## Assignment_2

**Question 1**

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

**Question 2**

```
#![feature(core_intrinsics)]
use std::fmt::Display;
use std::mem;
use std::intrinsics::size_of;

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

fn Size<T>(stru:&Bag<T>) -> usize {
    let mut byte_nums = 0;
    for element in stru.items.iter() {
        byte_nums = byte_nums + mem::size_of_val(element);
    }
    byte_nums

}

fn main() {
    let b1 = Bag {items: [1u8,2u8,3u8],};
    let b2 = Bag {items: [1u32,2u32,3u32],};
    println!("size of First Bag = {} bytes",Size(&b1));
    println!("size of Second Bag = {} bytes",Size(&b2));
}
```

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

**Question 3**

```
#![feature(core_intrinsics)]
use std::fmt::Display;
use std::mem;
use std::intrinsics::size_of;

struct  Bag_1 {
```

```
    items: [u8; 3],
}

struct  Bag_2 {
    items: [u32; 3],
}
fn Size_1(stru:&Bag_1) -> usize {
    let mut byte_nums = 0;
    for element in stru.items.iter() {
        byte_nums = byte_nums + mem::size_of_val(element);
    }
    byte_nums
}

fn Size_2(stru:&Bag_2) -> usize {
    let mut byte_nums = 0;
    for element in stru.items.iter() {
        byte_nums = byte_nums + mem::size_of_val(element);
    }
    byte_nums
}

fn main() {
    let b1 = Bag_1 {items: [1u8,2u8,3u8],};
    let b2 = Bag_2 {items: [1u32,2u32,3u32],};
    println!("size of First Bag = {} bytes",Size_1(&b1));
     println!("size of Second Bag = {} bytes",Size_2(&b2));
}
```

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

**Question 4**

```
#![feature(core_intrinsics)]
use std::fmt::Display;
use std::mem;
use std::intrinsics::size_of;

fn main() {
    let vec1 = vec![12,32,13];
    let vec2 = vec![44,55,16];
    {
        let vec1_iter = vec1.iter();
        let a = mem::size_of_val(&vec1_iter);
```

```
        println!("The size of vec1_iter is:{}",a);
    }

    {
        let vec_chained = vec1.iter().chain(vec2.iter());
        let b =mem::size_of_val(&vec_chained);
        println!("The size of vec_chained is:{}",b);
    }

    {
        let vec1_2 = vec![vec1,vec2];
        let vec_flattened = vec1_2.iter().flatten();
        let c = mem::size_of_val(&vec_flattened);
        println!("The size of vec_flattened is:{}",c);
    }

}
```

```
The size of vec1_iter is:16
The size of vec_chained is:32
The size of vec_flattened is:48
```

**Question 5**

```
#![feature(core_intrinsics)]
use std::fmt::Display;
use std::mem;
use std::intrinsics::size_of;

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

    {
        let vec1_iter = Box::from(vec1.iter());
        println!("The size of vec1_iter is:{}
bytes",mem::size_of_val(&vec1_iter));
    }

    {
        let vec_chained =
Box::from(vec1.iter()).chain(Box::from(vec2.iter()));
        println!("The size of vec_chained is:{}.
                  bytes",mem::size_of_val(&vec_chained));
```

```
    }

    {
        let vec1_2 = vec![vec1,vec2];
        let vec_flattened = Box::from(vec1_2.iter()).flatten();
        println!("The size of vec_flattened
                 is:{}bytes",mem::size_of_val(&vec_flattened));
    }
}
```

```
The size of vec1_iter is:8 bytes
The size of vec_chained is:16 bytes
The size of vec_flattened is:40 bytes
```

**Question 6**

We can find that the size of the boxing iterators are different from iterators, because boxing can store the data on the heap instead of stack. Storing data on heap need more time than storing on stack, but the boxing iterator's size is smaller than iterator. I think that means storing data on heap need longer time but smaller size.

**Question 7**

In object-oriented language, polymorphism is the interfaces' different ways implementation, in simple terms, it is the ability to pass an argument that can be of different types, to a function with only one implementation, instead of separate ones of each different type.

Rust can implement polymorphism through trait and generics. We can define a trait which has certain behaviors that are similar, but can be different in implementation. Because of the different implementation, we can define different methods, then we make a generic function, the different type variables can find the correct implementation. That's how rust implement polymorphism.

**Question 8**

The equal function has been called two times

The lines numbers in the assembly code is 1543 and 1554。
The lines numbers in the Rust code is 13.

```
13    compare(&x, &y);
```

**Question 9**

Zero times
Because the main called `Option::unwrap()` on a `None` value', it will panic and quit, the compare function is not called.