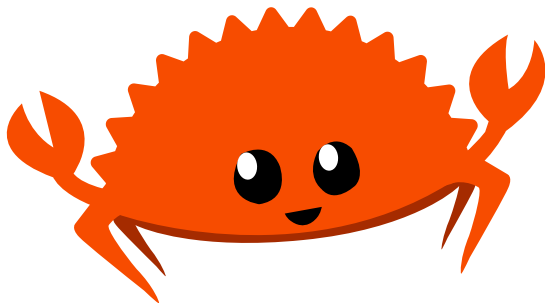




# Structs, Enums, and Patterns: Structuring and Matching Data in Rust

The Rust Programming Language

Summer Semester 2023



## Programming, scripting, and markup languages

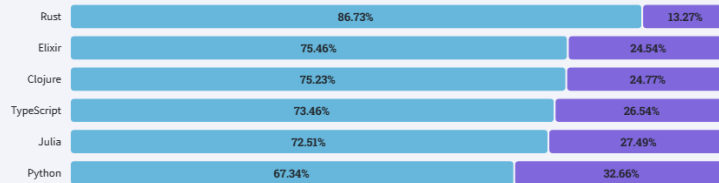
Rust is on its seventh year as the most loved language with 87% of developers saying they want to continue using it.

Rust also ties with Python as the most wanted technology with TypeScript running a close second.

Loved vs. Dreaded

Want

71,467 responses



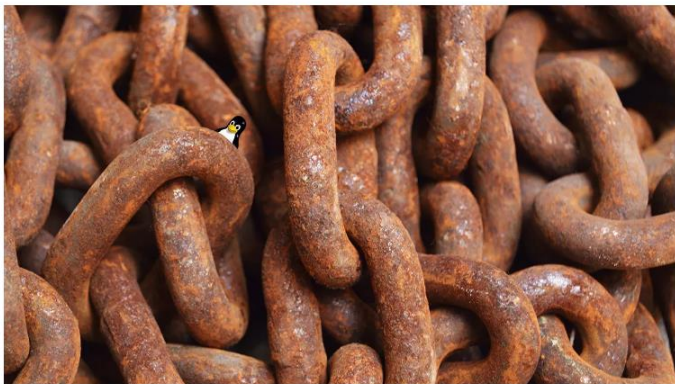
<https://survey.stackoverflow.co/2022/>

## Rust Programming Language To Land in Linux Kernel 6.1

By Ian Evenden published October 06, 2022

Linux will support the Rust programming language in its kernel from version 6.1.

 Comments (12)



(Image credit: Miguel Á. Padriñán)

<https://www.tomshardware.com/news/rust-in-linux-kernel/>

## A brief history of Rust at Facebook




FACEBOOK

<https://engineering.fb.com/2021/04/29/developer-tools/rust/>

## Microsoft is busy rewriting core Windows code in memory-safe Rust

Now that's a C change we can back

 Thomas Claburn

Thu 27 Apr 2023 / 20:45 UTC

Microsoft is rewriting core Windows libraries in the Rust programming language, and the more memory-safe code is already reaching developers.

David "dwizzle" Weston, director of OS security for Windows, announced the arrival of Rust in the operating system's kernel at BlueHat IL 2023 in Tel Aviv, Israel, last month.

"You will actually have Windows booting with Rust in the kernel in probably the next several weeks or months, which is really cool," he said. "The basic goal here was to convert some of these internal C++ data types into their Rust equivalents."

[https://www.theregister.com/2023/04/27/microsoft\\_windows\\_rust/](https://www.theregister.com/2023/04/27/microsoft_windows_rust/)



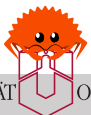
# Rust

- Focus on performance and memory safety
- For seventh time most loved language in the Stack Overflow survey
- Fastest growing programming language on GitHub
- Structs, enums and patterns are providing tools for organizing and managing data
- Working with data is essential for using the language effectively



# Structs

- Group values together and create a new type
  - Similar to the concept of a struct in C++ and to basic classes in languages like Java
- Rust differentiates between three types of structs
  - Named Structs
  - Tuple Structs
  - Unit Structs



# Named Structs

```
struct Person {  
    name: String,  
    age: u8,  
    is_student: bool,  
    height: f32,  
}
```

- `struct` used as keyword
- Followed by name of the struct (PascalCase)
- Each field is defined by a type and a name (snake\_case)



# Named Structs

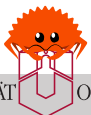
```
struct Person {  
    name: String,  
    age: u8,  
    is_student: bool,  
    height: f32,  
}
```

Struct declaration in Rust

```
struct Person {  
    std::string name;  
    uint8_t age;  
    bool isStudent;  
    float height;  
};
```

Struct declaration in C++

- Fundamentals and syntax very similar to what we know about structs from C++



```
let mut person = Person {  
    name: String::from("John"),  
    age: 42,  
    is_student: false,  
    height: 1.8,  
};
```

