Rust 程序语言设计 assignment1

姓名：向胤兴 学号： 2215012469 班级：计算机2102

1. B
2. B
3. D

4、D

5、B

6、A

7、C

8、A

9、D

10、A

11、B

12、

从内存安全的角度来看，Rust采用了独特的所有权系统和生命周期管理，确保内存安全，C++的内存管理相对复杂，需要程序员手动管理内存。内存错误大都会在编译时报错，能够帮助程序员告诉有效的定位到错误产生的原因，降低Debug难度。

在性能方面，C++作为接近底层的高级语言，本身运行速度很快。但Rust借助于其所有权系统和零成本抽象的设计，可以在编译时进行大量的优化。它的内存安全性和并发性特性使得Rust能够生成高度优化的机器码，同时减少了运行时的开销，标准Rust性能与标准C++性能不相上下。

13、

所有权(ownership)是Rust编程中一个特有的概念。这是Rust语言确保内存安全的核心机制。在Rust中，每个值都有一个明确的所有者，当所有者离开作用域时，其占用的内存会被自动释放。这种机制避免了手动内存管理可能带来的问题，如内存泄漏和悬挂指针。

14、

Rust数据交互方式：

1. **变量声明与赋值**

在 Rust 中，可以使用 let 关键字来声明变量并为其赋值。

示例：

解释

**rust**

|  |  |
| --- | --- |
|  | let x = 5; |
|  | let y: i64 = 10; |

1. **可变变量**

默认情况下，Rust 中的变量是不可变的，这有助于编写更安全的代码。但如果你确实需要一个可变的变量，可以使用 mut 关键字。

示例：

解释

**rust**

|  |  |
| --- | --- |
|  | let mut z = 20; |
|  | z = 30; |

1. **模式匹配与解构**

Rust 支持模式匹配和解构，这允许你同时从复杂的数据结构中提取多个值。

示例（使用元组）：

解释

**rust**

|  |  |
| --- | --- |
|  | let point = (4, 3); |
|  | let (x, y) = point; // 解构元组，x = 4, y = 3 |

示例（使用结构体）：

解释

**rust**

|  |  |
| --- | --- |
|  | struct Person { |
|  | name: String, |
|  | age: u8, |
|  | } |
|  |  |
|  | let person = Person { name: "Alice".to\_string(), age: 30 }; |
|  | let Person { name, age } = person; |

1. **引用与借用**

Rust 使用所有权和借用系统来管理内存。通过引用（&）和可变引用（&mut），你可以在不复制数据的情况下访问和操作数据。

示例：

解释

**rust**复制代码

|  |  |
| --- | --- |
|  | let arr = [1, 2, 3, 4, 5]; |
|  | let ref\_to\_arr = &arr; // 不可变引用 |
|  | let mut\_ref\_to\_arr = &mut arr; // 可变引用 |

需要注意的是，Rust 有一个重要的规则：在任意给定时间，你只能有一个可变引用到一个数据，或者可以有多个不可变引用，但不能同时有可变引用和不可变引用。  
5. **枚举与模式匹配**

Rust 的枚举类型允许你定义一组命名的值。你可以使用模式匹配与枚举值进行交互。

示例：

解释

**rust**复制代码

|  |  |
| --- | --- |
|  | enum Message { |
|  | Quit, |
|  | Move { x: i32, y: i32 }, |
|  | Write(String), |
|  | } |
|  |  |
|  | let msg = Message::Write("Hello".to\_string()); |
|  |  |
|  | match msg { |
|  | Message::Quit => println!("Quitting"), |
|  | Message::Move { x, y } => println!("Moving to ({}, {})", x, y), |
|  | Message::Write(text) => println!("Writing: {}", text), |
|  | } |

1. **函数与参数**

函数是 Rust 中与变量和数据交互的重要工具。你可以通过参数将数据传递给函数，并通过返回值将数据从函数中带出。

示例：

解释

**rust**

|  |  |
| --- | --- |
|  | fn add(x: i32, y: i32) -> i32 { |
|  | x + y |
|  | } |
|  |  |
|  | let sum = add(5, 3); // sum = 8 |

15、

use rand**::**Rng;

**fn** process\_vector(val**:** **&***mut* Vec<i32>) **->** (i32, i32) {

    let is\_first\_element\_one **=** val**.**first() **==** Some(**&**1);

    println!("Is the first element 1? {}", is\_first\_element\_one);

