한 번에 끝내는
블록체인 개발 A to Z

Chapter 2

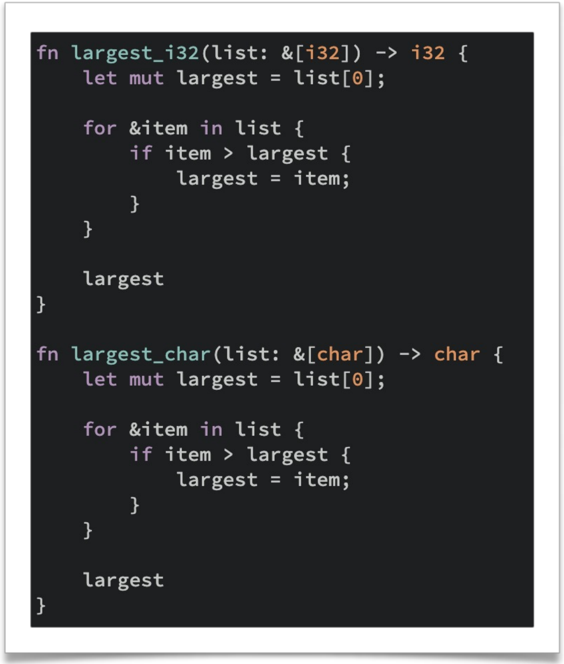
Rust Advanced

Chapter 2

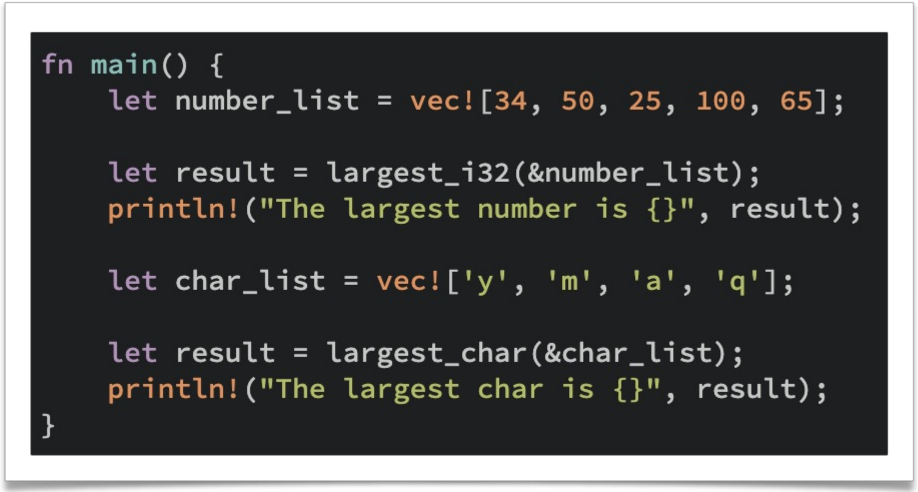
Rust Advanced

Generic Data Types

In Function Definitions

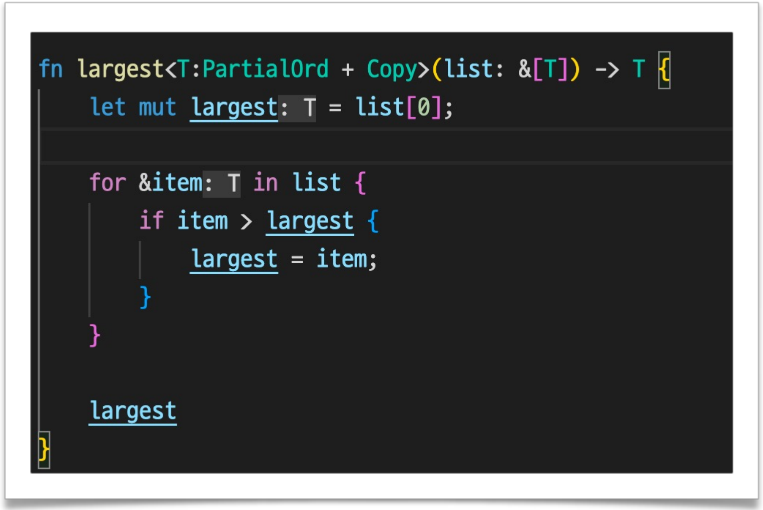


```rust
fn largest_i32(list: &[i32]) -> i32 {
    let mut largest = list[0];

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}

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

    for &item in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

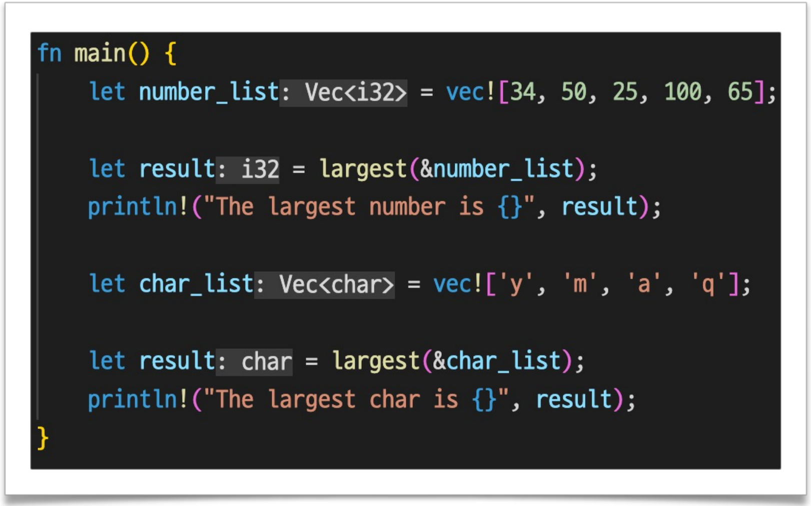


