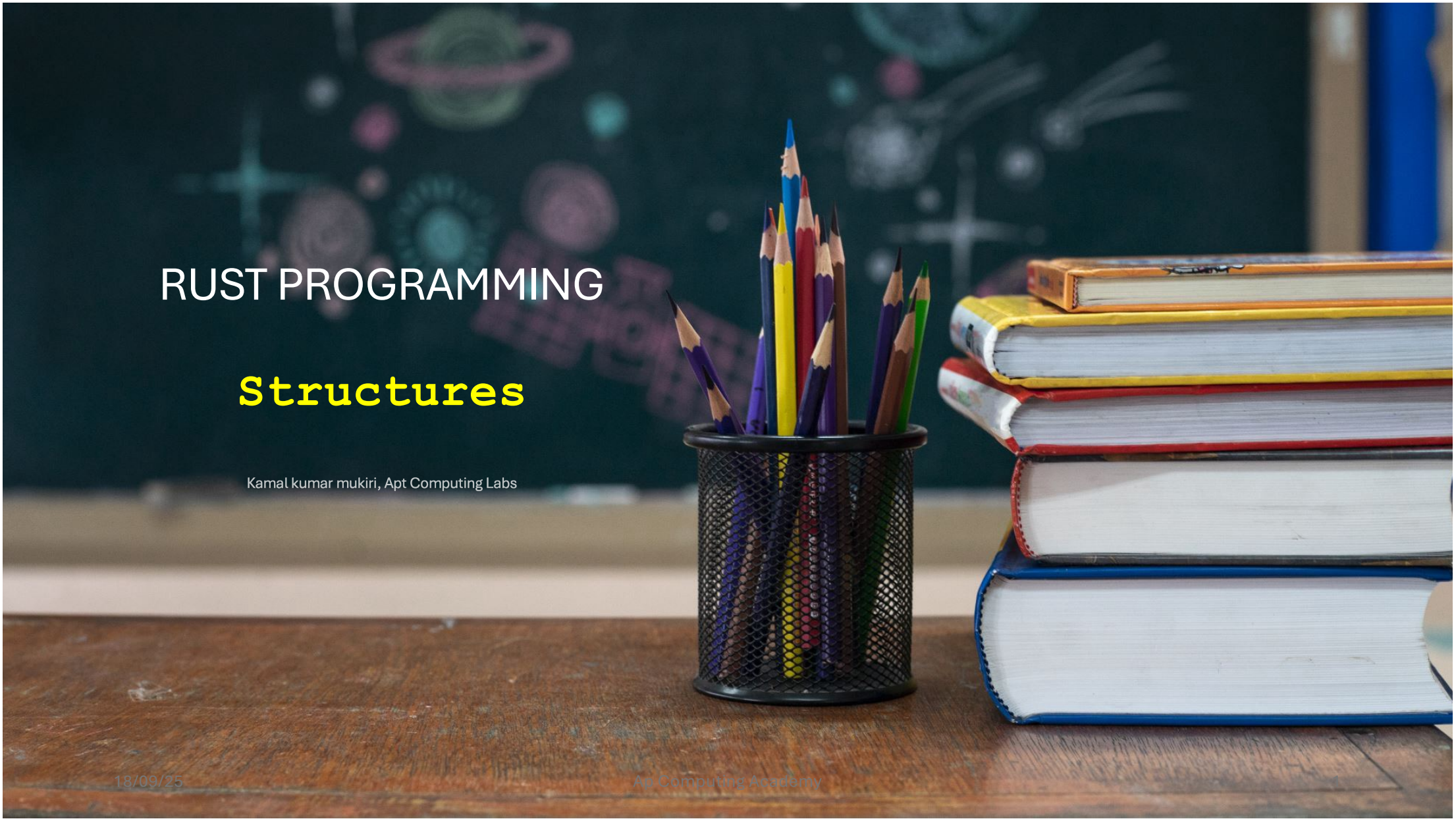
