Recap

What you know so far

Organize data with struct and enum

```
/// A pointer and a capacity
struct RawVec<T> {
   ptr: NonNull<T>,
   cap: usize,
}

/// A vector that can grow by pushing elements
in it.
pub struct Vec<T> {
   buf: RawVec<T>,
   len: usize,
}
```

```
enum Direction {
   North,
   East,
   South,
   West,
}

impl Direction {
   fn to_string(&self) -> String {
       match self {
           Direction::North => String::from("North"),
           Direction::East => String::from("East"),
           Direction::South => String::from("South"),
           Direction::West => String::from("West"),
       }
   }
}
```

Write trait and implement them

```
pub trait Summary {
    fn summarize(&self) -> String {
        String::from("(Read more...)")
    }
}

struct BlogArticle {
    text: String,
}

impl Summary for BlogArticle {
    fn summarize(&self) -> String {
        let chars = self.text.chars();
        let mut sub: String = chars.into_iter().take(10).collect();
        sub.push_str("...");
        sub
    }
}
```

Write generic functions

```
fn print_summary(x: &impl Summary) {
   println!("{}", x.summarize());
}

fn print_summary<T: Summary>(x: &T) {
   println!("{}", x.summarize());
}

fn print_summary<T>(x: &T)
where
   T: Summary,
{
   println!("{}", x.summarize());
}
```

```
fn print_summary(x: &dyn Summary) {
   println!("{}", x.summarize());
}
```

Understand ownership

```
// takes ownership of the string
// (the string is deallocated when it goes out of scope)
pub fn print_string(s: String) {
   println!("{s}");
}

// borrows the string
// (the string is """given back""" at the end of the function)
pub fn print_string(s: &String) {
   println!("{s}");
}
```

Most of Rust can be derived from that

- Modelling state: enum + struct
- Error handling: enum + struct + genericity
- Lifetime annotation: ownership + genericity
- Iterators: trait + genericity
- Destructors: trait
- …
- RAII: Destructors
- Builder: struct + genericity
- ...

Modelling state

How to prevent errors

Q: What could go wrong here ?

```
struct Person {
   has_cat: bool,
   cats: Vec<Cat>
}
```

Q: What could go wrong here ?

```
struct Person {
   has_cat: bool,
   cats: Vec<Cat>
}
```


```
struct Person {
   cats: Vec<Cat>,
}

impl Person {
   pub fn has_cats(&self) -> bool {
        !self.cats.is_empty()
   }
}
```

Another example of invalid states

```
struct Cat {
    is_sleeping: bool,
    is_eating: bool,
    is_playing: bool,
    is_hungry: bool,
}
```

Let’s fix some invalid states

```
struct Cat {
    is_sleeping: bool,
    is_eating: bool,
    is_playing: bool,
    is_hungry: bool,
}
```

```
enum CatActivity {
    Sleeping,
    Eating,
    Playing,
}

struct Cat {
    activity: CatActivity,
    is_hungry: bool,
}
```

Let's fix all invalid states

```
struct Cat {
    is_sleeping: bool,
    is_eating: bool,
    is_playing: bool,
    is_hungry: bool,
}
```

```
enum CatActivity {
    Sleeping,
    Eating,
    Playing,
}

struct Cat {
    activity: CatActivity,
    is_hungry: bool,
}
```

```
enum CatActivity {
    Sleeping,
    Eating,
    Playing(bool),
}

struct Cat {
    activity: CatActivity,
}

impl Cat {
    pub fn is_hungry(&self) -> bool {
        match self.activity {
            CatActivity::Playing(hungry) => hungry,
            _ => false,
        }
    }
}
```

Let’s make it more clear

```
enum CatActivity {
    Sleeping,
    Eating,
    Playing(bool),
}

struct Cat {
    activity: CatActivity,
}

impl Cat {
    pub fn is_hungry(&self) -> bool {
        match self.activity {
            CatActivity::Playing(is_hungry) => is_hungry,
            _ => false,
        }
    }
}
```

```
enum CatActivity {
    Sleeping,
    Eating,
    Playing { is_hungry: bool },
}

struct Cat {
    activity: CatActivity,
}

impl Cat {
    pub fn is_hungry(&self) -> bool {
        match self.activity {
            CatActivity::Playing{is_hungry} => is_hungry,
            _ => false,
        }
    }
}
```

Error handling

How to handle errors you could not prevent

How C++/Python/etc. does it

Functions have:

- a single entry point
- multiple return instructions

This makes the control flow of your function clear.

But you also have another, hidden control flow: **exceptions**.

Exceptions break your ability to reason about your code

```
std::mutex m;  // if you call `lock`, you must call `unlock`

void function_with_lock() {
   m.lock();
   do_stuff();
   m.unlock();
}
```

This may have a bug if do_stuff raises an exception.
Why not simply return an error in case of an error ?

