Rust, a code snippet introduction

*(P) **Codever** is an open source bookmarks and snippets manager for developers & co. See our How To guides to help you get started. Public bookmarks repos on Github* ⭐🙏

A basic introduction in Rust, with code snippets taken from the The Rust Programming Language and Rust by Example books, to present in a 40 minutes slot.

> *You can execute most of the snippets directly on Rust Playground*

- Hello world!
- `Println` macro
- Variables
  - Mutability
  - Variables Scope
- Primitives
  - Scalar Types
  - Compound Types
  - Snippet example
  - Tuples
- Functions
- Ownership
- Stack & Heap
  - Ownership rules

Hello world!

COP

```
// This is the main function
fn main() {
    // Statements here are executed when the compiled binary is called

    // Print text to the console
    println!("Hello World!");
}
```

Use the rust compiler `rustc` to compile the code

COP

```
rustc hello.rs
```

`rustc` will produce a `hello` binary that can be executed.

COP

```
./hello
```

COP

```
fn main() {
    // In general, the `{}` will be automatically replaced with any
    // arguments. These will be stringified.
    println!("{} days", 31);

    // Without a suffix, 31 becomes an i32. You can change what type 31 is
     / by providing a suffix. The number 31i64 for example has the type i64.

     / There are various optional patterns this works with. Positional
     / arguments can be used.
    println!("{0}, this is {1}. {1}, this is {0}", "Alice", "Bob");

    // As can named arguments.
    println!("{subject} {verb} {object}",
             object="the lazy dog",
             subject="the quick brown fox",
             verb="jumps over");

    // Special formatting can be specified after a `:`.
    println!("{} of {:b} people know binary, the other half doesn't", 1, 2);

    // You can right-align text with a specified width. This will output
    // "     1". 5 white spaces and a "1".
    println!("{number:>width$}", number=1, width=6);

    // You can pad numbers with extra zeroes. This will output "000001".
    println!("{number:0>width$}", number=1, width=6);

    // Rust even checks to make sure the correct number of arguments are
    // used.
    println!("My name is {0}, {1} {0}", "Bond");
    // FIXME ^ Add the missing argument: "James"
}
```

Variables

COP

```
const STARTING_MISSILES: i32 = 8;
const READY_AMOUNT: i32 = 2;

fn main() {
    let mut missiles: i32 = STARTING_MISSILES;
    let ready: i32 = READY_AMOUNT;
    println!("Firing {} of my {} missiles...", ready, missiles);
    missiles = missiles - ready;
    println!("{} missiles left", missiles);
}
```

Mutability

Variable bindings are **immutable by default**, but this can be overridden using the `mut` modifier.

COP

```
fn main() {
    let _immutable_binding = 1;
    let mut mutable_binding = 1;

    println!("Before mutation: {}", mutable_binding);

    // Ok
    mutable_binding += 1;

    println!("After mutation: {}", mutable_binding);

    // Error!
```

}

Variables Scope

COP

```
fn main() {
    let s = "world";

        let s = "hello";
        println!("{}", s);

    println!("{}", s);
}
```

> *Try to guess the output*

Primitives

Scalar Types

- signed integers: `i8`, `i16`, `i32`, `i64`, `i128` and `isize` (pointer size)
- unsigned integers: `u8`, `u16`, `u32`, `u64`, `u128` and `usize` (pointer size)
- floating point: `f32`, `f64`
- `char` Unicode scalar values like `'a'`, `'α'` and `'∞'` (4 bytes each)
- `bool` either `true` or `false`
- and the unit type `()`, whose only possible value is an empty tuple: `()`

Compound Types

Snippet example

COP

```
fn main() {
    // Variables can be type annotated.
    let logical: bool = true;

    let a_float: f64 = 1.0;  // Regular annotation
    let an_integer   = 5i32; // Suffix annotation

    // Or a default will be used.
    let default_float   = 3.0; // `f64`
    let default_integer = 7;   // `i32`

    // A type can also be inferred from context
    let mut inferred_type = 12; // Type i64 is inferred from another line
    inferred_type = 4294967296i64;

    // A mutable variable's value can be changed.
    let mut mutable = 12; // Mutable `i32`
    mutable = 21;

    // Error! The type of a variable can't be changed.
    mutable = true;

    // Variables can be overwritten with shadowing.
    let mutable = true;
}
```

Tuples

A tuple is a collection of values of different types. Tuples are constructed using parentheses `()`, and each tuple itself is a value with type signature number of values.