```rust
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

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

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

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

In Function Definitions

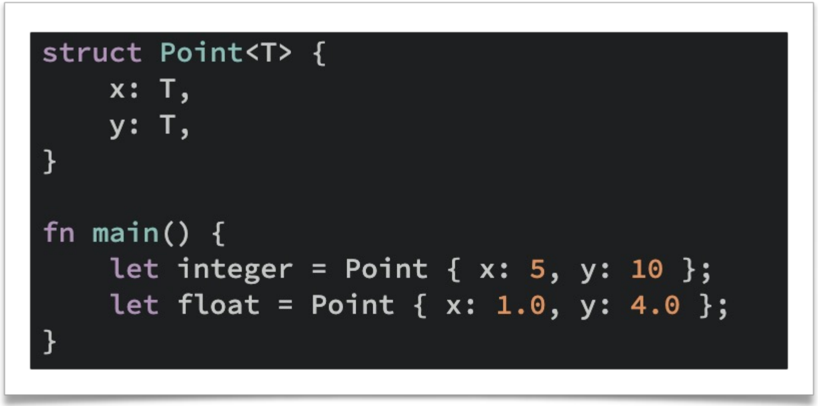


```rust
fn largest<T:PartialOrd + Copy>(list: &[T]) -> T {
    let mut largest: T = list[0];

    for &item: T in list {
        if item > largest {
            largest = item;
        }
    }

    largest
}
```

