Object Oriented Programming in Rust

Ryan Eberhardt and Julio Ballista
April 22, 2021
Logistics

● Week 3 exercises due today at 11:59 PST.
  ○ Please let us know if you get stuck / feel confused! We want you to sleep!
● Using myth? See announcements channel
● Participation incentive: At the end of the quarter, I’ll randomly select at least 3 people that participated 10 times throughout the quarter, and I’ll make you a custom mug or pot (see @pottedpeasceramics)
  ○ Asking or answering a question in lecture (out loud, or in the chat) or on Slack all count as participation
● Today: How can we write good code in Rust?
Object Oriented Programming in C++
Classes

- "Object" Oriented: Create an 'object' - movie database, and you can perform methods on this object.
- You can create instances of objects, and each would have their own set of variables. (Movie database with different files)
- Classes divided into public and private regions.
- public members can be accessible to anyone with reference to an instance
- private members only accessible to the implementer of the class

```cpp
class imdb {
public:
    imdb(const std::string& directory)
    bool getCredits(...) 
private:
    /* Elements 
    const char* kActorFileName;
}
```
What are some advantages to Classes?
Advantages to Class Design

- **Code-Reuse**: Want an object to be different based on the file it takes in? Add one parameter to its constructor, and suddenly you have two different implementations, but just one class!

- **Code-Hiding**: Don't need to expose parts of a class not needed for a user to interact with it. Could lead to misuse, and add too much overhead to contribute to a project.
class TeddyBear {
    public:
        TeddyBear(..);
        void roar_sound();
}

class PurpleTeddyBear {
    public:
        TeddyBear(..);
        void roar_sound();
        void purple_button_song();
}

class RedTeddyBear {
    public:
        TeddyBear(..);
        void roar_sound();
        void red_button_song();
}

class PurpleTeddyBear {
    public:
        TeddyBear(..);
        void roar_sound();
        void green_button_song();
}
We still have to repeat a bunch of code!
class TeddyBear {
  public:
    TeddyBear(…);
    void roar_sound();
};

class RedTeddyBear {
  public:
    red_button_song();
};

class PurpleTeddyBear {
  public:
    purple_button_song();
};

class GreenTeddyBear {
  public:
    green_teddy_bear();
};
Lets take a look!
Inheritance

- With Inheritance, we were able to use the same implementation of one method across many different kinds of objects, brought together through a parent-child relationship.
- Child subclasses inherit all methods and attributes. (constructors usually don't count here, depending on the language). They can choose to override parent functions (green bear roaring differently)
- Big concept in languages like Java (where everything inherits one base Object class)
What might be the weaknesses of Inheritance?
Inheritance Trees

A Change in DisplayObject could break implementations for the entire tree!
Questions?
Traits
How else can we decompose?

```rust
struct TeddyBear;
impl TeddyBear {
    fn roar(&self) {
        println!("ROAR!!");
    }
}

struct RedTeddyBear;
impl RedTeddyBear {
    fn roar(&self) {
        println!("ROAR!!");
    }
    fn red_button_song(&self){
        /* Red Song */
    }
}

struct PurpleTeddyBear;
impl PurpleTeddyBear {
    fn roar(&self) {
        println!("ROAR!!");
    }
    fn purple_button_song(&self){
        /* Purple Song */
    }
}

struct GreenTeddyBear;
impl GreenTeddyBear {
    fn roar(&self) {
        println!("ROAR!!");
    }
    fn green_button_song(&self){
        /* Green Song */
    }
}
```

Traits

Inject the code you want into the other classes! (Inject a trait into them!)
Let's make our first trait!
Traits Overview

- With traits, you write code that can be **injected** into any existing structure. (From TeddyBear to i32!) This code can have reference to **self**, so the code can be dependent on the instance.
- Trait methods do not need to be fully defined - you could define a function that must be implemented when implementing a trait for a type. (Similar to Java interfaces)
- No more deep inheritance hierarchies. Just think: "Does this type implement this trait?"
- Traits can specify functions instances **should** have, instead of just getting many from another "parent".
Advantages to Traits

- **Code-Reuse**: Want an object to be different based on the file it takes in? Create a *Trait* that has a parameterized function, and inject it to all objects!

- **Code-Hiding**: All parts of a trait are exposed, but because you specify which members / functions should be injected, there is no accidental spillover that inheritance structures can have!
Questions?
Big Standard Rust Traits
Traits to Know

- **Copy**: Will create a new copy of an instance, instead of moving ownership when using assignment (=)
- **Clone**: Will return a new copy of an instance when calling the `.clone()` function on the method.
- **Drop**: Will define a way to free the memory of an instance - called when the instance reaches the end of the scope.
- **Display**: Defines a way to format a type, and show it (used by println!)
- **Debug**: Similar to Display, though not meant to be user facing (Meant for you to debug your types!)
- **Eq**: Defines a way to determine equality (defined by an equivalence relation) for two objects of the same type.
- **PartialEq**: Defines a way to determine equality (defined by a partial equivalence relation) - f32!
Struct Point {
    x: u32,
    y: u32,
}

fn main() {
    let pt = Point {x: 3, y: 2};
    let pt2 = pt.clone();
}

Does not compile - clone() isn't defined
Let's Inject Clone!
Injecting Clone

- You can implement any traits into any structure (as we did with Clone to Point), so long as they are compatible (Drop is not compatible with Copy).
- You can use the Rust Documentation as a way to tell you which functions need to be implemented, along with their parameter types.
- You can use #[derive(x,y,z..)] to derive traits. The Rust compiler will try to implement the traits for you, if your structure satisfies some rules (given by the documentation). IE: You can derive Clone if all members in the struct already implement Clone.
Next Time [End]

- How can we write code that can accept many types?
- How can traits play a role in this?