제네릭, 트레잇, 라이프타임

2020.3

제네릭 타입

- 중복된 함수의 구현을 간단히.

Option<T>, Vec<T>, HashMap<K, V>, Result<T, E>

제네릭의 예~1

```rust
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn largest_char(list: &[char]) -> char {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest_i32(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest_char(&chars);
    println!("The largest char is {}", result);
}
```

```rust
fn largest<T>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

제네릭의 예~2

```
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

```
fn main() {
    let wont_work = Point { x: 5, y: 4.0 }; //  T 라는 동일 타입이어야 함. 컴파일 에러..
}
```

아래와 같이 해야 한다.

```
struct Point<T, U> {
    x: T,
    y: U,
}

fn main() {
    let both_integer = Point { x: 5, y: 10 };
    let both_float = Point { x: 1.0, y: 4.0 };
    let integer_and_float = Point { x: 5, y: 4.0 };
}
```

단형성화

```
let integer = Some(5);
let float = Some(5.0);

enum Option_i32 {
    Some(i32),
    None,
}

enum Option_f64 {
    Some(f64),
    None,
}

fn main() {
    let integer = Option_i32::Some(5);
    let float = Option_f64::Some(5.0);
}
```

트레잇: 공유동작

- 만일 우리가 서로 다른 타입에 대해 모두 동일한 메소드를 호출할 수 있다면 이 타입들은 동일한 동작을 공유하는 것입니다.

```
pub trait Summarizable {
    fn summary(&self) -> String;
}
```

다양한 타입에 대해 summary함수를 호출하고자 할때. pub가 없음을 주의.

트레잇 구현

```
pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summarizable for NewsArticle {
    fn summary(&self) -> String {
        format!("{}, by {} ({})", self.headline, self.author, self.location)
    }
}

pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summarizable for Tweet {
    fn summary(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

트레잇의 기본구현

```
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("(Read more…)”) // 기본 구현
    }
}

impl Summarizable for NewsArticle {} // 이렇게 하면 기본구현을 사용한다.
```

트레잇 바운드

```
pub fn notify<T: Summarizable>(item: T) { // 타입 T는 Summarizable을 구현한 타입으로 제한
    println!("Breaking news! {}", item.summary());
}
```

Where 를 사용하여 이쁘게 하기

```
fn some_function<T: Display + Clone, U: Clone + Debug>(t: T, u: U) -> i32 {
```

```
fn some_function<T, U>(t: T, u: U) -> i32
    where T: Display + Clone,
          U: Clone + Debug
{
```

컴파일 가능

```
use std::cmp::PartialOrd;

fn largest<T: PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest = list[0];

    for &item in list.iter() {
        if item > largest {
            largest = item;
        }
    }

    largest
}

fn main() {
    let numbers = vec![34, 50, 25, 100, 65];

    let result = largest(&numbers);
    println!("The largest number is {}", result);

    let chars = vec!['y', 'm', 'a', 'q'];

    let result = largest(&chars);
    println!("The largest char is {}", result);
}
```

라이프타임~1

- dangling reference

```
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r);
}
```

```
{
    let r;                // -------+-- 'a
                          //        |
    {                     //        |
        let x = 5; // -+-----+-- 'b
        r = &x;    //  |     |
    }              //-+     |
                          //        |
    println!("r: {}", r);  // |
                          //        |
                          // -------+
}
```

```
fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

라이프타임~2

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

```
error: `string2` does not live long enough
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ------- borrow occurs here
7 |     }
  |     ^ `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
9 | }
  | - borrowed value needs to live until here
```

라이프타임~3

```
struct ImportantExcerpt<'a> {
    part: &'a str, // 컴파일러가 반드시 라이프타임을 선언하도록 강제하므로.
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence }; // 댕글링 레퍼런스가 없으므로 컴파일 된다.
}

let s: &'static str = "I have a static lifetime.”; // 스트링 리터럴은 텍스트에 저장된다.

use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str // 모두 함께 써본 예제.
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```