COP

```
fn main() {
    let user = ("Adrian", 38);
    println!("User {} is {} years old", user.0, user.1);

     / tuples within tuples
     et employee = (("Adrian", 38), "die Mobiliar");
     rintln!("User {} is {} years old and works for {}", employee.0.1, employee.0.1,
```

Functions

COP

```
fn main() {
    println!("Sum result {}", sum(5,8));
}

fn sum(x:i32, y:i32) -> i32 {
    let z = {
         x + y
    };
   z
}
```

Ownership

- both parts of memory, but structured differently
- **Stack** (organized & fast)
  - data is stored **in order**
  - all data stored on the stack must have a **known, fixed** size
  - LIFO (Last In First Out) - *push* data on the stack and *pop* data of the stack

  Heap (less organized & slow)
  - stores data with an **unkonwn/variable size** at **compile time** or a **size that might change**
  - **unordered** storage
  - data is *allocated* and a *pointer* (known, fixed size *stored on the stack*) the location/space on the heap is returned

Ownership rules

Ownership is Rust's most unique feature, and it enables Rust to make memory safety guarantees **without needing a garbage collector**.

1. Each value in Rust has a variable that's called its *owner*.
2. Only one owner of a value
3. When the owner goes out of scope, the value will be dropped.

Take ownership - snippet

COP

```
#[allow(unused_variables)]
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    println!("{}, world!", s1);
}
```

Another **move** situation:

COP

```
fn main() {
    let s1 = String::from("hello");
    do_stuff_with_string(s1);

    println!("{}, world!", s1);



     ow(unused_variables)]
     _stuff_with_string(s: String) {
    //do_stuff
}
```

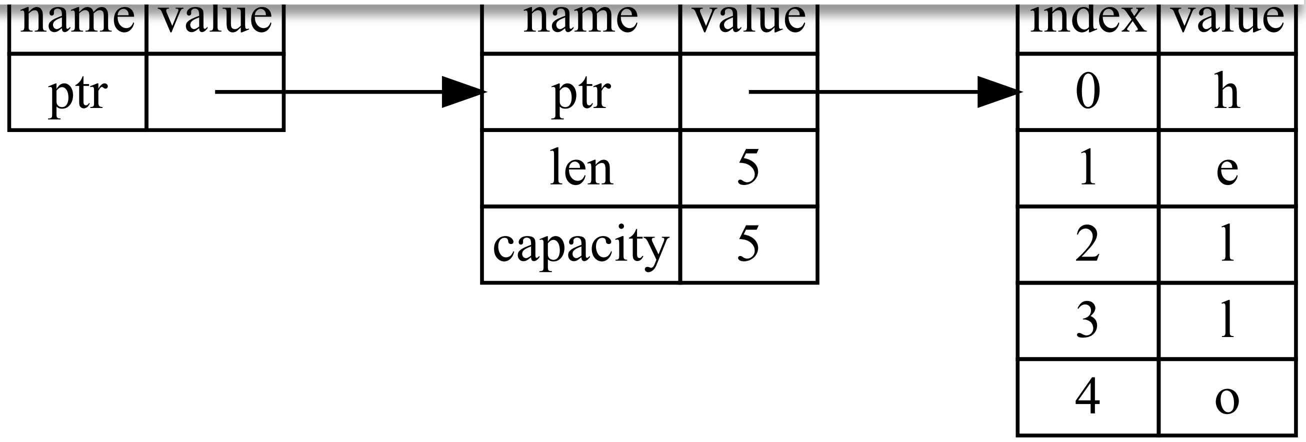
Fix

COP

```
fn main() {
    let s1 = String::from("hello");
    do_stuff_with_string(&s1);

    println!("{}, world!", s1);
}

#[allow(unused_variables)]
fn do_stuff_with_string(s: &String) {
    //do_stuff
}
```



ructs

> A *struct*, or *structure*, is a custom data type that lets you name and package together multiple related values that make up a meaningful group. If you're familiar with an object-oriented language, a *struct* is like an object's data attributes.