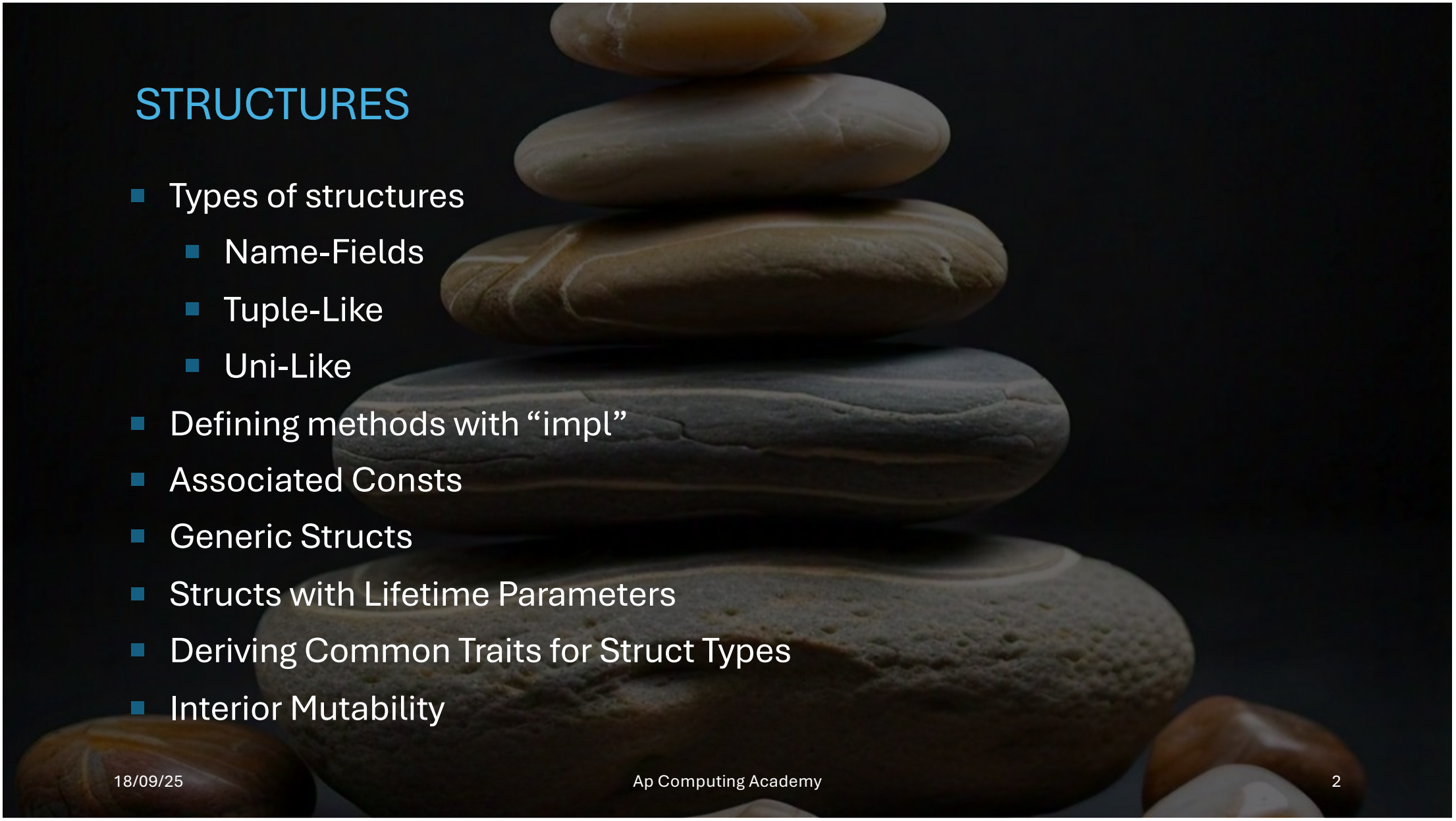

