ECE 421
Assignment 2

Arun Woosaree
XXXXXXX

February 2, 2020

1

```
struct Bag<T> {
 items: [T; 3],
}
```

2

```
#[allow(non_snake_case)]
fn BagSize<T>(_bag: Bag<T>) -> usize {
    std::mem::size_of::<Bag<T>>()
}
```

3

```
#[allow(non_camel_case_types)]
struct Bag_u8 {
 items: [u8; 3],
}

#[allow(non_camel_case_types)]
struct Bag_u32 {
 items: [u32; 3],
}

#[allow(non_snake_case)]
fn BagSize_u8 (_bag: Bag_u8) -> usize{
    std::mem::size_of::<Bag_u8>()
}
```

```rust
#[allow(non_snake_case)]
fn BagSize_u32 (_bag: Bag_u32) -> usize{
    std::mem::size_of::<Bag_u32>()
}

fn main() {
    let b1 = Bag_u8 {items: [1u8, 2u8, 3u8], };
    let b2 = Bag_u32 {items: [1u32, 2u32, 3u32], };

    println!("size of First Bag = {} bytes", BagSize_u8(b1));
    println!("size of Second Bag = {} bytes", BagSize_u32(b2));
}
```

4

```rust
use std::mem::size_of_val;

fn main() {

    let vec1 = vec![12, 32, 13];
    let vec2 = vec![44, 55, 16];
    {
        let vec1_iter = vec1.iter();
        println!("size of vec1_iter = {} bytes", size_of_val(&vec1_iter));
    }
    {
        let vec_chained = vec1.iter().chain(vec2.iter());
        println!("size of vec_chained = {} bytes", size_of_val(&vec_chained));
    }
    {
        let vec1_2=vec![vec1, vec2];
    println!("{}", size_of_val(&vec1_2));
        let vec_flattened = vec1_2.iter().flatten();
        println!("size of vec_flattened = {} bytes", size_of_val(&vec_flattened));
    }

}

/* Output:
size of vec1_iter = 16 bytes
size of vec_chained = 40 bytes
size of vec_flattened = 64 bytes
*/
```

5

```rust
use std::mem::size_of_val;

fn main() {

    let vec1 = Box::new(vec![12, 32, 13]);
    let vec2 = Box::new(vec![44, 55, 16]);
    {
        let vec1_iter = Box::new((*vec1).iter());
        println!("size of vec1_iter = {} bytes", size_of_val(&vec1_iter));
    }
    {
        let vec_chained = Box::new((*vec1).iter().chain((*vec2).iter()));
        println!("size of vec_chained = {} bytes", size_of_val(&vec_chained));
    }
    {
        let vec1_2=Box::new(vec![*vec1, *vec2]);
        let vec_flattened = Box::new((*vec1_2).iter().flatten());
        println!("size of vec_flattened = {} bytes", size_of_val(&vec_flattened));
    }

}

/* Output:
size of vec1_iter = 8 bytes
size of vec_chained = 8 bytes
size of vec_flattened = 8 bytes
*/
```

6

todo 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?]

in number 5 the values are stored on the heap, and we are printing the size of the pointer which points to the area on the heap. That's why in number 5, the size of the value is always 8 bytes

7

Polymorphism is the ability to process objects differently depending on their data type while still presenting the same interface.

Rust supports polymorphism, but in a dffernt way than you're used to programming if you're used to Java. According to the Rust book: "Rust instead

uses generics to abstract over different possible types and trait bounds to impose constraints on what those types must provide. This is sometimes called bounded parametric polymorphism." https://doc.rust-lang.org/1.30.0/book/2018-edition/ch17-01-what-is-oo.html

8

Polymorphism is the ability to process objects differently depending on their data type while still presenting the same interface.

Rust supports polymorphism, but in a dffernt way than you're used to programming if you're used to Java. According to the Rust book: "Rust instead uses generics to abstract over different possible types and trait bounds to impose constraints on what those types must provide. This is sometimes called bounded parametric polymorphism." https://doc.rust-lang.org/1.30.0/book/2018-edition/ch17-01-what-is-oo.html

9

It was called 2 times. Line numbers in the output: 1801 and 1812

10

With the -O flag, it was called 0 times. I'm guessing it was inlined instead.