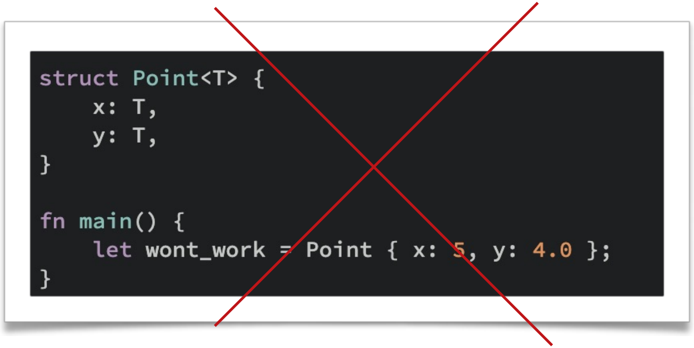


```rust
fn main() {
    let number_list: Vec<i32> = vec![34, 50, 25, 100, 65];

    let result: i32 = largest(&number_list);
    println!("The largest number is {}", result);

    let char_list: Vec<char> = vec!['y', 'm', 'a', 'q'];

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

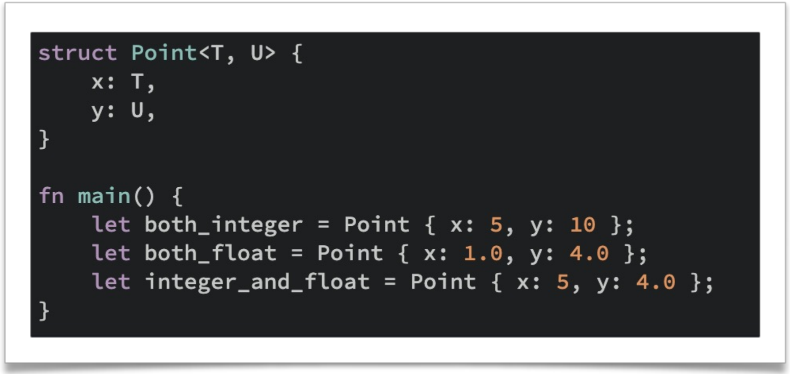
In Struct Definitions



```rust
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 };
}
```

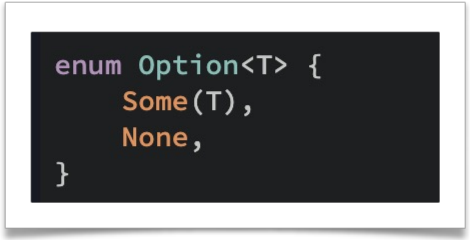


```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let wont_work = Point { x: 5, y: 4.0 };
}
```

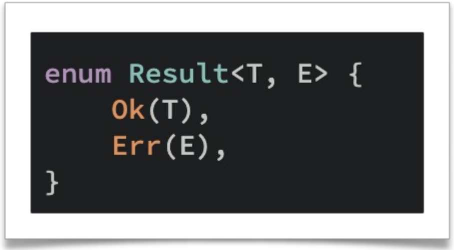


```rust
struct Point<T, U> {
    x: T,
    y: U,
}

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 };
}
```

In Enum Definitions

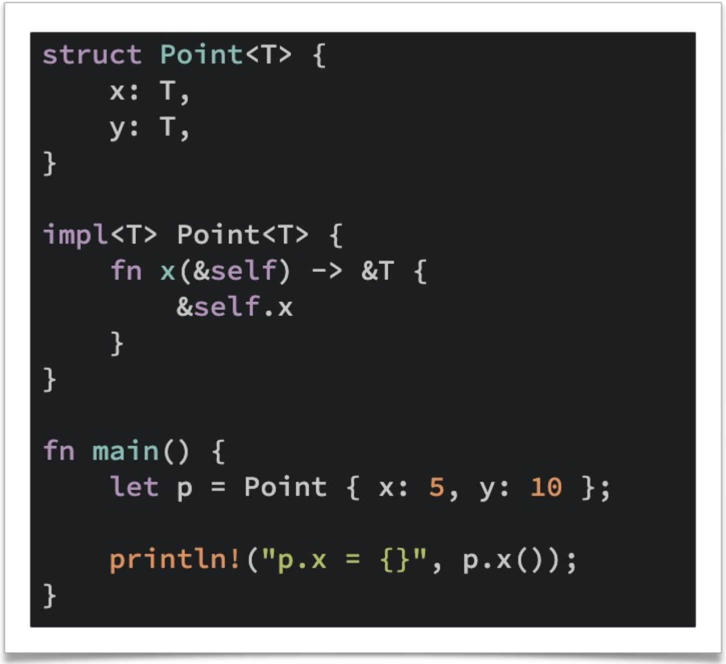


