

Харківський університет радіоелектроніки
Факультет комп'ютерних наук
Кафедра програмної інженерії

ЗВІТ

до практичної роботи з дисципліни «Аналіз та
рефакторинг коду»
на тему: «Основні рекомендації написання коду для обраної мови
програмування»

Виконав ст. гр ПЗПІ-22-2 Верясов
Владислав Олексійович

Перевірив
Доцент кафедри ПІ
Лещинський Володимир
Олександрович

Харків 2024

МЕТА

Метою даної роботи є ознайомлення з основними рекомендаціями та принципами написання коду на мові програмування Rust. Rust стрімко набирає популярність завдяки своїм унікальним можливостям забезпечення безпеки пам'яті та високої продуктивності. Основна задача — зрозуміти, як ключові особливості Rust, такі як система власності (ownership), управління позичанням (borrowing) та типи Option і Result, допомагають уникати типових помилок, що трапляються в інших мовах програмування.

Ця робота також націлена на вивчення підходів до організації багатопотокових додатків у Rust, де мова пропонує безпечне використання ресурсів і синхронізацію потоків, без необхідності складних механізмів для захисту даних, як це є в інших мовах. Крім того, важливим аспектом є ознайомлення з культурою та кращими практиками Rust, такими як використання системи тестування та інструментів форматування коду, що сприяють полегшенню розробки та підтримки програмного забезпечення.

ЗАВДАННЯ

Завданням практичної роботи є написання основних рекомендацій написання коду для мови програмування Rust.

ХІД РОБОТИ

Rust — це системна мова програмування, створена для забезпечення безпеки пам'яті та продуктивності, без використання сміттєзбирача. Вона поєднує високорівневі можливості з низькорівневим доступом до системних ресурсів, що робить її ідеальною для розробки високопродуктивних додатків.

Переваги Rust:

1. **Безпека пам'яті:** Використовує систему власності (ownership) для керування ресурсами, що запобігає помилкам типу "null pointer" або "use-after-free".
2. **Висока продуктивність:** Код компілюється в машинний код і працює так само швидко, як C або C++.
3. **Безпечна багатопотоковість:** Вбудовані механізми Rust запобігають виникненню умов гонки під час багатопотокової обробки.
4. **Екосистема та інструменти:** Мова має потужний менеджер пакетів (Cargo) та зручні інструменти для тестування й форматування коду.

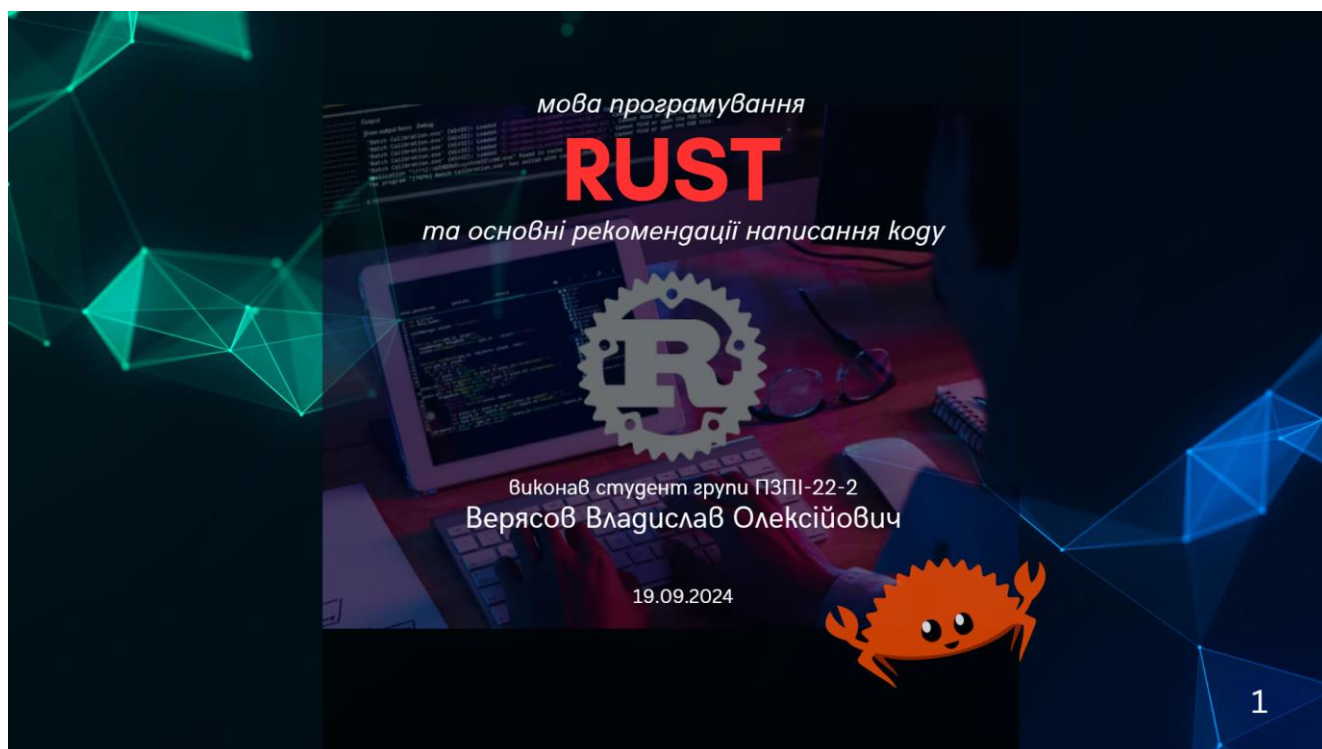
Це робить Rust одним із найкращих варіантів для розробки системного програмного забезпечення та проектів, де критично важливі продуктивність та безпека.