COP

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

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
```

```
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

Note:

- to define the function within the context of `Rectangle`, we start an `impl` (implementation) block.
- we've chosen `&self` here for the same reason we used `&Rectangle` in the function version: we don't want to take ownership, and we just want to read the data in the struct, not write to it.

Traits

A *trait* tells the Rust compiler about functionality a particular type has and can share with other types. We can use traits to define shared behavior in an abstract way. We can use trait bounds to specify that a generic can be any type that has certain behavior.

> *Note: Traits are similar to a feature often called interfaces in other languages, although with some differences.*

COP

```
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}
```

```
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
     ub reply: bool,
     ub retweet: bool,


impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}



fn main() {
    let tweet = Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    };

    println!("1 new tweet: {}", tweet.summarize());
}
```

Enums

```
    Alabama,
    Alaska,
    // --snip--
}

enum Coin {
     enny,
     ickel,
     ime,
     uarter(UsState),


fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}

fn main() {
    let value = value_in_cents(Coin::Quarter(UsState::Alaska));
    println!("{}", value);
}
```

Control flow

- no brackets in if conditions

COP

```
} else if num == 4 {
    msg = "four";
} else {
    msg = "other";
}
```

ı be rewritten to(`if` is **an expression** not a statement):

COP

```
 if num == 5 {
 five"
} else if num == 4 {
    "four"
} else {
    "other"
}; //Note the semicolon at the end
//all the branches must return the same type (rust is strongly typed)
```

Unconditional loop

COP

```
loop {
    break;
}
```

COP

```
'outer_loop: loop { //define loop label "tick outer_loop"
    loop {
        loop {
            break 'outer_loop;
        }
    }
}
```

COP

```
}
```

syntachtic sugar to

COP

```
loop {
    if !must_evaluate_to_bool_expresion() { break }
    /do stuff
```

`do while`, but

COP

```
loop {
    //do stuff
    if !must_evaluate_to_bool_expresion() { break }
}
```

`for` iterates over `iterable` values

COP

```
for num in [1, 2, 3].iter() {
    // do stuff with num
}
```

COP

```
let array = [(1,2), (3,4)];
for (x, y) in array.iter() { //destructuring
    // do stuff with x and y // x, y are local
}
```

Ranges

COP

```
}

for num in 0..=50 { //both start and end inclusive
    // do stuff with num
}
```

ollections

ctors

COP

```
fn main() {

/*
    let mut v = Vec::new();

    v.push(1);
    v.push(2);
    v.push(3);
    v.push(4);
    v.push(5);
*/

    let v = vec![1, 2, 3, 4, 5];

    let third: &i32 = &v[2];
    println!("The third element is {}", third);

    match v.get(2) {
        Some(third) => println!("The third element is {}", third),
        None => println!("There is no third element."),
    }
```

```
    //let does_not_exist = v.get(100);
}
```

Hashmaps

COP

```
fn main() {
     se std::collections::HashMap;

     et mut scores = HashMap::new();
     cores.insert(String::from("Blue"), 10);

    scores.entry(String::from("Yellow")).or_insert(50);
    scores.entry(String::from("Blue")).or_insert(50);

    println!("{:?}", scores);
}
```

Generics

COP

```
struct Point<T> {
    x: T,
    y: T,
}

#[allow(unused_variables)]
fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };

    let wont_work = Point { x: 5, y: 4.0 }; // Try fix me
```

Rust Module System

Rust has a number of features that allow you to manage your code's organization, including which details are exposed, which details are private, and what names are in each scope in your programs. These features, sometimes collectively referred to as the *module system*, include:

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

COP

```
crate
 └── front_of_house
     ├── hosting
     │   ├── add_to_waitlist
     │   └── seat_at_table
     └── serving
         ├── take_order
         ├── serve_order
         └── take_payment
```

Subscribe to our newsletter for more code resources and news

RUST

BY ADRIAN MATEI (AKA ADIXCHEN)

LIKE TWEET

Life force expressing itself as a coding capable human being

Follow @adixchen

Follow @CodepediaOrg

Read More

routerLink with query params in Angular html template

routerLink with query params in Angular html template code snippet Continue reading

---

How to not select attribute in mongoose schema
Published on April 21, 2023

Calculate life span and age in years in javascript/typescript
Published on March 31, 2023