RUST PROGRAMMING
Structures
Kamal kumar mukiri, Apt Computing Labs



STRUCTURES

- Types of structures
  - Name-Fields
  - Tuple-Like
  - Uni-Like
- Defining methods with “impl”
- Associated Consts
- Generic Structs
- Structs with Lifetime Parameters
- Deriving Common Traits for Struct Types
- Interior Mutability

Named-Field Structures: Concept Overview

1. Structures in C/C++ are simple containers for data, with members public by default in C but configurable in C++.
2. Rust structures are more powerful, supporting features like ownership and borrowing.
3. Rust's structs don't include methods or access control inherently but gain them through impl blocks

```rust
struct MyStruct {
    x: i32,          // Stack
    y: String,       // Heap (the String data, not the pointer/length/capacity)
    z: Box<i64>,     // Heap (the i64 inside the Box)
}

fn main() {
    let a: i32 = 10;                // Stack
    let b = String::from("hi"); // Heap
    let c = Box::new(20);      // Heap

    let my_struct: MyStruct = MyStruct {
        x: 5,                      // Stack
        y: String::from("Rust"),   // Heap
        z: Box::new(100),          // Heap
    };
}
```

Named-Field Structures: Basics

```rust
#[derive(Debug)]
struct FriendNode {
    name: String,
    age: i32,
    friends: Vec<FriendNode>,
}

fn main() {
    let mut david = FriendNode {
                        age: 25,
                        name: "Kamal".to_string(),
                        friends: Vec::new()};
    println!("{:?}", david);

    david.name = "David Paul".to_string();
    println!("{:?}", david.name);
}
```

1
2
3

Output

```
./struct_rc
FriendNode { name: "Kamal", age: 25, friends: [] }
"David Paul"
```

1. Defining structure
   - Field name and type are separated by :
   - Each field is separated ,
   - No ; at the end of structure definition
2. Creating an instance and initializing
   - Mention field name along with value
   - All the fields should be initialized
   - Order does not matter
3. Modifying and accessing members
   - Use . to access or modify

Named-Field Structures: Copy

```
fn main() {
    let david = FriendNode {
                        age: 25,
                        name: "Kamal".to_string(),
                        friends: Vec::new()};

    let john = FriendNode {
                        name: "John".to_string(),
                        .. david}; // Copy partially from david to john
    println!("john details: {:?}", john);

    let peter = john; // Clone david to peter
    println!("peter details: {:?}", peter);
    // error[E0382]: borrow of partially moved value: `david`
    //println!("david details: {:?}", david);
    // error[E0382]: borrow of moved value: `john`
    //println!("john details: {:?}", john);
}
```

1

2

Output

```
% ./struct_basic1
john details: FriendNode { name: "John", age: 25, friends: [] }
peter details: FriendNode { name: "John", age: 25, friends: [] }
```

1. Copying data from one to another instance using .. EXPR
2. Partially copied instance (david) loses the ownership

Named-Field Structures:

```
fn main() {
    let david: FriendNode = FriendNode {
                            age: 25,
                            name: "David".to_string(),
                            friends: Vec::new()};

    let john: FriendNode = FriendNode {
                            friends: Vec::new(),
                            name: "John".to_string(),
                            .. david}; // Copy david to john
    println!("john details: {:?}", john);

    let peter: FriendNode = john;
    println!("peter details: {:?}", peter);
    // No ERROR :)
1   println!("david details: {:?}", david);
}
```

Output

```
% ./struct_basic1
john details: FriendNode { name: "John", age: 25, friends: [] }
peter details: FriendNode { name: "John", age: 25, friends: [] }
david details: FriendNode { name: "Kamal", age: 25, friends: [] }
```

CHALLENGE

1. No ERROR: Why?

Defining methods with “impl”:

```rust
#[derive(Debug)]
1 implementation
struct IdsEvent {
    id: u32,
    event: String,
}

impl IdsEvent {
    fn new(id: u32, event: &str) -> IdsEvent {
        if event.len() == 0 {
            return IdsEvent {
                id: id,
                event: "No Event".to_string(),
            };
        } else {
            return IdsEvent {
                id: id,
                event: event.to_string(),
            };
        }
    }

    fn get_id(&self) -> u32 {
        self.id
    }
}

fn main() {
    let mut ids_event1: IdsEvent = IdsEvent::new(id: 100, event: "");
    println!("ids_event1: {:?}", ids_event1);
    println!("ids_event1 id: {}", ids_event1.get_id());
}
```

1
2
3
4

Output

```
% ./struct_impl
ids_event1: IdsEvent { id: 100, event: "No Event" }
ids_event1 id: 100
```

1. Implementing member function for structures using impl
2. new function called as Associated Functions which does not need “self”
3. Writing a member function, called on instance
4. Invoking “new” associated function to construct instance

“impl” methods:

Methods in “impl” can be categorized two ways:

- Instance methods:
  - **Consumable Instance methods**: Which demands ”self” (depends on definition, no default behavior)
  - **Mutable Instance methods**: Which demands “&mut self”
  - **Instance methods**: Which demands “&self”
    - Ex: “get_id”
- Associated functions: Functions that are defined within an impl block for a struct but don't take self as a parameter.
  - Ex: “new”

Example with all types of functions in “impl”:

```rust
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    // Immutable borrowing: the method borrows `self` immutably.
    pub fn area(&self) -> u32 {
        self.width * self.height
    }

    // Mutable borrowing: the method borrows `self` mutably, allowing modification.
    pub fn scale(&mut self, factor: u32) {
        self.width *= factor;
        self.height *= factor;
    }

    // Taking ownership: the method takes ownership of `self`.
    pub fn destroy(self) {
        println!("Rectangle with dimensions {}x{} is destroyed.", self.width, self.height);
        // The instance is consumed here and can't be used afterward.
    }

    // Associated function: does not take `self`.
    pub fn new(width: u32, height: u32) -> Self {
        Self { width, height }
    }
}
```

```rust
fn main() {
    // Using the associated function `new` to create an instance of `Rectangle`.
    let mut rect: Rectangle = Rectangle::new(width: 30, height: 50);

    // Immutable borrowing: calling the `area` method, which doesn't modify `rect`.
    println!("The area of the rectangle is {} square pixels.", rect.area());

    // Mutable borrowing: calling the `scale` method, which modifies `rect`.
    rect.scale(factor: 2);
    println!("After scaling, the area is {} square pixels.", rect.area());

    // Taking ownership: calling the `destroy` method, which consumes `rect`.
    rect.destroy();
    // At this point, `rect` can no longer be used because it has been consumed.
}
```

Output

```
 % ./all_types_of_methods
The area of the rectangle is 1500 square pixels.
After scaling, the area is 6000 square pixels.
Rectangle with dimensions 60x100 is destroyed.
```

General associative methods used:

- **new**
  - Creates and returns a new instance of the struct.
  - Example: MyStruct::new().
- **default**
  - Returns a default instance of the struct. Often used when implementing the Default trait.
  - Example: MyStruct::default().
- **from / from_***
  - Converts from another type to the struct. Often used for custom conversions.
  - Example: MyStruct::from("string").
- **with_***
  - Alternative constructors that create an instance with specific fields initialized.
  - Example: MyStruct::with_default_values().
- **create_***
  - Factory methods that generate instances, sometimes in complex or multiple-step scenarios.
  - Example: MyStruct::create_pair(1, 2).

- **max_value / min_value**
  - Returns the maximum or minimum value that a struct or its fields can hold.
  - Example: MyStruct::max_value().
- **description**
  - Returns a static description or string related to the struct.
  - Example: MyStruct::description().
- **from_str**
  - Converts a string to an instance of the struct, commonly used for parsing.
  - Example: MyStruct::from_str("input string").
- **from_tuple**
  - Converts a tuple to an instance of the struct.
  - Example: MyStruct::from_tuple((1, 2)).
- **from_*_slice**
  - Converts a slice of a specific type to an instance of the struct.
  - Example: MyStruct::from_u8_slice(&[1, 2, 3]).

Cont...

- **constants**
  - Associated constants that represent fixed values associated with the struct.
  - Example: MyStruct::MAX_VALUE.
- **is_valid**
  - Returns a boolean indicating whether some condition is met for a type or value.
  - Example: MyStruct::is_valid(some_value).
- **parse**
  - Parses a value into the struct, often used for implementing the FromStr trait.
  - Example: MyStruct::parse("input").
- **get_instance**
  - Returns a singleton or globally accessible instance of the struct.
  - Example: MyStruct::get_instance().
- **to_string**
  - Converts the struct to a string representation. Often used with the ToString trait.
  - Example: MyStruct::to_string().
- **to_bytes / from_bytes**
  - Converts the struct to or from a byte array, useful for serialization or binary formats.
  - Example: MyStruct::to_bytes().
- **zero**
  - Returns a zeroed-out instance of the struct, often used in numeric contexts.
  - Example: MyStruct::zero().
- **unit**
  - Returns a unit instance (for unit-like structs or types that represent an empty state).
  - Example: MyStruct::unit().
- **init**
  - Initializes some internal state or system related to the struct.
  - Example: MyStruct::init().
- **configure**
  - Configures some settings or parameters related to the struct.
  - Example: MyStruct::configure(settings).