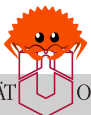
Instantiating a struct

```
let height = 1.8;  
let age = 42;  
let mut person = Person {  
    age,  
    name: String::from("John"),  
    is_student: false,  
    height,  
};
```

Instantiating using field init shorthand syntax

```
let mut person2 = Person {  
    age: 42,  
    is_student: true,  
    ..person  
};
```

Instantiating a struct using struct update syntax





```
person2.name = String::from("Jane");  
  
println!("{}", person2.name, person2.age);
```

- Field can be addressed using the name of the struct instance, followed by the `.` operator and the name of the field
- If the instance is mutable, the values can be changed, if not, only reading them is possible



```
let mut person = Person {  
    name: String::from("John"),  
    age: 42,  
    is_student: false,  
    height: 1.8,  
};
```

```
Person person = {"John", 42, false, 1.8};
```

```
Person person = {  
    .name = "John",  
    .age = 42,  
    .isStudent = false,  
    .height = 1.8  
};
```

```
person.name = "Jane";  
std::cout << person.name << std::endl;
```

- In C++ order of passed values has to be the same order as in struct definition



# Tuple Structs

```
struct Complex(f64, f64);
```

- Values are unnamed
  - Called elements
  - Meaning of struct is derived from its name and order of elements
- Each element has a possibly different type
- Each tuple struct defines its own new type



```
struct Complex(f64, f64);  
struct Vector(f64, f64);  
  
fn main() {  
    let z = Complex(1.0, 2.0);  
    let v = Vector(1.0, 2.0);  
  
    println!("Complex: ({} , {})",  
            z.0, z.1);  
    println!("Vector: ({} , {})",  
            v.0, v.1);  
}
```

- Instantiation and accessing values similar to tuples
  - Instead of using the name of the field, like with a named struct, use the index of the element instead
- `Complex` and `Vector` have the same elements, but are different types



# Unit structs

- Has no elements
  - Takes no space in memory
- Can have methods/functions
  - Helpful when working with traits

```
struct UnitStruct;  
  
fn main() {  
    let unit_struct = UnitStruct;  
}
```



# Methods

```
struct Person {  
    name: String,  
    age: u32,  
}  
  
impl Person {  
    fn say_hello(&self) {  
        println!("{}", self.name, self.age);  
    }  
}
```

- Each method has to take `self` as first argument
- Inside an `impl` block, `Self` is shorthand for the name of the current struct
- Following abbreviations are possible:
  - `self -> self: Self`
  - `&self -> self: &Self`
  - `&mut self -> self: &mut Self`
- Current instance of can be referenced by using `self` inside the method
- No implicit access for member variables possible



# Methods

```
struct Person {  
    name: String,  
    age: u32,  
}  
  
impl Person {  
    fn say_hello(&self) {  
        println!("{}", self.name, self.age);  
    }  
}
```

```
fn main() {  
    let person = Person {  
        name: String::from("Alice"),  
        age: 25,  
    };  
    person.say_hello();  
}
```

Alice is 25 years old.

- Methods can be called on instances of the struct using the `.` operator



# Methods on structs in C++

```
struct Person {  
    std::string name;  
    unsigned int age;  
  
    void sayHello() {  
        std::cout << name << " is "  
                    << age << " years old."  
                    << std::endl;  
    }  
};
```

```
int main() {  
    Person person = {"Alice", 25};  
    person.sayHello();  
    return 0;  
}
```





# Associated Functions

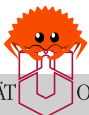
```
struct Circle;

impl Circle {
    fn area(r: f64) -> f64 {
        std::f64::consts::PI * r * r
    }
}

fn main() {
    let radius = 2.0;
    let area = Circle::area(radius);
    println!("The area is {}", area);
}
```

```
The area is 12.566370614359172.
```

- Functions inside an `impl` block are called type-associated functions or associated function
- Do not take `self` as first argument
  - Are not associated with an instance of a struct but instead with the type itself
- Can be used like static functions in other languages
- Are called using the name of the struct, followed by `::` and the name of the function
- Methods can be called like associated functions with instance as first argument



# Associated Functions

```
struct Circle;

impl Circle {
    fn area(r: f64) -> f64 {
        std::f64::consts::PI * r * r
    }
}

fn main() {
    let radius = 2.0;
    let area = Circle::area(radius);
    println!("The area is {}", area);
}
```

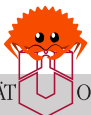
```
struct Circle {
    static double area(double r) {
        return M_PI * r * r;
    }
};

int main() {
    double radius = 2.0;
    double area = Circle::area(radius);
    std::cout << "The area is "
                << area << "."
                << std::endl;
    return 0;
}
```



# Comparison with C++

- Core concept of grouping data together is the same in Rust and C++
- Similar syntax
  - `impl` block
  - `self` for methods as first argument
- C++ classes/structs support inheritance
  - Rust does not, supports traits instead
- Concept of constructor and destructor does not exist in Rust
  - Helper functions (`new`)
  - Drop trait



