Struct

Structs and enums are the building blocks for creating new types in your program's domain to take full advantage of Rust's compile time type checking.

Structure Definition

- A structure is defined as a group of *fields*.

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

Structure Creation and Accessing Fields

- Create an instance by specifying key: value pair for each field.
- Order of the fields doesn’t matter.
- To get a specific value from a struct, use dot notation like `user1.email`.

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

let email = user1.email;
```

Mutable Structure Instance

- In order to mutate a structure instance, the entire instance must be mutable (**inherited mutability**); Rust doesn’t allow us to mark only certain fields as mutable.

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

user1.email = String::from("anotheremail@example.com");
```

Using the Field Init Shorthand

```
fn build_user(email: String, username: String) -> User {
    User {
        email,    // values omitted because fields and parameters
        username, // have the same names.
        active: true,
        sign_in_count: 1,
    }
}
```

Creating Instances From Other Instances With **Struct Update Syntax**

- The syntax `..` specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

```
let user1 = User { … }

let user2 = User {
    active: user1.active,
    username: user1.username,
    email: String::from("another@example.com"),
    sign_in_count: user1.sign_in_count,
};
```

```
let user2 = User {
    email: String::from("another@example.com"),
    ..user1   // must come last to be effective
};
```

*Since **move** semantics applied, user1 is no longer available after user2 is created.*

Tuple Structs

- Tuple structs are useful when you want to give the whole tuple a name and make the tuple a different type from other tuples.

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

fn main() {
    let black = Color(0, 0, 0);
    let origin = Point(0, 0, 0);
}
```

Note that the black and origin values are different types

Unit-Like Structs Without Any Fields

- You can also define structs that don’t have any fields!
- Useful when you need to implement a trait on some type but don’t have any data that you want to store in the type itself.

```
struct AlwaysEqual;

fn main() {
    let subject = AlwaysEqual; // No need for curly brackets or parentheses!
}
```

[Very Hard!!!] Ownership of Struct Data

- This code doesn't compile.
- Guess why?

```
struct User {
    active: bool,
    username: &str,
    email: &str,
    sign_in_count: u64,
}

fn main() {
    let user1 = User {
        email: "someone@example.com",
        username: "someusername123",
        active: true,
        sign_in_count: 1,
    };
}
```

Newtype Pattern with Single Element Tuple Struct

- The simplest use of the **newtype pattern** is to indicate additional semantics for a type, over and above its normal behaviour.

```
struct Username(String);

struct Password(String);

fn login(username: Username, password: Password) {
    // ...
}
```

- The other common, but more subtle, scenario that requires the newtype pattern revolves around Rust's **orphan rule**.

Newtype Limitations

- Every operation that involves the newtype needs to forward to the inner type.
- Any trait implementations on the inner type are lost.

Associated Functions and Methods

We can associate some behaviors to a specific type or type instances like static methods and instance methods in OO languages.

- **Associated Functions**
  - Associated functions are functions that are defined on a type generally
  - Corresponds to *static methods* in OO languages
- **Methods**
  - Methods are associated functions that are called on a particular instance of a type.
  - Corresponds to *instance methods* in OO languages

Associated Functions

```
struct Point { x : f64, y: f64 }
```

- We may have one or more implementation blocks for a given type.

```
// Implementation block, all `Point` associated functions & methods go in here
impl Point {
    // This is an "associated function" because this function is associated with
    // a particular type, that is, Point.
    //
    // Associated functions don't need to be called with an instance.
    // These functions are generally used like constructors.
    fn origin() -> Point {
        Point { x: 0.0, y: 0.0 }
    }

    // Another associated function acting as constructors:
    fn new(x: f64, y: f64) -> Point {
        Point { x: x, y: y }
    }
}
```

Methods

```
struct Rectangle { p1: Point, p2: Point }
```

```
impl Rectangle {
    // `&self` is sugar for `self: &Self`, where `Self` is the type of the
    // caller object. In this case `Self` = `Rectangle`
    fn area(&self) -> f64 {
        // `self` gives access to the struct fields via the dot operator
        let Point { x: x1, y: y1 } = self.p1;
        let Point { x: x2, y: y2 } = self.p2;

        ((x1 - x2) * (y1 - y2)).abs() // `abs` is a `f64` method
    }

    // This method requires the caller object to be mutable
    // `&mut self` is sugar for `self: &mut Self`
    fn translate(&mut self, x: f64, y: f64) {
        self.p1.x += x; self.p2.x += x;
        self.p1.y += y; self.p2.y += y;
    }
}
```

Calling Associated Functions and Methods