Cont...

- **all**
  - Returns a collection of all possible instances, often used with enums or specific struct types.
  - Example: MyEnum::all().
- **empty**
  - Returns an empty instance of a struct, often used for collections or containers.
  - Example: MyStruct::empty().
- **from_iter**
  - Creates an instance of the struct from an iterator, often used for collections.
  - Example: MyStruct::from_iter(iterator).
- **len / is_empty**
  - Returns the length or checks if the struct is empty, typically used for collection-like structs.
  - Example: MyStruct::len() or MyStruct::is_empty().
- **default_instance**
  - Returns a predefined default instance that might be used frequently.
  - Example: MyStruct::default_instance().

Tuple-Like Structures: Basics and “impl”

```rust
#[derive(Debug)]
1 implementation
struct Color(u8, u8, u8);
impl Color {
    fn new(r: u8, g: u8, b: u8) -> Color {
        Color(r, g, b)
    }

    fn print_color (&self) {
        println!("Color values: {}, {}, {}", self.0, self.1, self.2);
    }
}

fn main() {
    let black: Color = Color(0, 0, 0);
    let mut white: Color = Color(255, 255, 255);

    println!("Black color values: {}", black.0);
    white.0 = 0;
    white.print_color();
}
```

1
2
3

Output

```
% ./tuple_strucut_basic
Black color values: 0
Color values: 0, 255, 255
```

1. Defining structure
   - No need of fields names, only types are enough.
2. Creating instance
   1. Order matters
3. Modifying and accessing members
   - Use .index to access or modify

Misc: Structures

```rust
use std::mem;

#[derive(Debug)]
0 implementations
struct IdsEvent {
    pub sensor_id: u16,
    pub id: u32,
    pub severity: u8,
    pub timestamp: u64,
    pub data: [u8; 10],
}

#[repr(C)]
#[derive(Debug)]
0 implementations
struct IdsEvent2 {
    pub sensor_id: u16,
    pub id: u32,
    pub severity: u8,
    pub timestamp: u64,
    pub data: [u8; 10],
}

fn main() {
    let event: IdsEvent = IdsEvent {
        sensor_id: 0,
        id: 1,
        severity: 2,
        timestamp: 3,
        data: [4; 10],
    };

    println!("Size of IdsEvent: {}", mem::size_of::<IdsEvent>());
    println!("Size of IdsEvent2: {}", mem::size_of::<IdsEvent2>());
}
```

1

Output

```
% ./strucut_padding
Size of IdsEvent: 32
Size of IdsEvent2: 40
```

Rust does not promise the order of fields in memory.

1. The #[repr(C)] attribute in Rust is used to control the memory layout of data structures, making them compatible with the C programming language's memory layout.

Important when you're interfacing with C libraries.

Structures: Default traits

```rust
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
0 implementations
struct Point {
    x: i32,
    y: i32,
}

fn main() {
    let point: Point = Point { x: 10, y: 20 };
    let mut point2 = point.clone();
    let mut point3: Point = point;

    point2.x = 11;
    point3.x = 12;
    println!("{:?}", point);
    println!("{:?}", point2);
    println!("{:?}", point3);

    println!("Eq = {}", point == point2);
}
```

1

- In Rust, traits define shared behavior that types can implement.
- Deriving common traits is often needed when you want to use standard behaviours like equality checks, debugging, or cloning without having to manually implement them.

1. Rust has several built-in traits that can be derived automatically using the #[derive] attribute.

Defining a simple trait:

```rust
trait Greet {
    fn greet(&self) -> String;
}

1 implementation
struct Person {
    name: String,
}

impl Greet for Person {
    fn greet(&self) -> String {
        format!("Hello, my name is {}!", self.name)
    }
}

fn main() {
    let person: Person = Person {
        name: String::from("Alice"),
    };
    println!("{}", person.greet());
}
```

1. Define own trait
2. Implementing for Person

Defining a simple trait: Trait functions

```
fn say_hello<T: Greet>(entity: &T) {
        println!("{}", entity.greet());
}
```

1. “say_hello” can be called for any structure which has implemented trait “Greet”

Thanks