# Enums

- Custom data type that allows for a set of values but only ever holds exactly one variant
- Can be used to model different states or options

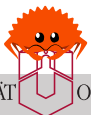


# Enums

```
enum Direction {  
    Up,           // 0  
    Down,         // 1  
    Left = 10,    // 10  
    Right,        // 11  
}
```

- enum as keyword
- Followed by name and curly braces
- Inside the curly braces are the possible variants
- By default, first variant 0, second variant 1, etc.
- Assigning custom integer is possible

```
let up = Direction::Up;  
let down = Direction::Down;
```



# Enums

```
enum Direction {  
    Up,           // 0  
    Down,         // 1  
    Left = 10,    // 10  
    Right,        // 11  
}
```

```
enum Direction {  
    Up,           // 0  
    Down,         // 1  
    Left = 10,    // 10  
    Right,        // 11  
};
```

- Basic enums are very similar in both languages
- In both languages an enum can be casted to an integer
- Rust does not allow casting an integer to an enum to ensure that the current variant is always valid
- C++ does allow casting integers to enums



# Enums with data

```
enum Status {  
    Error(String),  
    Pending(i32, String),  
    Warning { message: String,  
             code: i32 },  
    Valid,  
}
```

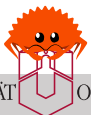
- Enums can optionally hold additional data
- Allows for three different kinds of variants:
  - Tuple variant
  - Struct variant
  - Unit variant
- These variants work as we would expect from the different struct variants



# Functionality on Enums

```
enum Direction {  
    Up,           // 0  
    Down,         // 1  
    Left = 10,    // 10  
    Right,        // 11  
}  
  
impl Direction {  
    fn is_up(self) -> bool {  
        self as i32 ==  
            Direction::Up as i32  
    }  
}
```

- Very similar to structs
- `impl` block with methods inside
  - `self` as first argument
- Enum does not have fields but instead exactly one current variant
  - Can be accessed using `self`





# Functionality on Enums

```
enum Direction {  
    Up,           // 0  
    Down,         // 1  
    Left = 10,    // 10  
    Right,        // 11  
}  
  
impl Direction {  
    fn is_up(self) -> bool {  
        self as i32 ==  
            Direction::Up as i32  
    }  
}
```

```
fn main() {  
    let up = Direction::Up;  
    let down = Direction::Down;  
  
    println!("Is up? {}", up.is_up());  
    println!("Is down? {}", down.is_up());  
}
```

Is up? true

Is down? false



# Result and Option

- Result

- Generic enum
- `Ok` and `Err` as variants
- Used to signal if an operation was successful or not
  - Successful result in `Ok` variant
  - Resulting Error in `Err` variant

```
enum Result<T, E> {  
    Ok(T),  
    Err(E),  
}
```

- Option

- Generic enum
- Rust does not have null as value
- `None` variant can represent absence of value
- `Some` variant holds value if present
- Forces to check if value is present
  - No null pointer dereferences
  - Only variables of type `Option` have to be checked

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

sources: <https://doc.rust-lang.org/std/result/enum.Result.html>  
<https://doc.rust-lang.org/std/option/enum.Option.html>



# Comparison with C++

- Fundamental concept is mostly the same
- C++ can also hold other types than just integers
  - Limited to integral types
  - Rust enums can hold any kind of data
- Rust enums can have functionality

```
enum Colors : char {  
    RED = 'r',  
    GREEN = 'g',  
    BLUE = 'b'  
};
```



# Match expression

```
let opt = Option::Some(5);

match opt {
  Option::Some(x) => println!("Some({})", x),
  Option::None => println!("None"),
}
```

Some(5)

- Match takes an expression
- Matches expression against several patterns
- If a pattern matches, match arm is executed
  - Only match arm of the first pattern that matches will be executed
  - Body is expression; value of expression is return value of whole match expression
- Has to be exhaustive

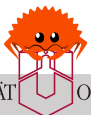


# Match expression

```
let opt = Option::Some(5);

match opt {
  Option::Some(x) => println!("Some({})", x),
  Option::None => println!("None"),
}
```

- Variables
  - Binds the value of the expression to the variable
    - Original value might get moved
  - Bound variable can be used in match arm and shadows variable with same name

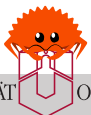


# Literals

- Literals

- Simplest form of patterns
- Fixed value
  - Value evaluated at compile time
- For example, integer, char, boolean, etc.
- Using only literals with match is similar to switch statement

```
let number = 2;  
match number {  
  1 => println!("One"),  
  2 => println!("Two"),  
  3 => println!("Three"),  
  _ => println!("Other"),  
}
```

