Vec<\_> = errors  .into\_iter()  .map(Result::unwrap\_err)  .collect(); |
| --- |

## Generics, Traits, and Lifetimes

**Using generics**

| **struct** **Point**<T, U> {  x: T,  y: U, }  **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,  }  } } |
| --- |

**Defining traits**

| **trait** **Animal** {  **fn** **new**(name: &'static str) -> **Self**;  **fn** **noise**(&**self**) -> &'static str { "" } }  **struct** **Dog** { name: &'static str }  **impl** Dog {  **fn** **fetch**() { *// ...* } }  **impl** Animal **for** Dog {  **fn** **new**(name: &'static str) -> Dog {  Dog { name: name }  }   **fn** **noise**(&**self**) -> &'static str {  "woof!"  } } |
| --- |

**Default implementations with Derive**

| *// A tuple struct that can be printed* **#[derive(Debug)]** **struct** **Inches**(i32); |
| --- |

**Trait bounds**

| **fn** **largest**<T: PartialOrd + Copy>(list: &[T]) -> T {  **let** **mut** largest = list[0];   **for** &item **in** list {  **if** item > largest {  largest = item;  }  }   largest } |
| --- |

**impl trait**

| **fn** **make\_adder\_function**(y: i32) -> **impl** Fn(i32) -> i32 {  **let** closure = **move** |x: i32| { x + y };  closure } |
| --- |

**Trait objects**

| **pub** **struct** **Screen** {  **pub** components: Vec<Box<dyn Draw>>, } |
| --- |

**Operator overloading**

| **use** std::ops::Add;  **#[derive(Debug, Copy, Clone, PartialEq)]** **struct** **Point** {  x: i32,  y: i32, }  **impl** Add **for** Point {  **type** **Output** = Point;   **fn** **add**(**self**, other: Point) -> Point {  Point {  x: **self**.x + other.x,  y: **self**.y + other.y,  }  } } |
| --- |

**Supertraits**

| **use** std::fmt;  **trait** **Log**: fmt::Display {  **fn** **log**(&**self**) {  **let** output = **self**.to\_string();  println!("Logging: {}", output);  } } |
| --- |

**Lifetimes in function signatures**

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

**Lifetimes in struct definitions**

| **struct** **User**<'a> {  full\_name: &'a str, } |
| --- |

**Static lifetimes**

| **let** s: &'static str = "Let’s Get Rusty!"; |
| --- |

## Functions, Function Pointers & Closures

**Associated functions and methods**

| **struct** **Point** { x: i32, y: i32, }  **impl** Point {  *// Associated function*  **fn** **new**(x: i32, y: i32) -> Point {  Point { x: x, y: y }  }   *// Method*  **fn** **getX**(&**self**) -> i32 { **self**.x } } |
| --- |

**Function pointers**

| **fn** **do\_twice**(f: **fn**(i32) -> i32, arg: i32) -> i32 {  f(arg) + f(arg) } |
| --- |

**Creating closures**

| **let** add\_one = |num: u32| -> u32 {  num + 1 }; |
| --- |

**Returning closures**

| **fn** **add\_one**() -> **impl** Fn(i32) -> i32 {  |x| x + 1 }  **fn** **add\_or\_subtract**(x: i32) -> Box<dyn Fn(i32) -> i32> {  **if** x > 10 {  Box::new(**move** |y| y + x)  } **else** {  Box::new(**move** |y| y - x)  } } |
| --- |

**Closure traits**

* **FnOnce** - consumes the variables it captures from its enclosing scope.
* **FnMut** - mutably borrows values from its enclosing scope.
* **Fn** - immutably borrows values from its enclosing scope.

**Store closure in struct**

| **struct** **Cacher**<T> **where**  T: Fn(u32) -> u32, {  calculation: T,  value: Option<u32>, } |
| --- |

**Function that accepts closure or function pointer**

| **fn** **do\_twice**<T>(f: T, x: i32) -> i32  **where** T: Fn(i32) -> i32 {  f(x) + f(x) } |
| --- |

## Pointers

**References**

| **let** **mut** num = 5; **let** r1 = &num; *// immutable reference* **let** r2 = &**mut** num; *// mutable reference* |
| --- |

**Raw pointers**

| **let** **mut** num = 5; *// immutable raw pointer* **let** r1 = &num **as** \***const** i32; *// mutable raw pointer* **let** r2 = &**mut** num **as** \***mut** i32; |
| --- |

### Smart pointers

**Box<T>** - for allocating values on the heap

| **let** b = Box::new(5); |
| --- |

**Rc<T>** -multiple ownership with reference counting

| **let** a = Rc::new(5); **let** b = Rc::clone(&a); |
| --- |

**Ref<T>, RefMut<T>, and RefCell<T>** - enforce borrowing rules at runtime instead of compile time.

| **let** num = 5; **let** r1 = RefCell::new(5); *// Ref - immutable borrow* **let** r2 = r1.borrow(); *// RefMut - mutable borrow* **let** r3 = r1.borrow\_mut(); *// RefMut - second mutable borrow* **let** r4 = r1.borrow\_mut(); |
| --- |

**Multiple owners of mutable data**

| **let** x = Rc::new(RefCell::new(5)); |
| --- |

## Packages, Crates, and Modules

**Definitions**

* **Packages** - A Cargo feature that lets you build, test, and share crates.
* **Crates** - A tree of modules that produces a library or executable.
* **Modules** and **use** - Let you control the organization, scope, and privacy of paths.
* **Paths** - A way of naming an item, such as a struct, function, or module.

**Creating a new package with a binary crate**

| $ cargo new my-project |
| --- |

**Creating a new package with a library crate**

| $ cargo new my-project --lib |
| --- |

**Defining and using modules**

| **fn** **some\_function**() {}  **mod** outer\_module { *// private module*  **pub** **mod** inner\_module { *// public module*  **pub** **fn** **inner\_public\_function**() {  super::super::some\_function();  }   **fn** **inner\_private\_function**() {}  } }  **fn** **main**() {  *// absolute path*  crate::outer\_module::  inner\_module::inner\_public\_function();   *// relative path path*  outer\_module::  inner\_module::inner\_public\_function();   *// bringing path into scope*  **use** outer\_module::inner\_module;  inner\_module::inner\_public\_function(); } |
| --- |

**Renaming with *as* keyword**

| **use** std::fmt::Result; **use** std::io::Result **as** IoResult; |
| --- |

**Re-exporting with *pub use***

| **mod** outer\_module {  **pub** **mod** inner\_module {  **pub** **fn** **inner\_public\_function**() {}  } }  **pub** **use** crate::outer\_module::inner\_module; |
| --- |

**Defining modules in separate files**

| *// src/lib.rs* **mod** my\_module;  **pub** **fn** **some\_function**() {  my\_module::my\_function(); }  *// src/my\_module.rs* **pub** **fn** **my\_function**() {} |
| --- |