Assignment 2: Generics in RUST

Rust Generics

Generics is the topic of generalizing types and functionalities to broader cases. Generics is useful as it helps reduce duplicated code. However, using generics requires taking great care to specify over which types a generic type is actually considered valid. The simplest and most common use of generics is for type parameters. For example, defining a generic function named myFunction that takes an argument T of any type:

```
fn myFunction<T>(arg: T) { ... }
```

Although types in Rust are erased, generics are also reified. Consider the following code:

```
struct Bag<T> {
    t: T,
}

fn main() {
    let b = Bag { t: 42 };
}
```

A Bag<u8> is not the same type as a Bag<u32>. Although it implements the same methods, it has a different size.

Question 1: Rewrite the Bag struct to hold an array of 3 items of type T.

Question 2: Write a function named *BagSize* that takes a Bag as an input and returns the size of a Bag.

Test your function with the following code:

```
let b1 = Bag {items: [1u8, 2u8, 3u8], };
let b2 = Bag {items: [1u32, 2u32, 3u32], };
```

The code should give the following output:

```
size of First Bag = 3 bytes
size of Second Bag = 12 bytes
```

Question 3: Rewrite the code for b1 and b2 without using any generics to produce the same output. [Hint: you will have to duplicate a lot of code]

However, monomorphization (https://en.wikipedia.org/wiki/Monomorphism) increases binary size, which is considered as its negative aspect. For example, if the Bag struct defined above has a lot of methods which uses a lot of different Ts, this will result in creating a lot of variants. Imagine if we have multiple generic types!

Question 4: Consider the following code:

```
let vec1 = vec![12, 32, 13];
let vec2 = vec![44, 55, 16];
{ let vec1_iter = vec1.iter(); }
{let vec_chained = vec1.iter().chain(vec2.iter());}
{ let vec1_2=vec![vec1, vec2];
  let vec_flattened = vec1_2.iter().flatten(); }
```

Similar to Question2, report the sizes of vec1_iter, vec_chained, and vec_flattened.

Question 5: Run the code implemented in Question 4 but after boxing (https://doc.rust-lang.org/std/boxed/struct.Box.html) all three of the iterators. Again, report the sizes of vec1_iter, vec_chained, and vec_flattened.

Question 6: Explain the results obtained in Question 4 and 5 while illustrating the contents of both the stack and the heap in each case. [i.e., How does Rust model the iterators in each case?]

Question 7: What is polymorphism? Does Rust support polymorphism?

Working with the Compiler Explorer

Now, let's explore one advantage of monomorphization. Since, in this case, the compiler knows the instantiated types ahead of time, this generates a lot of opportunities for optimizing code. Consider the following code:

```
fn equal<T>(x: T, y: T) -> bool
where T: PartialEq,
{
    x == y
}
fn compare(x: &str, y: &str) {
  equal(x, y);
  equal(x.len(), y.len());
}
pub fn main() {
    let (x, y) = ("3","2");
    compare(&x, &y);
    println!("x = {}, y = {}", x, y);
}
```

Question 8: Analyze the code using the compiler explorer (https://godbolt.org/) and report how many times the function *equal* has been called (Please mention the lines numbers). (Hints: The highlighted Rust code is equivalent to the same coloured highlighted assembly code. Also, equal is being called whenever you see "*call example::equal*").

Question 9: Now, use the optimization flag -O and recompile the code. How many times has the function equal been called? Explain what happened.