```
let rectangle = Rectangle {
    // Associated functions are called using double colons
    p1: Point::origin(), p2: Point::new(3.0, 4.0),
};

// Methods are called using the dot operator
// Note that the first argument `&self` is implicitly passed, i.e.
// `rectangle.perimeter()` === `Rectangle::perimeter(&rectangle)`
println!("Rectangle perimeter: {}", rectangle.perimeter());
println!("Rectangle area: {}", rectangle.area());

// Error! `rectangle` is immutable, but this method requires a mutable object
rectangle.translate(1.0, 0.0);

let mut square = Rectangle {
    p1: Point::origin(), p2: Point::new(1.0, 1.0),
};
```

Enums and Pattern Matching

Enumerations (or Enums)

- Enums allow you to define a type by enumerating its possible “variants”, i.e., a possible set of values.

```
// IpAddrKind is now a custom data type
enum IpAddrKind {
    V4,   // unit-like variant
    V6,
}

// Create instances of each of the two variants of IpAddrKind.
let four = IpAddrKind::V4;
let six = IpAddrKind::V6;

// The `route` function is defined to take an `IpAddrKind` enum.
fn route(ip_kind: IpAddrKind) { … }

route(IpAddrKind::V4);
route(IpAddrKind::V6);
```

Each variant acts as a constructor.

enum can have variants with different types

- The important detail is that each enum variant can have data to go along with it.

```
enum Animal {
    // Enum constructors can have either named or unnamed fields:
    Dog(String, f64),   // tuple-like
    Cat {               // struct-like
        name: String,
        weight: f64,
    },
}

let mut a: Animal = Animal::Dog("Cocoa".to_string(), 37.2);
a = Animal::Cat { name: "Spotty".to_string(), weight: 2.7 };
```

Discriminant

- Each enum instance has a *discriminant* which is an integer associated to it.

```
enum Foo {
    Bar,  // 0 - starts at 0 by default
    Baz,  // 1
    Quux, // 2
}

let baz_discriminant = Foo::Baz as isize;
assert_eq!(baz_discriminant, 1);

enum Fruit {
    Apple = -1,     // -1
    Banana = 42,    // 42
    Orange,         // 43
}
```

If an enum has only unit variants, then the numeric value of the discriminant can be accessed with an [`as`] cast:

Discriminant (cont'd)

```
#[derive(Debug)]
enum Fruit { Apple, Banana, Orange }

impl From<u8> for Fruit {
    fn from(discriminant: u8) -> Self {
        match discriminant {
            0 => Fruit::Apple,
            1 => Fruit::Banana,
            2 => Fruit::Orange,
            _ => unreachable!(),
        }
    }
}
```

```
fn to_fruit(ordinal: u8) -> Fruit
{
    Fruit::from(ordinal)
}

for ordinal in 0..3 {
    let fruit = to_fruit(ordinal);
    println!("fruit: {fruit:?}");
}
```

Option<T> enum

- "Null References: The Billion Dollar Mistake," Tony Hoare
- Rust doesn't have the null feature that many other languages have.
- `Option` is an **enum** defined by the standard library.

```
enum Option<T> {
    None,
    Some(T),
}
```

Option type encodes the very common scenario in which a value could be something or it could be nothing.

```
let some_number: Option<i32> = Some(5);
let some_char: Option<char> = Some('e');

let absent_number: Option<i32> = None; // explicitly set to None required
```

Find the square of the second even number which is greater than 11.

```
let vs = [1, 11, 16, 7, 4, 15, 6, 14, 9];
```

```
fn find(vs: &[i32]) -> Option<i32> {
    let mut count = 0;
    for v in vs {
        if *v % 2 == 0 && *v > 11 {
            count += 1;
            if count == 2 {
                return Some(v * v);
            }
        }
    }
    None
}

assert_eq!(find(&vs), Some(196));
```

```
fn find(vs: &[i32]) -> Option<i32> {
    vs.iter()
        .filter(|&v| *v % 2 == 0 && *v > 11)
        .skip(1)
        .map(|v| v * v)
        .next()
}

assert_eq!(find(&vs), Some(196));
```

Pattern Matching

C++ Switch vs. Rust Match

C++ switch statement

```
int whatIsThis = 10;
switch (whatIsThis) {
case 8:
case 10:
  doSomething(); break;
case 12:
  doSomethingElse(); break;
default:
  doDefault();
}
```

Rust match expression