*// 生成一个介于 10 到 20 之间的随机整数*

    let *mut* rng **=** rand**::**thread\_rng();

    let random\_num **=** rng**.**gen\_range(10**..=**20);

    val**.**push(random\_num);

*// 对向量进行排序*

    val**.**sort();

    println!("Sorted vector: {:?}", val);

*// 返回向量中的最大值和最小值*

    let min\_val **=** **\***val**.**first()**.**unwrap(); *//unwrap取出Option<&i32>的内部&i32值，最后\*解引用*

    let max\_val **=** **\***val**.**last()**.**unwrap();

**return** (max\_val, min\_val);

}

**fn** main() {

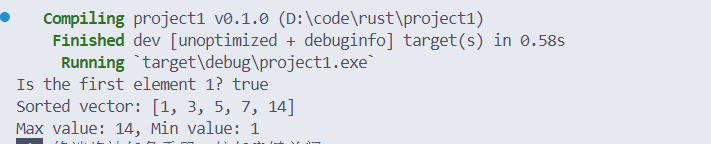
    let *mut* vec **=** vec![1, 3, 5, 7];

    let (max, min) **=** process\_vector(**&***mut* vec);

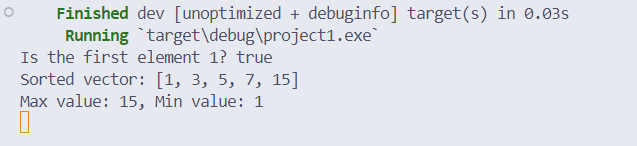
    println!("Max value: {}, Min value: {}", max, min);

}

运行结果：



再次运行：



可以看出是随机生成。

16、

struct Counter{

    count**:**i32

}

impl Counter{

**fn** new()**->***Self*{

*Self*{count**:**0}

    }

**fn** increate(**&***mut* *self*){

*self***.**count**+=**1;

    }

}

**fn** add\_two(counter**:&***mut* Counter,*mut* num**:**i8){

    num**+=**2;

    counter**.**increate();

    println!("The number is {num} after add\_two");

    println!("The func add\_two has been used {0} times",counter**.**count);

}

**fn** main(){

    let number**:**i8**=**10;

    let *mut* cnt**:**Counter**=**Counter**::**new();

    add\_two(**&***mut* cnt, number);

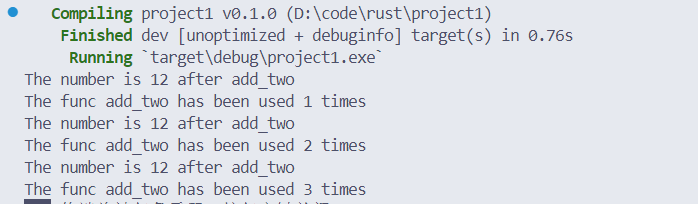
    add\_two(**&***mut* cnt, number);

    add\_two(**&***mut* cnt, number);

}

原理：每次调用add\_two函数，都会传入一个Counter结构体的引用，专门用来计数，一次达到记录使用多少次函数的功能。

运行结果：



17、代码：

**fn** parse\_config(contents**:** **&**str, key**:** **&**str) **->** Option<i32> {

**for** line in contents**.**split('\n') {

        let parts**:**Vec<**&**str>**=** line**.**splitn(2, '=')**.**collect();

**if** parts**.**len()**!=**2

            {**continue**;}

        let (current\_key,value) **=** (parts[0],parts[1]);

**if** current\_key **==** key {

**match** value**.**parse**::**<i32>() { *//尝试转化为i32*

                Ok(value) **=>** **return** Some(value), *// 解析成功则立即返回*

                Err(\_) **=>** **return** None, *// 解析失败则返回 None*

            }

        }

    }

    None *// 所有行都检查过后仍未找到匹配项，则返回 None*

}

**fn** main() {

*// 测试用例*

    let config **=** "max\_connections=100\ndefault\_timeout=60";

    assert\_eq!(parse\_config(config, "max\_connections"), Some(100)); *// 应返回 Some(100)*

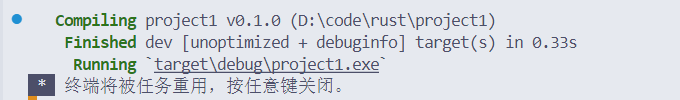
    assert\_eq!(parse\_config(config, "default\_timeout"), Some(60));

    assert\_eq!(parse\_config(config, "min\_connections"), None);

    assert\_eq!(parse\_config("invalid=abc", "invalid"), None);

    }

运行结果：



可以看到运行正常