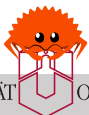


# Wildcards

- Wildcards
  - Underscore `_`
  - Matches any expression
  - Does not bind

```
let number = 2;  
match number {  
  1 => println!("One"),  
  2 => println!("Two"),  
  3 => println!("Three"),  
  _ => println!("Other"),  
}
```

Two



# Wildcards

- Wildcards
  - Underscore `_`
  - Matches any expression
  - Does not bind

```
let number = 5;  
match number {  
  1 => println!("One"),  
  2 => println!("Two"),  
  3 => println!("Three"),  
  _ => println!("Other"),  
}
```

Other





# Destructuring

- Allows breaking enums, structs, and tuples into pieces
- `let` takes a pattern
  - Allows to assign multiple values at once
  - Known as unpacking in other languages
- Function parameters take patterns as well
  - Allows destructuring the parameters

```
let triple = (0, -2, 3);  
let (x, y, z) = triple;  
// x: 0, y: -2, z: 3
```

```
struct Complex(f64, f64);  
  
fn printer(Complex(re, im): Complex) {  
    println!("re: {}, im: {}", re, im);  
}  
  
fn main() {  
    let c = Complex(3.3, 7.2);  
    printer(c);  
}
```



# Destructuring

- Rest Pattern
  - Two dots/points . .
  - Matches against remaining elements
  - Has to be unambiguous what is matched against
    - can only be used once in a pattern

```
let triple = (0, -2, 3);

match triple {
  (0, y, z) => println!("x: 0, y: {y}, z: {z}"),
  (1, ..) => println!("x is 1, rest doesn't matter"),
  (3, .., 5) => println!("x: 3, z: 5"),
  _ => println!("It doesn't matter what they are"),
}
```



# Destructuring

- Rest Pattern
  - Two dots/points . .
  - Matches against remaining elements
  - Has to be unambiguous what is matched against
    - can only be used once in a pattern

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

```
let p = Point { x: 0, y: 7, z: 0 };  
match p {  
    Point { x: 0, y, z } => println!("x: 0, y: {y}, z: {z}"),  
    Point { x: 1, y: a, z: b } => println!("x: {x}, y: {a}, z: {b}"),  
    Point { x, .. } => println!("x: {x}, rest doesn't matter"),  
}
```



# if let, while let, for loop

```
let option = Some(5);  
  
if let Some(value) = option {  
    println!("Value: {}", value);  
}
```

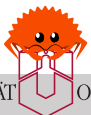
```
let mut vec = vec![1, 2, 3, 4, 5];  
while let Some(value) = vec.pop() {  
    println!("Value: {}", value);  
}
```

```
let a = [1, 2, 3, 4, 5];  
for (i, num) in a.iter().enumerate() {  
    println!("{}", i, num);  
}
```



# Refutable and Irrefutable Patterns

- Irrefutable patterns:
  - Always match, no matter the expression
- Refutable patterns:
  - Might not match the given expression
- Where to use refutable patterns
  - `if let`
  - `while let`
  - `match arms`
- Where to use irrefutable patterns
  - `let`
  - `for loops`
  - `Last match arm`



# Matching multiple patterns at once

- Concatenate patterns together using `|`
  - Each pattern will be checked, and arm gets executed if one of them matches
- Range pattern `..=` allows us to match against a range of values (13 and 19 both included)
  - Exclusive pattern `..` (19 not included) possible, but currently experimental
  - Inclusive pattern `...=` is possible but deprecated

```
let number = 7;
match number {
  1 => println! ("One"),
  2 | 3 | 5 | 7 | 11 => {
    println! ("Prime")
  },
  13..=19 => println! ("Teen"),
  x => println! ("{x} not special"),
}
```



# Refine Matches with Guards

- Additional `if` condition
- Variables bound by the pattern can be used by the `if` condition
- `if` condition only gets checked if pattern already matched
- match arm only gets executed if both pattern and `if` condition match
- Allows to check for something that cannot be expressed with patterns only

```
let opt = Some(5);
let number = 7;

match opt {
  Some(x) if x == number => {
    println!("opt is Some(7)");
  },
  Some(x) if x == 5 => {
    println!("opt is Some(5)");
  },
  _ => println!("Default case"),
}
```



# Combine everything

```
#[derive(Debug)]
struct Everything {
    first: i32,
    second: (i32, i32, i32, i32),
    third: OtherStruct,
    fourth: OtherEnum,
    fifth: Option<i32>,
}

#[derive(Debug)]
struct OtherStruct {
    first: i32,
    second: String,
}

#[derive(Debug)]
enum OtherEnum {
    First(i32),
    Second(OtherStruct),
}
```

```
let condition = true;

let everything = Everything {
    first: 1,
    second: (2, 3, 4, 5),
    third: OtherStruct {
        first: 6,
        second: String::from("Hello"),
    },
    fourth: OtherEnum::Second(OtherStruct {
        first: 7,
        second: String::from("World"),
    }),
    fifth: Some(8),
};
```





```

match everything {
  Everything {
    first: 1 | 2,
    second: (2..=4, ..),
    ..
  } => println!("First is 1 or 2"),

  Everything { .. } if condition => println!("Condition is true"),

  Everything { first, second, .. } => println!("First is {}, second is {:?}", first, second),

  Everything {
    first: _,
    second: _,
    third: _,
    fourth: OtherEnum::Second(OtherStruct { second, .. }),
    ..
  } => println!("String is {}", second),

  Everything {
    first: _,
    second: _,
    third: OtherStruct { second, .. },
    ..
  } => println!("String is {}", second),

  _ => println!("Default"),
}