```
let whatIsThis = 8;
match whatIsThis {
    8 | 10 => do_something(),
    12     => do_something_else(),
    _      => do_default(),
}
```

Pattern Matching
a.k.a.
Switch on steroids

Slide from Karel Smutný

Pattern Matching
a.k.a.
Batman's toolbelt

Given the following code …

```
#[derive(Debug, Clone)]
enum Character {
  Civilian {
    name: String,
    wealth: Wealth,
  },
  SuperHero {
    name: String,
    powers: Vec<Power>,
    alterEgo: Option<Box<Character>>,
  },
}
use Character::*;
```

```
let tonyStark = Character::Civilian {
    name: "Tony Stark".to_string(),
    wealth: 1000000000.0,
};

let ironMan = Character::SuperHero {
    name: "Iron Man".to_string(),
    powers: vec![100, 200, 300],
    alterEgo: Some(Box::new(tonyStark)),
};

let unknownPerson: Character =
    get_Character();
```

The Problem

What are the super powers of an unknown person if it is a super hero who’s alter ego is Tony Stark?

Java

```java
if (unknownPerson instanceof SuperHero) {
  final SuperHero hero = (SuperHero) unknownPerson;
  if (hero.alterEgo.equals(tonyStark)) {
    return hero.powers;
  } else {
    return null;
  }
} else {
  return null;
}
```

Rust

```rust
match unknownPerson {
    SuperHero {
        name: _,
        powers,
        alterEgo: tonyStark,
    } => Some(powers),
    _ => None,
};
```

How cool is that?

Pattern Matching

```
match expression {
    pattern1 => expression1,
    pattern2 => expression2,
    pattern3 => expression3,
}
```

Each “pattern => expression” pair is called an “arm”

```
enum Coin { Penny, Nickel, Dime, Quarter, }

fn value_in_cents(coin: Coin) -> u8 {

    match coin {
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }

}
```

Pattern matching should be exhaustive

- Matches in Rust are exhaustive. The arms’ patterns must cover all possibilities.

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    // Error: non-exhaustive patterns: `None` not covered
    match x {
        Some(i) => Some(i + 1),
    }
}
```

```
Error: non-exhaustive patterns: `None` not covered
```

Catch All Pattern (*aka* Wildcard Pattern)

- Note that we have to ***put the catch-all arm last*** because the patterns are *evaluated in order*.
- The "_" pattern will match any value.

By putting it last, the "_" will match all the possible cases that aren't specified before it.

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => (), // _ is the placeholder pattern
}
```

Multiple Patterns

```
let x = get_int();

match x {
    1 | 2 => println!("one or two"),
    3 => println!("three"),
    _ => println!("anything"),
}
```

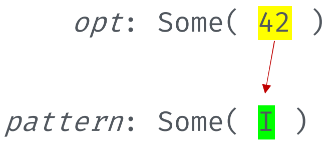
Matching Ranges of Values with .. or ..=

```
let x = get_int();

match x {
    1..=5 => println!("one through five"),
    _ => println!("something else"),
}

let x = get_char();

match x {
    'a'..='j' => println!("early ASCII letter"),
    'k'..='z' => println!("late ASCII letter"),
    _ => println!("something else"),
}
```

Patterns that Bind to Values

- Match arms can bind to the parts of the values that match the pattern.

```
fn plus_one(opt: Option<i32>) -> Option<i32> {
    match opt {
        None => None,
        Some(i) => {
            println!("current value = {i}");
            Some(i + 1)
        }
    }
}
```

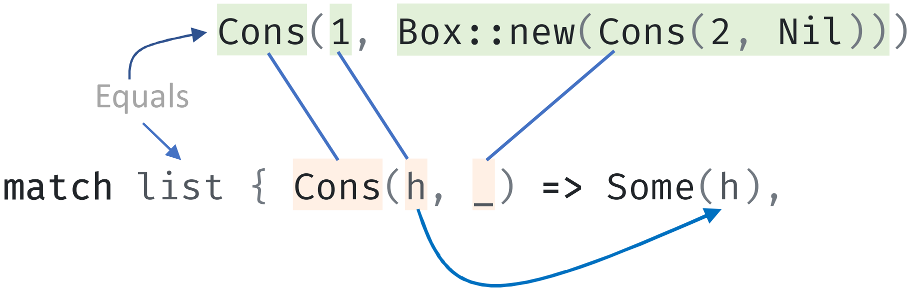


opt: Some(42)

pattern: Some(I)

Destructuring to Break apart Values

