

**Shared Functionality** | Generic Structures

# ■ Generic Structures

- ◆ Store data of any type within a structure
  - Trait bounds restrict the type of data the structure can utilize
    - ▶ Also known as “generic constraints”
- ◆ Useful when making your own data collections
- ◆ Reduces technical debt as program expands
  - New data types can utilize generic structures and be easily integrated into the program

# ■ Conceptual Example

- ◆ Generic structure for template rendering
  - Template Source Paths
  - Variable substitution data
  - Generic render target
    - ▶ File
    - ▶ Terminal
    - ▶ Image
    - ▶ Bytes

# ■ Syntax

```
struct Name<T: Trait1 + Trait2, U: Trait3> {  
    field1: T,  
    field2: U,  
}
```

```
struct Name<T, U>  
where  
    T: Trait1 + Trait2,  
    U: Trait3,  
{  
    field1: T,  
    field2: U,  
}
```

## ■ Example – Definition

```
trait Seat {  
    |     fn show(&self);  
}  
  
struct Ticket<T: Seat> {  
    |     location: T,  
}
```

# ■ Example – Types of seating

```
#[derive(Clone, Copy)]
enum ConcertSeat {
    FrontRow,
    MidSection(u32),
    Back(u32),
}

impl Seat for ConcertSeat {
    fn show(&self) { ...
}
}
```

```
#[derive(Clone, Copy)]
enum AirlineSeat {
    BusinessClass,
    Economy,
    FirstClass,
}

impl Seat for AirlineSeat {
    fn show(&self) { ...
}
}
```

## ■ Example – Usage with single type

```
trait Seat {  
    fn show(&self);  
}  
  
struct Ticket<T: Seat> {  
    location: T,  
}  
  
fn ticket_info(ticket: Ticket<AirlineSeat>) {  
    ticket.location.show();  
}  
  
let airline = Ticket { location: AirlineSeat::FirstClass };  
ticket_info(airline);
```

## ■ Example – Usage with generic type

```
trait Seat {  
    fn show(&self);  
}  
  
struct Ticket<T: Seat> {  
    location: T,  
}  
  
fn ticket_info<T: Seat>(ticket: Ticket<T>) {  
    ticket.location.show();  
}  
  
let airline = Ticket { location: AirlineSeat::FirstClass };  
let concert = Ticket { location: ConcertSeat::FrontRow };  
ticket_info(airline);  
ticket_info(concert);
```



# ■ Details

```
struct Ticket<T: Seat> {  
    location: T,  
}
```

```
fn ticket_info<T: Seat>(ticket: Ticket<T>) {  
    ticket.location.show();  
}
```

```
let airline = Ticket { location: AirlineSeat::FirstClass };  
let concert = Ticket { location: ConcertSeat::FrontRow };  
ticket_info(airline);  
ticket_info(concert);
```

## ■ Details – Behind the scenes

```
struct AirlineTicket {  
    location: AirlineSeat,  
}
```

```
struct ConcertTicket {  
    location: ConcertSeat,  
}
```

```
fn airline_ticket_info(ticket: AirlineTicket) {  
    ticket.location.show();  
}
```

```
fn concert_ticket_info(ticket: ConcertTicket) {  
    ticket.location.show();  
}
```

# ■ Details – Heterogeneous vector

```
let airline = Ticket { location: AirlineSeat::FirstClass };
let concert = Ticket { location: ConcertSeat::FrontRow };
ticket_info(airline);
ticket_info(concert);

let tickets = vec![airline, concert];
```

error[E0308]: mismatched types

--> src/main.rs:89:33

```
89 |         let tickets = vec![airline, concert];
    |                                ^^^^^^^^^ expected enum `AirlineSeat`,
    |                                found enum `ConcertSeat`
= note: expected type `Ticket<AirlineSeat>`
       found struct `Ticket<ConcertSeat>`
```

# ■ Recap

- ◆ Generic structures allow storage of arbitrary types
  - May be any type or constrained by traits
- ◆ Cannot mix generic structures in a single collection
  - Generic structures expand to structures of a specific type
- ◆ Two different syntaxes

# ■ Recap – Syntax

```
struct Name<T: Trait1 + Trait2, U: Trait3> {  
    field1: T,  
    field2: U,  
}
```

```
struct Name<T, U>  
where  
    T: Trait1 + Trait2,  
    U: Trait3,  
{  
    field1: T,  
    field2: U,  
}
```