```



# Comparison with C++

- C++ `switch` statement similar to `match` expression in Rust
  - `match` expression is more versatile
  - Does not support a system similar to Rust's patterns
- Newer C++ features allow for some things possible with patterns
  - structured bindings
- C++ has no counterpart to Rust's patterns

```
int main() {  
    std::tuple<int, std::string> tup{42, "hello"};  
  
    auto [i, str] = tup;  
  
    std::cout << "i: " << i << std::endl;  
    std::cout << "str: " << str << std::endl;  
  
    return 0;  
}
```



# Memory Layout

- Representation
  - How data is laid out in memory
  - default representation/rust representation
  - C representation
  - primitive representation
  - transparent representation



# Structs

- Support default representation and C representation
- default representation
  - Each field is properly aligned
  - Fields do not overlap
  - Alignment of the struct itself is at least the same as the maximum alignment of its fields
  - Does not guarantee order of fields in memory
- Alignment
  - Address of field in memory is always multiple of alignment
- Size
  - Offset of bytes between successive elements of same type
  - Always a multiple of alignment
- C representation
  - follows the same rules as structs in C
  - Order of fields in memory is same as definition order



# Structs

- Alignment modifiers
  - `#[repr()]`
  - `align` – define alignment of struct
  - `packed` – pack data together, no padding



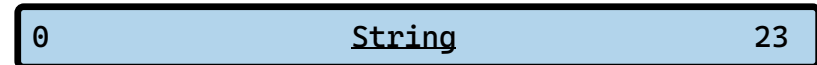
# Structs

```
struct StatusError(String);

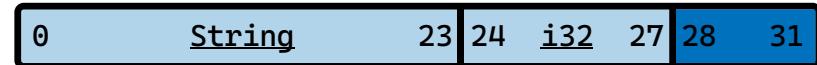
struct StatusPending(i32, String);

struct StatusWarning {
    message: String,
    code: i32,
    severity: String,
}
```

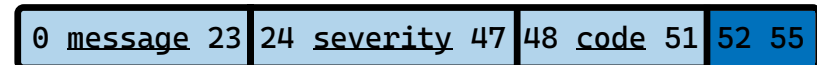
## StatusError



## StatusPending

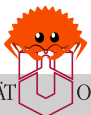


## StatusWarning



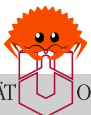
# Enums

- Supports default representation, the C representation and the primitive representation
- default representation and the C representation very similar to structs
  - Can be combined with primitive representation
- Enums have a tag field
  - Identifies current variant
  - At least size of smallest integer type than can represent all variants
- Change representation of tag with primitive representation through `#[repr(u16)]`, `#[repr(u32)]`, ...
- Tag is followed by possibly padding and then the data the enum variant holds
  - No guarantee how the data is laid out



# Enums

- Size of enum is size of tag field + size of largest possible variant (+ padding)
- If current variant is smaller than largest variant, additional space will be unused





# Enums

```
enum Status {
    Error(String),
    Pending(i32, String),
    Warning {
        message: String,
        code: i32,
        severity: String,
    },
    Valid,
}
```

Status::Error

0	Tag	3	4	7	8	String	31	32	55
---	-----	---	---	---	---	--------	----	----	----

Status::Pending

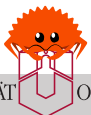
0	Tag	3	4	i32	7	8	String	31	32	55
---	-----	---	---	-----	---	---	--------	----	----	----

Status::Warning

0	Tag	3	4	code	7	8	message	31	32	severity	55
---	-----	---	---	------	---	---	---------	----	----	----------	----

Status::Valid

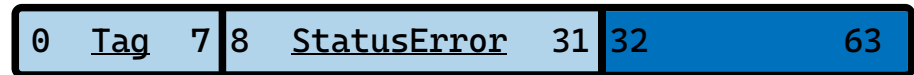
0	Tag	3	4								55
---	-----	---	---	--	--	--	--	--	--	--	----



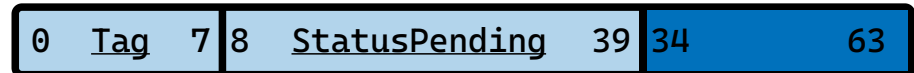
# Enums

```
enum StatusStructs {
    Error(StatusError),
    Pending(StatusPending),
    Warning(StatusWarning),
    Valid(StatusValid),
}
```