```
enum Option<T> {
    Some(T),
    None,
}
```

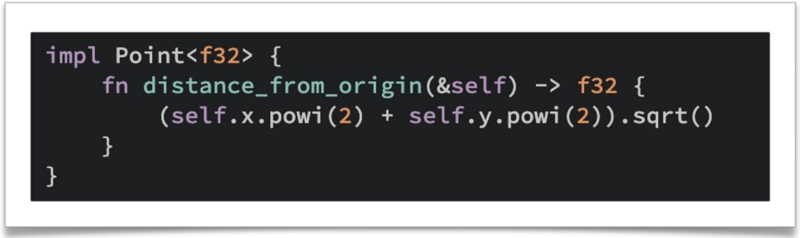


```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

In Method Definitions



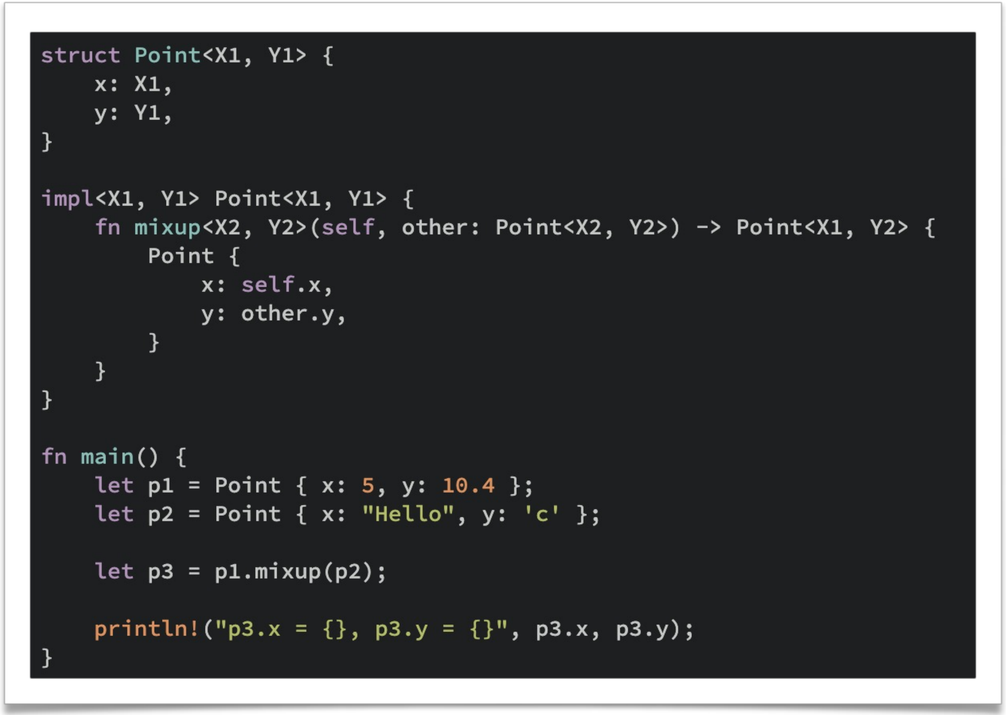
```
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}

fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```



```
impl Point<f32> {
    fn distance_from_origin(&self) -> f32 {
        (self.x.powi(2) + self.y.powi(2)).sqrt()
    }
}
```

We can also specify constraints on generic types when defining methods on the type. This code means the type **Point<f32>** will have a distance_from_origin method; other instances of **Point<T>** where **T** is not of type **f32** will not have this method defined.

In Method Definitions



```rust
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```