Introducing: Option and Result

```
pub enum Option<T> {
    /// No value.
    None,
    /// Some value of type `T`.
    Some(T),
}
```

```
pub enum Result<T, E> {
    /// Contains the success value
    Ok(T),
    /// Contains the error value
    Err(E),
}
```

Let’s look at an Option

```
fn main() {
    let x: Option<u32> = Some(5);
    match x {
        None => println!("Nothing to see here"),
        Some(value) => println!("The option has the value {value}."),
    }
}
```

Remember, Options are generic

```
struct PhoneNumber {
  // data, e.g. indicator
}

fn main() {
   let call_me = PhoneNumber { /* data */ };
   // Option of my very own type
   let call_me_maybe: Option<PhoneNumber> = Some(call_me);
   // 'expect' stops the program if call_me_maybe is None
   let ring_ring = call_me_maybe.expect("why don't you give me a call?");
}
```

Option: practical use

```
use std::collections::HashSet;

fn get_and_do_something(set: &HashSet<u32>, key: u32) -> Option<u32> {
    let value_opt: Option<&u32> = set.get(&key);
    let value: &u32 = match value_opt {
        Some(actual_value) => actual_value,
        None => {
            return None;
        }
    };

    // do something here
    let value = *value +1;

    // return the new value
    Some(value)
}
```

Option: practical use - without boilerplate

```rust
use std::collections::HashSet;

fn get_and_do_something(set: &HashSet<u32>, key: u32) -> Option<u32> {
    let value: &u32 = set.get(&key)?;  // early return here

    // do something here
    let value = *value + 1;

    // return the new value
    Some(value)
}
```

Let’s look at a Result

```
fn from_vec_to_string_uppercase(vec: Vec<u8>) -> Result<String, FromUtf8Error> {
    let my_string_res: Result<String, FromUtf8Error> = String::from_utf8(vec);
    let string = match my_string_res {
        Ok(valid_string) => valid_string,
        Err(utf8_error) => {
            return Err(utf8_error);
        }
    };
    Ok(string.to_uppercase())
}
```

Let’s look at a Result - and remove the boilerplate

```
fn from_vec_to_string_uppercase(vec: Vec<u8>) -> Result<String, FromUtf8Error> {
   Ok(String::from_utf8(vec)?.to_uppercase())
}

// or
fn from_vec_to_string_uppercase(vec: Vec<u8>) -> Result<String, FromUtf8Error> {
   String::from_utf8(vec).map(|string| string.to_uppercase())
}
```

Lifetime annotation

“I fell for a local variable… but it was never meant to last.”

What is a lifetime ?

Types:

- describe what your data is

Generic over type:

- describe all possible types accepted by a function

Lifetime:

- describe when your data is

Lifetime annotation:

- describe a set of possible lifetime

Example of a lifetime annotation

```
struct Cat {
    // data
}

struct Person {
    cat: &Cat,  // because cats can be shared
}
```

Does not compile…

Example of a lifetime annotation

```
struct Cat {
    // data
}

struct Person<'a> {
    cat: &'a Cat, // because cats can be shared
}
```

Tells the compiler a Person is invalid if its Cat goes out of scope

Example of a lifetime annotation - with an impl block

```
struct Cat {
    // data
}

struct Person<'a> {
    cat: &'a Cat, // because cats can be shared
}

impl<'a> Person<'a> {
    fn new(cat: &'a Cat) -> Person<'a> {
        Person { cat }
    }
}
```

Example of a lifetime annotation - with an impl block

```
struct Cat {
    // data
}

struct Person<'a> {
    cat: &'a Cat, // because cats can be shared
}

impl Person<'_> {
    fn new(cat: &Cat) -> Person {
        Person { cat }
    }
}
```

Lifetime can be deduced by the compiler here

Example of a lifetime annotation - with an impl block

```
struct Cat {
    // data
}

struct Person<'a> {
    cat: &'a Cat, // because cats can be shared
}

impl Person<'_> {
    fn new(cat: &Cat) -> Person {
        Person { cat }
    }
}
```

```
pub fn main() {
    let cat = Cat {};
    let person = Person::new(&cat);
    drop(cat);
    drop(person); // compiler error
}
```

Destructors and RAII

How would you write the drop function ?

https://doc.rust-lang.org/nightly/src/core/mem/mod.rs.html#935

Writing your own destructor

A destructor is a method that is run when the object goes out of scope.
In Python, it's a "magic method".

In Rust, it's simply a trait!

```
impl Drop for Person<'_> {
    fn drop(&mut self) {
        println!("ciao");
    }
}
```