The Beauty of Rust

Generics and Traits

Peter Zdankin

What will we do today?

- Deeper dive into Rusts type system
- Currently we are working with defined types
- But more often than not, we don’t *really* care about the type, but the behaviour of that type

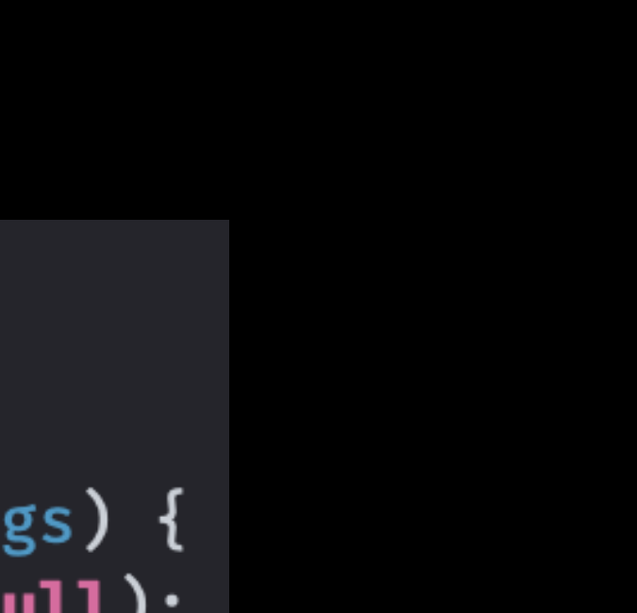
- For this we will look into how to write
  - that we don’t care about a type
  - that we do care about what a type can do

Billion Dollar Mistake

- Sir Charles A. R. Hoare introduced null references in ALGOL
- This carried on into most programming languages
- Object x = null;
  Object x = new Object();
- No distinction on a type level between actual object or empty reference
- Caused countless bugs, crashes etc.
- (Eventual) Solution: Wrapper types around Objects
- Optional<Object> x = Optional.empty();
  Optional<Object> x = Optional.of(new Object());

```java
import java.util.Optional;

public class Op{
  public static void main(String [] args) {
    Optional<Object> x = Optional.of(null);
    System.out.println(x);
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException
    at java.base/java.util.Objects.requireNonNull(Objects.java:208)
    at java.base/java.util.Optional.of(Optional.java:113)
    at Op.main(Op.java:5)
```



Problem

- We can put as many layers of abstractions around our null as we like
- We still can use null erroneously
- We need a way to make it clear if there is a value
- Compile-time checks

Strong Type System

- Rust has a very strong type system
- No null values in Rust!!!
- Why not use it to our advantage?

- Have an enum, which holds either our value or a None type

- How can we prevent writing this enum for each possible type?

Generics

Types are just a social construct

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

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

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

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

```
error[E0308]: mismatched types
 --> src/main.rs:7:38
  |
7 |     let wont_work = Point { x: 5, y: 4.0 };
  |                                      ^^^ expected integer, found floating-point number
  |
  = note: expected type `{integer}`
             found type `{float}`
```

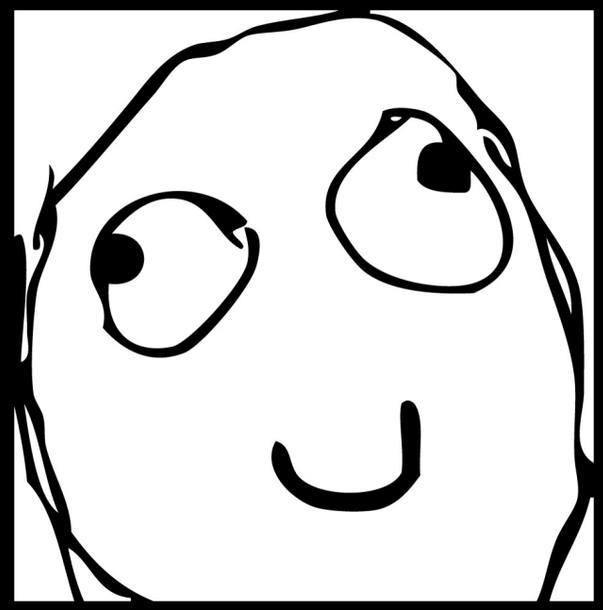
```rust
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

```rust
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

How to write generic functions?

