

ECE 421

Assignment 2

Arun Woosaree

February 2, 2020

1

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

2

```
#[allow(non_snake_case)]
fn BagSize<T>(_: 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_u8) -> usize{
    std::mem::size_of::<Bag_u8>()
}

#[allow(non_snake_case)]
```

```

fn BagSize_u32 (_: 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

```

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];
        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

```

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

a)

In question 4, most of the variables are on the stack, while the values in the vectors are on the heap. Below is an attempt to demonstrate this. The iterators are basically structs containing references to the actual values on the heap, coupled with something that allows the iterator to keep track of its state, so it knows where to go next.

Stack

Heap

Vec1

Vec ϵ

rawvec

length

3

12

32

13

Vec2

Vec ϵ

rawvec

length

3

44

55

16

VecIterator

(16 bytes)

iterator

current

end

Vec_chained chain ϵ

(40 bytes)

a
b
chain state

Vec1_2

(24 bytes)

Vec ϵ

rawvec

length

3

12

32

13

44

55

16

Vec_flattened

(64 bytes)

Flatten ϵ

FlattenCompart

3

b)

In question 5, almost everything is stored on the heap. Structs that were on the stack in question 4 are now instead on the heap, and we have pointers on the stack pointing to them. That's why in number 5, the size of the value is always 8 bytes, since 8 byte are needed to point to a location in memory on a 64-bit machine.

Stack

Box(vec1)

heap

vec1

12

32

13

Vec^E
RawVec
length

Box(vec2)

vec2

44

55

16

Vec^E
RawVec
length

Box(vec1_iter)

Iterator^E

current
end

}

Box(vec-channel)

chan^E

a

b

chanState

}

Box(vec1_2)

vec1_2

12

32

13

44

55

16

Box(vec-flatten)

flatten

flattenCompat

}

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 different 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

It was called 2 times. On lines 1801 and 1812 in the output, we see:

```
call    example::equal
```

9

With the -O flag, it seems like the equal function was called 0 times. It is likely that the compiler inlined the function to produce faster code.