StatusStructs::Error



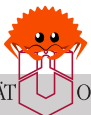
StatusStructs::Pending



StatusStructs::Warning



StatusStructs::Valid



# Summary

- Structs
  - Group data together, create a new type and add functionality to it
  - Named Struct: Has named fields
  - Tuple Struct: Has unnamed fields
  - Unit Struct: Does not have any data
- Enums
  - Encode options
  - Can hold data associated with variant
  - Can have functionality



# Summary

- Patterns
  - Allows extracting data from enums and other data types
  - Flexible way of checking for multiple scenarios
- Memory Layout
  - Loose default rules
  - Compiler tries to optimize for efficient layout
  - If needed, manually changing adjustments is possible



# Conclusion

- Core concept and even syntax are similar between Rust and C++
  - Rust extends on these core concepts
- Rust has some intentional design differences
- C++ does not have a feature that is similar to Rust patterns
  - Newer features, like structured bindings, allow unpacking, but there is no direct equivalent
- Rust has similar/same functionality and extends on core concepts





source: <https://www.rustacean.net/more-crabby-things/dancing-ferris.gif>



# References

- Jim Blandy, Jason Orendorff, and Leonora Tindall. 2021. Programming Rust: Fast, Safe Systems Development (Paperback ed.). O'Reilly Media.
- Catalin Cimpanu. 2017. Rust Programming Language Takes More Central Role in Firefox Development. Retrieved April 29, 2023 from <https://www.bleepingcomputer.com/news/software/rust-programming-language-takes-more-central-role-in-firefox-development/>
- Thomas Claburn. 2023. Google polishes Chromium code with a layer of Rust. Retrieved April 29, 2023 from [https://www.theregister.com/2023/01/12/google\\_chromium\\_rust/](https://www.theregister.com/2023/01/12/google_chromium_rust/)
- Thomas Claburn. 2023. Microsoft is busy rewriting core Windows code in memory-safe Rust. Retrieved April 29, 2023 from [https://www.theregister.com/2023/04/27/microsoft\\_windows\\_rust/](https://www.theregister.com/2023/04/27/microsoft_windows_rust/)
- The Rust Project Developers. 2018. The Rust Reference. Retrieved May 19, 2023 from <https://github.com/rust-lang/reference/blob/bbcacbb633a9963434ad8aaa5407e3c6ad95e4b9/src/expressions/match-expr.md>
- The Rust Project Developers. 2023. Rust API Guidelines - Naming. Retrieved May 22, 2023 from <https://rust-lang.github.io/api-guidelines/naming.html>
- The Rust Project Developers. 2023. Rust by Example. Retrieved April 29, 2023 from <https://doc.rust-lang.org/stable/rust-by-example/>
- The Rust Project Developers. 2023. Rustdoc. Retrieved April 29, 2023 from <https://doc.rust-lang.org/stable/rustdoc/>
- The Rust Project Developers. 2023. The Rustonomicon. Retrieved April 29, 2023 from <https://doc.rust-lang.org/stable/nomicon/>
- The Rust Project Developers. 2023. The Rust Reference. Retrieved April 29, 2023 from <https://doc.rust-lang.org/stable/reference/>
- The Rust Project Developers. 2023. Rust RFCs. Retrieved May 19, 2023 from <https://rust-lang.github.io/rfcs/>
- The Rust Project Developers. 2023. Option in std::option. Retrieved May 22, 2023 from <https://doc.rust-lang.org/std/option/enum.Option.html>



# References

- The Rust Project Developers. 2023. Result in std::result. Retrieved May 22, 2023 from <https://doc.rust-lang.org/std/result/enum.Result.html>
- The Rust Project Developers. 2023. Rust's Unsafe Code Guidelines Reference. Retrieved May 22, 2023 from <https://rust-lang.github.io/unsafe-code-guidelines/>
- The Rust Project Developers. 2023. Unsafe Code Guidelines Reference. Retrieved May 19, 2023 from <https://rust-lang.github.io/unsafe-code-guidelines/layout/enums.html>
- Ian Evenden. 2022. Rust Programming Language To Land in Linux Kernel 6.1. Retrieved April 29, 2023 from <https://www.tomshardware.com/news/rust-in-linux-kernel/>
- Facebook. 2021. A brief history of Rust at Facebook. Retrieved April 29, 2023 from <https://engineering.fb.com/2021/04/29/developer-tools/rust/>
- GitHub. 2022. The top programming languages | The State of the Octoverse. Retrieved May 22, 2023 from <https://octoverse.github.com/2022/top-programming-languages>
- Steve Klabnik and Carol Nichols. 2023. The Rust Programming Language. Retrieved April 29, 2023 from <https://doc.rust-lang.org/stable/book/>
- Stanley B. Lippman, Josée Lajoie, and Barbara E. Moo. 2012. C++ Primer (5th Edition) (Paperback ed.). Addison-Wesley Professional.
- Stack Overflow. 2022. Stack Overflow Developer Survey 2022. Retrieved April 29, 2023 from <https://survey.stackoverflow.co/2022/>
- Bjarne Stroustrup. 2014. Programming: Principles and Practice Using C++ (2nd Edition) (Paperback ed.). Addison-Wesley Professional.
- Karen Rustad Tölva. 2023. Rustacean.net: Home of Ferris the Crab. Retrieved May 22, 2023 from <https://www.rustacean.net/>





Hey there curious reader!