- You write functions to remove duplication
- You write generic functions if you have the same algorithm for different types

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let mut largest = number_list[0];
    for number in number_list {
        if number > largest {
            largest = number;
        }
    }
    println!("The largest number is {}", largest);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];
    let mut largest = number_list[0];
    for number in number_list {
        if number > largest {
            largest = number;
        }
    }
    println!("The largest number is {}", largest);
}
```



Same same, lets write a fucntion!

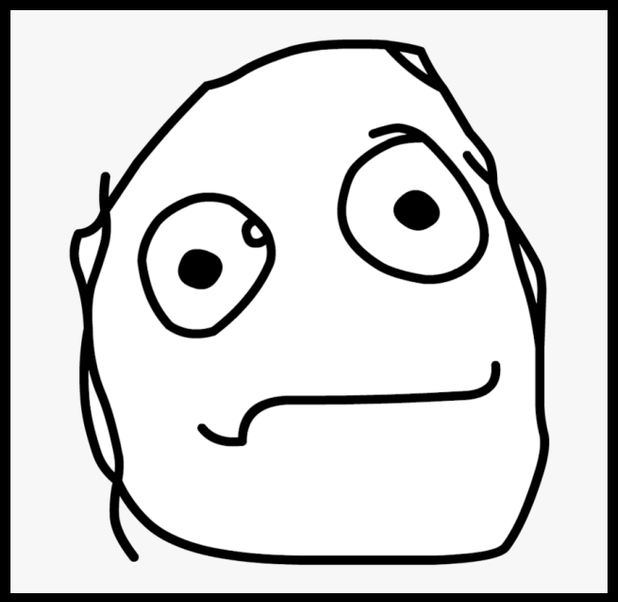
```rust
fn largest(list: &[i32]) -> i32 {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}

fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];
    let result = largest(&number_list);
    println!("The largest number is {}", result);
}
```

What if we have different types?

```rust
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

```rust
fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest_i32(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest_char(&char_list);
    println!("The largest char is {}", result);
}
```



Let’s write a generic function!

```rust
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];
    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

We have a &[T] so list[0] is defined

Same here

Is > defined on T?

```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];
    let result = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list = vec!['y', 'm', 'a', 'q'];
    let result = largest(&char_list);
    println!("The largest char is {}", result);
}
```

```
joy on □ master [?] via R v1.34.1 → cargo run
   Compiling joy v0.1.0 (/Users/peterzdankin/Projects/rust/joy)
error[E0369]: binary operation `>` cannot be applied to type `T`
 --> src/main.rs:5:12
  |
5 |         if item > largest {
  |            ^^^^^^^^^^^^^^
  |
  = note: `T` might need a bound for `std::cmp::PartialOrd`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0369`.
error: Could not compile `joy`.
```

Traits

Traits

- Define behaviour of types
- Describes what some type is able to do
- E.g. Display trait => Can be displayed
- E.g. PartialOrd trait => Implements the > operation

- Like interfaces in other languages
- A type may implement a trait

```rust
struct RoundPizza{
    price: f64,
    radius: f64,
}
```

```rust
trait Pricing{
    fn price_per_square_cm(&self) -> f64;
}
```

```rust
impl Pricing for RoundPizza{
    fn price_per_square_cm(&self) -> f64{
        let pi = core::f64::consts::PI;
        self.price / self.radius.powf(2.0) * pi
    }
}
```

```rust
fn main(){
    let pizza = RoundPizza{price: 6.0, radius: 17.0};
    println!("{}",pizza.price_per_square_cm());
}
```

Default implementations

```
trait Pricing{
    fn price_per_square_cm(&self) -> f64{
        2.99
    }
}

struct GrocerystorePizza{
}

impl Pricing for GrocerystorePizza{

}
```

Express that a type must implement a trait

```rust
fn notify(item: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}

fn notify<T: Summary>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Which option is better?

You tell me

```
fn notify(item1: impl Summary, item2: impl Summary) {
    println!("Breaking news! {}", item.summarize());
}
```

```
fn notify<T: Summary>(item1: T, item2: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Multiple Traits?

```
fn notify(item: impl Summary + Display) {
    println!("Breaking news! {}", item.summarize());
}
```

```
fn notify<T: Summary + Display>(item: T) {
    println!("Breaking news! {}", item.summarize());
}
```

Different types implement different traits?

```rust
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32
{
    println!("Breaking news!");
}
```

```rust
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug {
    println!("Breaking news!");
}
```

Going back to our sorting problem

```
fn largest<T: PartialOrd>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

```
joy on □ master [?] via R v1.34.1 → cargo run
   Compiling joy v0.1.0 (/Users/peterzdankin/Projects/rust/joy)
error[E0508]: cannot move out of type `[T]`, a non-copy slice
 --> src/main.rs:2:23
  |
2 |     let mut largest = list[0];
  |                       ^^^^^^^
  |                       |
  |                       cannot move out of here
  |                       help: consider using a reference instead: `&list[0]`

error[E0507]: cannot move out of borrowed content
 --> src/main.rs:4:9
  |
4 |     for &item in list.iter() {
  |         ^----
  |         ||
  |         |hint: to prevent move, use `ref item` or `ref mut item`
  |         cannot move out of borrowed content

error: aborting due to 2 previous errors
```

Copy or reference?

- We must annotate, whether this type can be copied, or must be referenced

```
fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

Exercises

- Write a function “position_of_number”, which returns either the index of a number in a long number, or None
(e.g. (1234, 2) == Some(1), (1234, 4) == Some(3), (1234, 5) = None

- Create a struct and implement the Display trait on it! Google how it works!

- You get a Vec of numbers. Find the largest unique number in this vec

- Create a trait Closeable, with the methods open and close.
Create two structs that implement this trait.
Try to store some Closeable objects in a vec.