```
enum List { Nil, Cons(i32, Box<List>) }
use List::{Cons, Nil};

let list = Cons(1, Box::new(Cons(2, Nil)));

let maybe_head = match list {
    Cons(h, _) => Some(h),
    _ => None,
};
```



```
Cons(1, Box::new(Cons(2, Nil)))
```

Equals

```
match list { Cons(h, _) => Some(h),
```

Destructuring Structs

```
struct Point { x: i32, y: i32 }

let p = Point { x: 0, y: 7 };
let Point { x: a, y: b } = p;
// Shorthand if the variables and fields have the same name:
let Point { x, y } = p;

match p {
    Point { x, y: 0 } => println!("On the x axis at {x}"),
    Point { x: 0, y } => println!("On the y axis at {y}"),
    Point { x, y }    => println!("On neither axis: ({x}, {y})"),
}
```

Since all items (x and y) are Copy, Point is automatically Copy.

Destructuring Enums

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}

let msg = Message::ChangeColor(0, 160, 255);

match msg {
    Message::Quit => { println!("The Quit variant has no data to destructure."); }
    Message::Move { x, y } => {
        println!("Move in the x direction {x} and in the y direction {y}");
    }
    Message::Write(text) => { println!("Text message: {text}"); }
    Message::ChangeColor(r, g, b) => {
        println!("Change the color to red {r}, green {g}, and blue {b}")
    }
}
```

Destructuring nested structs and enums

```
enum Color {
    Rgb(i32, i32, i32),
    Hsv(i32, i32, i32),
}
```

```
enum Message {
    Quit,
    Move { x: i32, y: i32},
    Write(String),
    ChangeColor(Color),
}
```

```
let msg = ChangeColor(Hsv(0, 160, 255));

match msg {
    ChangeColor(Rgb(r, g, b)) => {
        println!("Change color to red {r}, green {g}, and blue {b}");
    }
    ChangeColor(Hsv(h, s, v)) => {
        println!("Change color to hue {h}, saturation {s}, value {v}");
    }
    _ => (),
}
```

Ignoring Values in a Pattern

There are a few ways to ignore entire values or parts of values:

- using the _ pattern (as a catch-all pattern),
- using the _ pattern within another pattern,
- using a name that starts with an underscore, or
- using .. to ignore remaining parts of a value.

Ignoring an Entire Value with _

```
fn foo(_: i32, y: i32) {
    println!("This code only uses the y parameter: {y}");
}

fn main() {
    foo(3, 4);
}
```

Ignoring Parts of a Value with Nested _

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {setting_value:?}");
```

Ignoring an Unused Variable by Starting Its Name with _

```
fn main() {
    let _x = 5; // No compiler warning!
    let y = 10;

    println!("y = {y}");
}
```

Ignoring Remaining Parts of a Value with ..

```
struct Point { x: i32, y: i32, z: i32 }
let origin = Point { x: 0, y: 0, z: 0 };
match origin {
    Point { x, .. } => println!("x is {x}"),
}
```

```
let numbers = (2, 4, 8, 16, 32);
match numbers {
    (first, .., last) => {
        println!("Some numbers: {first}, {last}");
    }
}
```

Extra Conditionals with Match Guards

```
let num = Some(4);
match num {
    Some(x) if x % 2 == 0 => println!("The number {x} is even"),
    Some(x) => println!("The number {x} is odd"),
    None => (),
}
```

```
let x = 4;
let y = false;
match x {
    4 | 5 | 6 if y => println!("yes"),
    _ => println!("no"),
}
```

Extra Conditionals with Match Guards (Cont’d)

```
let x = Some(5);
let y = 10;

match x {
    Some(50) => println!("Got 50"),
    Some(n) if n == y =>
        println!("Matched, n = {n}"),
    _ => println!("Default case, x = {x:?}"),
}

println!("at the end: x = {x:?}, y = {y}");
```

The @ Bindings

```
enum Message {
    Hello { id: i32 },
}

let msg = Message::Hello { id: 5 };

match msg {
    Message::Hello { id: id_variable @ 3..=7 } =>
        println!("Found an id in range: {id_variable}"),
    Message::Hello { id: 10..=12 } => {
        println!("Found an id in another range")
    }
    Message::Hello { id } => println!("Found some other id: {id}"),
}
```

Patterns everywhere

let Statement

let PATTERN = EXPRESSION;

```
let (x, _, z) = (1, 2, 3);
println!("x = {x}");
println!("z = {z}");
```

≡

```
match (1, 2, 3) {
    (x, _, z) => {
        println!("x = {x}");
        println!("z = {z}");
    }
}
```

let Statement (Cont’d)

let PATTERN = EXPRESSION;

