ECE 522 Assignment 2

Zhaoyi Wang 1689747

Question 1

```rust
fn main() {
    let the_bag = Bag {
        list: [1, 2, 3],
    };

    println!("{:?}", the_bag)
}

#[derive(Debug)]
struct Bag<T> {
    list: T,
}
```

The output is:

```
Bag { list: [1, 2, 3] }
```

Question 2

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

impl<T> Bag<T> {
    fn BagSize(&self) -> usize {
        let mut length = 0;
        for elem in self.items.iter() {
            length = length+std::mem::size_of_val(elem);
        }
        length
    }
}

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

```
    };

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

The output is:

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

Question 3

```
fn main() {

    let b1 = u8_bag{
        items:[1u8,2u8,3u8],
    };

    println!("size of First Bag = {} bytes",b1.get_length());

    let b2 = u32_bag{
        items:[1u32, 2u32, 3u32],
    };

    println!("size of Second Bag = {} bytes",b2.get_length());
}

// For b1 - u8 case
struct u8_bag{
    items:[u8;3],
}

impl u8_bag {
    fn get_length(&self) -> usize {
        let mut length = 0;
        for elem in self.items.iter() {
            length = length + std::mem::size_of_val(elem);
        }
        length
    }
}

// For b2 - u32 case
struct u32_bag{
    items:[u32;3],
}

impl u32_bag {
    fn get_length(&self) -> usize {
        let mut length = 0;
        for elem in self.items.iter() {
            length = length + std::mem::size_of_val(elem);
        }
        length
    }
```

```
}
```

Question 4

```
fn main() {
    let vec1 = vec![12, 32, 13];
    let vec2 = vec![44, 55, 16];

    {
        let vec1_iter = vec1.iter();
        println!("vec1_iter: {}",std::mem::size_of_val(&vec1_iter));
    }

    {
        let vec_chained = vec1.iter().chain(vec2.iter());
        println!("vec_chained: {}",std::mem::size_of_val(&vec_chained));
    }

    {
        let vec1_2 = vec![vec1, vec2];
        let vec_flattened = vec1_2.iter().flatten();
        println!("vec_flattened: {}",std::mem::size_of_val(&vec_flattened));
    }
}
```

The output is:

```
vec1_iter: 16
vec_chained: 32
vec_flattened: 48
```

Question 5

```
fn main() {
    let vec1 = vec![12, 32, 13];
    let vec2 = vec![44, 55, 16];
    {
        let vec1_iter = Box::new(vec1.iter()) ;
        println!("vec1_iter after box: {}",std::mem::size_of_val(&vec1_iter));
    }
    {
        let vec_chained = Box::new(vec1.iter()).chain(Box::new(vec2.iter()));
        println!("vec_chained after box: {}",std::mem::size_of_val(&vec_chained));
    }
    {
        let vec1_2 = vec![vec1, vec2];
        let vec_flattened = Box::new(vec1_2.iter()).flatten();
        println!("vec_flattened after box: {}",std::mem::size_of_val(&vec_flattened));
    }
}
```

The output is:

```
vec1_iter after box: 8
vec_chained after box: 16
vec_flattened after box: 40
```

Question 6

We can see that, when we are using `Box::new()`, the memory space it uses is less than the `.iter()`. This is because the `Box` stores data on the heap rather than stack. Saving data on the heap will spend more time than stack, but the memory usage is allocated by the program.

Question 7

Polymorphism is the ability (in programming) to present the same interface for differing underlying forms (data types).

For example, in many languages, integers and floats are implicitly polymorphic since you can add, subtract, multiply and so on, irrespective of the fact that the types are different. They're rarely considered as objects in the usual term.

In Rust, this approach leverages "Trait Objects" and "Generics" to achieve polymorphism.

Question 8

The `equal()` function was called twice. Both of them happened in `line 12: compare(&x, &y)`.

Question 9

Zero times. This is because it doesn't call `compare(&x, &y)`.