Generic Types

Extracting Functions

- Just to give the idea of why we use functions anyways
- Functions are meant to prevent code duplication of the same logic
- For instance.

  ```
  let numbers = vec![1, 2, 3, 4, 5];

  let mut max  = 0;
  for i in numbers {
      if max < i {
          max = i;
      }
  }
  ```

  - Here we find the maximum value in a vector
  - You can imagine that we could be wanting to do this for more than one `Vec<i32>`
  - So let's abstract this to a function

- Example of function version:

  ```
  fn max(vector : &Vec<i32>) -> i32{
      let mut max  = 0;
      for i in vector {
          if max < *i {
              max = *i;
          }
      }
      max
  }
  ```

  - This now takes the reference of a vector and returns the maximum value. We don't want to take ownership so we took the reference
  - And since the vector is taken by reference so are the values, so we must dereference it

- Ok so this is a good **abstraction** step, but you can imagine that is may be even worse in rust specific
  - The `Vec<i32>` is specific to only `i32` types, but there are more types of integers, unsigned integers, floats. So then we need a max function for each. Which is very tedious
  - Beyond just numbers, we may want to find the max when comparing characters in a list or strings or more
  - This is why we need an extra layer of abstraction for **generic types**

Generic Function Arguments

- In order to use a generic type argument we need to specify that the function takes a generic
- Generic types are defined with `<>` and usually represented with `<T>` for "template"
  - Template meaning that the type can be anything
  - But if you use more than one argument, you can use any letter to represent a generic type
  - Such as: `<T, U, S...>`
- Let's try an example with the Max function: (this example cannot run)

```
fn max<T>(vector : &Vec<T>) -> T{
    let mut max  = &vector[0];
    for i in vector {
        if *max < *i {
            max = i;
        }
    }
    *max
}
```

  - We did a lot of good work here to make it more modular
  - We first changed the starting value to a reference from the original vector
  - We also added in a template type to handle any type
  - However, there is a big issue with this, not every type has a comparison operator
  - So we need to specify that we want any type that can be compared
  - To fix this problem we need to add a **trait** which we will discuss more later but essentially is a characteristic of a type.
  - In our case we need all types that can be **ordered** and **copied**
  - So we add it like this:

```
fn max<T: PartialOrd + Copy>(vector : &Vec<T>) -> T{
    let mut max  = &vector[0];
    for i in vector {
        if *max < *i {
            max = i;
        }
    }
    *max
}
```

- Here we can see that we added `PartialOrd + Copy` which means it needs to have the ordered and copy traits.
- Copy trait is being used here so if you remember: integers, floats, unsigned integers, chars, bools, are all copied by default
- This limits the *genericness* of the type to only the types of lists that have types which can be copied by default and compared against each other.

Generics With Structs and Enums

Structs

- You can imagine in a similar fashion we would want structs to have certain fields that are a Templates of types of a specific trait or group

- For instance if we want to have a `Point` struct, we would want to have the points to be of floats and integers

- Example:

```
struct Point<T,U>{
    x: T,
    y: U,
    z: u64,
}

let point1 = Point{ x: 10, y: 3.2, z: 0 };
let point2 = Point{ x: -21.2, y: 13.2, z: 3};
let point3 = Point{ x: -1.5, y: 10, z: 3};
```

  - As you can see we define the Struct with **two** generic types. `<T, U>`
  - These types can be the same but doesn't have to be
  - We also see that there can be more fields that don't have to be generic
  - We can see in the first point, that x is `i32`, and y is `f64`
  - The second point, they are both of the same type
  - And the third point the types are swapped

Enums

- We actually have seen Enums use Generic Types before with the `Option<T>` and `Result<T,E>` enums

- They are very useful for many different use cases

- Just a quick refresher on how they look like:

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

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

Generic Types with Methods

- Unlike normal functions, if you recall, methods are specific associated funcitions that apply to an instance of a struct or enum
- We use the `impl` block to form the associated functions for the Struct
- When using generics in the implementation block, you need to put the generic into scope
    - `impl <T, U, V...>`
- The only restriction is that the number of templat variables must match the same in the Struct or Enum
- Let's take a look back at the Point struct and lets say we want to add a method to get back a reference to one of the axis.

```rust
struct Point<T,U>{
    x: T,
    y: U,
    z: u64,
}

impl <V,W> Point<V, W>{
    fn x(&self) -> &V{
        &self.x
    }
}
```

    - We can see that just cause `Point<T,U>` is defined with `<T,U>`, the implementation block is not tied to the template from the definition
    - Instead we use `<V,W>`
    - To make this idea a little more clear take a look at the following

````
```rust
impl <X> Point<X, f64>{
    fn y(&self) -> &f64{
        &self.y
    }
}
```
````

        - This shows that within an implementation block, we provide concrete types or any types that we choose to have
        - This gives us more flexibility with how we want to define a group of methods within an implementation block
        - Even though the type of `y` or `x` is generic, we can specify that a group of methods require a specific type, group of types, or any type

- Now that we looked into the `impl` block, we can also specify types better within the method itself.

- Let's say we want to make a method, that mixes up the values of two points and returns a third point

- Example:

```
impl <V,W> Point<V, W>{
    fn x(&self) -> &V{
        &self.x
    }

    fn mixup<X,Y>(self, other: Point<X,Y>) -> Point<V, Y>{
        Point{
            x : self.x,
            y : other.y,
            z : other.z
        }
    }
}
```

    - Read into this carefully, and its best to follow the types
    - First we define two types for the instance of our `Point` as `<V, W>` in our implementation block meaning that all objects of this Struct that uses this `impl` block will have their types referenced as `<V, W>`.
        - In other words the `self` parameter will have type `<V, W>`
    - Then we see that the `mixup` method has two other generics defined as `<X, Y>` and these types are used for the second parameter's generics
        - Here we are saying that the generic type of the `Point<X,Y>` **may or may not** match the type used for the implementation block
    - Lastly we need to specify the type of the returned `Point` struct.
    - The returned type is of `Point<V, Y>`
        - We can see that the returned object can be a mix of the types from the implementation block and the method block
        - Even without knowing the logic of the method, we can see that whatever the type for `self.x` and `other.y` is, they will convey into the new Point's x and y coordinates.
    - A small example of how it can be used: `rust let p1 = point2; let p2= Point {x : "Hello", y: 'c', z : 0}; let p3 = p1.mixup(p2); println!("mix ups -> {:?}", p3);`

Performace

- So it would be worth while to discuss how performance is impacted with the Generic Types
- Similar to Java, Generics in Rust are determined in compile time
- This means that once the compiler knows what type you want to use for the generic, it will clone the code for each type use case and then run it
- This means the error is found in compile time and that the speed of running the program is not affected
- For instance if you use Option enum for the `i32` and `i64` types, the compiler will create the following automatically:

  ```
  enum Option<i32>{
      Some<i32>,
      None,
  }
  enum Option<i64>{
      Some<i64>,
      None,
  }
  ```

  - This doesn't take any extra time from the run time to determine types or create this difference
  - As for space, it is the same amount of space as if you never used the generic type and made a separate enum, struct, or function for each type
- Generics are more for the user and to help with developement time for the code.