There are some topics I had already slides prepared for that had to be cut because of the time constraint



They are kept here for your interested mind



# Visibility of Structs

- Visibility is managed on module basis

```
mod my_module {
    struct Person {
        name: String,
        age: u32,
    }

    impl Person {
        fn say_hello(&self) {
            println!("{}", self.name, self.age);
        }
    }
}

fn main() {
    let person = my_module::Person {
        name: String::from("Alice"),
        age: 25,
    };

    person.say_hello();
}
```

**error[E0603]: struct `Person` is private**

--> structvisibility1.rs:16:29

```
16 | let person = my_module::Person {
    |                      ^^^^^^^ private struct
```



# Visibility of Structs

```
mod my_module {
    pub struct Person {
        name: String,
        age: u32,
    }

    impl Person {
        fn say_hello(&self) {
            println!("{}", "is {} years old.",
                self.name, self.age);
        }
    }
}

fn main() {
    let person = my_module::Person {
        name: String::from("Alice"),
        age: 25,
    };

    person.say_hello();
}
```

- Visibility is managed on module basis
  - pub keyword to make items accessible from outside the module

```
error[E0624]: method `say_hello` is private
--> structvisibility2.rs:21:12
8 |         fn say_hello(&self) {
  |         ----- private method defined here
...
21 |     person.say_hello();
   |           ^^^^^^^^^ private method
```



# Visibility of Structs

```
mod my_module {
    pub struct Person {
        name: String,
        age: u32,
    }

    impl Person {
        pub fn say_hello(&self) {
            println!("{}", "is {} years old.",
                self.name, self.age);
        }
    }
}

fn main() {
    let person = my_module::Person {
        name: String::from("Alice"),
        age: 25,
    };

    person.say_hello();
}
```

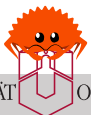
- Visibility is managed on module basis
  - pub keyword to make items accessible from outside the module
- Declaring methods as public makes it possible to access the methods from outside if struct is also public

```
error[E0451]: field `name` of struct `Person` is private
--> structvisibility3.rs:17:9
```

```
17 |         name: String::from("Alice"),
    |         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^ private field
```

```
error[E0451]: field `age` of struct `Person` is private
--> structvisibility3.rs:18:9
```

```
18 |         age: 25,
    |         ^^^^^ private field
```



# Visibility of Structs

```
mod my_module {  
    pub struct Person {  
        name: String,  
        age: u32,  
    }  
  
    impl Person {  
        pub fn say_hello(&self) {  
            println!("{}", is {} years old.",  
                self.name, self.age);  
        }  
    }  
}  
  
fn main() {  
    let person = my_module::Person {  
        name: String::from("Alice"),  
        age: 25,  
    };  
  
    person.say_hello();  
}
```

- By default, all fields of a struct are private
- If fields are private, they can not be initialized from outside the module
- Two solutions:
  - Make fields public
  - Construct the struct from within the module



# Visibility of Structs

```
mod my_module {  
    pub struct Person {  
        pub name: String,  
        pub age: u32,  
    }  
  
    impl Person {  
        pub fn say_hello(&self) {  
            println!("{}", self.name, self.age);  
        }  
    }  
}  
  
fn main() {  
    let person = my_module::Person {  
        name: String::from("Alice"),  
        age: 25,  
    };  
  
    person.say_hello();  
}
```

- All fields of the struct have to be public
- Otherwise not all fields could be initialized when constructing an instance of the struct, but Rust requires all fields to be initialized

Alice is 25 years old.

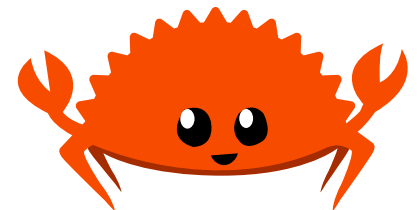


image: <https://www.rustacean.net/assets/rustacean-flat-happy.svg>



# Visibility of Structs

```
mod my_module {  
    pub struct Person {  
        name: String,  
        age: u32,  
    }  
  
    impl Person {  
        pub fn new(name: String, age: u32) -> Person {  
            Person {  
                name,  
                age,  
            }  
        }  
  
        pub fn say_hello(&self) {  
            println!("{}", self.name, self.age);  
        }  
    }  
}
```