```
let x = 5; // “bind everything to the variable x, whatever the value is.”

let (x, y) = (1, 2, 3); // error: expected tuple with 2 elements,
                        // found tuple with 3 elements
let (x, y, _) = (1, 2, 3);

// Destructuring Structs and Tuples
let ((feet, inches), Point { x, y }) = ((3, 10), Point { x: 3, y: -10 });
```

for-Loops

- In a for-loop, the value that directly follows the keyword `for` is a pattern.
- For example, in ``for x in y`` the `x` is the pattern.

```
let v = vec!['a', 'b', 'c'];

for (index, value) in v.iter().enumerate() {
    println!("{value} is at index {index}");
}
```

Function Parameters

- Function parameters can also be patterns.

```
fn main() {
    let point = (3, 5);
    print_coordinates(&point);
}

fn print_coordinates(&(x, y):  &(i32, i32)) {
    println!("Current location: ({x}, {y})");
}
```

Concise Control Flow with `if let`

- The `if let` syntax lets you combine `if` and `let` into a less verbose way to handle values that match one pattern while ignoring the rest.

```
let config_max = Some(3u8);

match config_max {
    Some(max) => println!("maximum: {max}"),
    _ => (),
}

// The syntax `if let` takes a pattern and an expression
// separated by an equal sign.
if let Some(max) = config_max {
    println!("maximum: {max}");
}
```

`if let` can have `else if` or `else`

```
let coin = Coin::Quarter(UsState::Alaska);
let mut count = 0;

if let Coin::Quarter(state) = coin {
    println!("State quarter from {state:?}!");
} else {
    count += 1;
}
```

`while let` Conditional Loop

```
let mut stack = Vec::new();

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop() {
    println!("{top}");
}
```

The ref Pattern

- The `ref` keyword makes a variable **bound to the reference** of a target.
- When doing pattern matching or destructuring via the `let` binding, the `ref` keyword can be used to take references to the fields of a struct/tuple.

```
#[derive(Debug)]
struct Person { name: String, age: Box<u8> }

let person = Person { name: String::from("Alice"), age: Box::new(20) };

// `name` is moved out of person, but `age` is referenced
let Person { name, ref age } = person;
```

The ref Pattern (Cont'd)

```
#[test]
fn both_are_the_same() {
    let s = String::from("Rusty!");
    let     y = &s;
    let ref x =  s;  // also works, but discouraged
                     // #[warn(clippy::toplevel_ref_arg)]
    assert_eq!(*x, *y);
}
```

Notes on `ref`

- `ref` on an entire `let` pattern is discouraged, take a reference with `&` instead.

```
let s = String::from("Rusty!");
let ref x =  s; // discouraged
let x = &s; // preferred

fn foo(ref _x: u8) {} // discouraged
fn foo(x: &u8) {} // preferred
```

- *Note that `&` takes part in the matching process, whereas `ref` does not.*

So, when to use `ref`?

“You want to pattern match against a variable,
but want to ***bind to the reference of the variable***
from then on.”

Partial Moves

- Within the destructuring of a single variable, both *by-move* and *by-reference* pattern bindings can be used at the same time.
- This **partial move** means that parts of the variable will be moved while other parts stay.
- In such a case, the parent variable cannot be used afterwards as a whole, however the parts that are only referenced (and not moved) can still be used.

Partial Moves (Cont’d)

```
#[derive(Debug)]
struct Person {
    name: String,
    age: Box<u8>,
}

let person = Person { name: String::from("Alice"), age: Box::new(20) };

// `name` is moved out of person, but `age` is referenced
let Person { name, ref age } = person;

println!("The person's age is {age} and name is {name}");

// Error! borrow of partially moved value: `person` partial move occurs
println!("The person struct is {person:?}");

// `person` cannot be used but `person.age` can be used as it is not moved
println!("The person's age from person struct is {}", person.age);
```

Summary

- Use **enum**s to create custom types that can be one of a set of enumerated values.
- Standard library's `Option<T>` type helps you use the type system to prevent errors.
- When **enum** values have data inside them, you can use `` `match` `` or `` `if let` `` to extract and use those values, depending on how many cases you need to handle.
- Your Rust programs can now express concepts in your domain using **struct**s and **enum**s.
- Creating custom types to use in your API ensures ***type safety***: the compiler will make certain your functions get only values of the type each function expects.