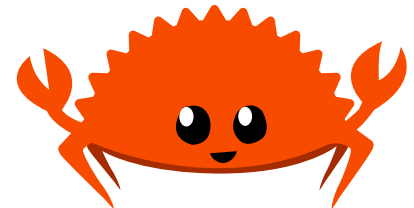


# Visibility of Structs

```
mod my_module {  
    pub struct Person {  
        name: String,  
        age: u32,  
    }  
  
    impl Person {  
        pub fn new(name: String, age: u32) -> Person {  
            Person {  
                name,  
                age,  
            }  
        }  
  
        pub fn say_hello(&self) {  
            println!("{}", self.name, " is {} years old.",  
                self.age);  
        }  
    }  
}
```

```
fn main() {  
    let person = my_module::Person::new(  
        String::from("Alice"),  
        25,  
    );  
  
    person.say_hello();  
}
```

Alice is 25 years old.





# Visibility of Structs

```
mod my_module {  
    pub struct Person {  
        name: String,  
        age: u32,  
    }  
  
    impl Person {  
        pub fn new(name: String, age: u32) -> Person {  
            Person {  
                name,  
                age,  
            }  
        }  
  
        pub fn say_hello(&self) {  
            println!("{}", self.name, self.age);  
        }  
    }  
}
```

- new is a helper function that has can create an instance of the struct, because it is inside the module
- new is not a special name or a language feature
  - But often used (e.g. `String::new()`)
- Rust does not have constructors as a part of the language
  - But the helper function acts similar



# if let

- Can be used to match a certain case and ignore the others
  - takes a pattern, followed by an equals sign and an expression
- Just like match arms, if let can introduce new variables
- Can be chained together with another else if let, but also with a normal if or else
  - They do not have to be related

```
let option = Some(5);  
let other = false;  
  
if let Some(value) = option {  
    println!("Value: {}", value);  
} else if other {  
    println!("Other is true");  
} else {  
    println!("Option is None");  
}
```



# while let

- Similar to if let
- Body will get executed as long as the expression matches the pattern

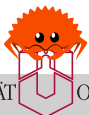
```
let mut vec = vec![1, 2, 3, 4, 5];  
while let Some(value) = vec.pop() {  
    println!("Value: {}", value);  
}
```



# for Loops

- Also takes a pattern
- Allows destructuring

```
let a = [1, 2, 3, 4, 5];  
for (i, num) in a.iter().enumerate() {  
    println!("{}", i, num);  
}
```



# @ Bindings

```
struct Person {  
    name: String,  
    age: u8,  
}  
  
impl Person {  
    fn new(name: String, age: u8) -> Self {  
        Person { name, age }  
    }  
  
    fn name(&self) -> &String {  
        &self.name  
    }  
  
    fn age(&self) -> u8 {  
        self.age  
    }  
}  
  
fn print_person(person: Person) {  
    println!("{}", person.name(), person.age());  
}
```

```
fn main() {  
    let person = Person::new(String::from("Alice"), 20);  
  
    match person {  
        Person { name: n, age: a } => {  
            print_person(Person::new(n, a));  
        }  
    }  
}
```



# @ Bindings

```
mod structs {
    pub struct Person {
        name: String,
        age: u8,
    }

    impl Person {
        pub fn new(name: String, age: u8) -> Self {
            Person { name, age }
        }

        pub fn name(&self) -> &String {
            &self.name
        }

        pub fn age(&self) -> u8 {
            self.age
        }
    }
}

fn print_person(person: structs::Person) {
    println!("{}", person.name(), person.age());
}
```

```
fn main() {
    let person = structs::Person::new(String::from("Alice"), 20);

    match person {
        structs::Person { name: n, age: a } => {
            print_person(structs::Person::new(n, a));
        }
    }
}
```

- Not possible to match against private fields

error[E0451]: field `name` of struct `Person` is private

```
30 |         structs::Person { name: n, age: a } => {
    |                               ^^^^^^^ private field
```

error[E0451]: field `age` of struct `Person` is private

```
30 |         structs::Person { name: n, age: a } => {
    |                               ^^^^^^^ private field
```

error: aborting due to 2 previous errors

For more information about this error, try `rustc --explain E0451`.



# @ Bindings

```
mod structs {  
    pub struct Person {  
        name: String,  
        age: u8,  
    }  
  
    impl Person {  
        pub fn new(name: String, age: u8) -> Self {  
            Person { name, age }  
        }  
  
        pub fn name(&self) -> &String {  
            &self.name  
        }  
  
        pub fn age(&self) -> u8 {  
            self.age  
        }  
    }  
}  
  
fn print_person(person: structs::Person) {  
    println!("{}", person.name(), person.age());  
}
```

```
fn main() {  
    let person = structs::Person::new(String::from("Alice"), 20);  
  
    match person {  
        p @ structs::Person { .. } => {  
            print_person(p);  
        }  
    }  
}
```

- Allows to bind the value of a matched (sub-) pattern to a variable