ВИСНОВКИ

У процесі виконання даної роботи було розглянуто низку важливих принципів та рекомендацій для написання якісного та ефективного коду на мові Rust.

У результаті, Rust є чудовим вибором для розробки додатків, де важливі надійність, безпека пам'яті та продуктивність. Використання рекомендацій, описаних у роботі, дозволяє створювати код, що не тільки легше підтримується, але й гарантує високий рівень якості та безпеки, що стає все більш важливим у сучасному світі розробки програмного забезпечення.

Додаток А: Презентація



ЩО TAKE RUST?

Rust — це системна мова програмування, створена для забезпечення безпеки пам'яті та продуктивності, без використання сміттєзбирача. Вона поєднує високорівневі можливості з низькорівневим доступом до системних ресурсів, що робить її ідеальною для розробки високопродуктивних додатків.

Чому варто обирати Rust:

- безпечність пам'яті
- швидкість
- безпечна багатопотоковість
- відсутність сміттєзбирача.


```
node.setup_interrupt_group(InterruptGroup::Rx1);
interrupt_group: InterruptGroup::Rx1;
Interrupt: Interrupt::Rx Fifo0 newMess
.unwr

fn init_can_stb_pin() {
    let gpio20 = pac::P20.split();
    let mut stb = gpio20.p20_6.into_push_pull();
    stb.set_low();
}

#[export_name = "main"]
fn main() -> ! {
```

2

ЯК ЖЕ ПРАВИЛЬНО ПИСАТИ КОД НА RUST?



```
fn divide(a: f64, b: f64) -> Option<f64> {
    if b == 0.0 {
        None // Уникаємо ділення на нуль
    } else {
        Some(a / b)
    }
}

fn main() {
    match divide(10.0, 2.0) {
        Some(result) => println!("Результат: {}", result),
        None => println!("Помилка: ділення на нуль"),
    }
}
```

Загальні принципи написання коду

- Писати чіткий і зрозумілий код.
- Уникати губликів коду (DRY – Don't Repeat Yourself).
- Використовувати Rust-форматування за допомогою інструменту rustfmt.

3

РЕКОМЕНДАЦІЇ ЩОДО ЗМІННИХ

- Використовуйте константи там, де це можливо: `const` і `static`.
- Намагайтеся використовувати іменовані змінні, що чітко описують їх призначення.
- Застосовуйте змінні лише з потрібною областю видимості.



4

УПРАВЛІННЯ ПАМ'ЯТТЮ

Приклад:

```
fn main() {  
    let s1 = String::from("hello");  
    let s2 = &s1; // Позичання  
    println!("{}", s2); // Виведення позиченого значення  
}
```

- Використовуйте власність (ownership) та позичання (borrowing) для ефективного керування пам'яттю.
- Використовуйте посилання (&) для зменшення копій даних.

5

ВИКОРИСТАННЯ "OPTION" ТА "RESULT"

Приклад:

```
fn divide(a: i32, b: i32) -> Result<i32, String> {  
    if b == 0 {  
        Err(String::from("Ділення на нуль"))  
    } else {  
        Ok(a / b)  
    }  
}
```

- Уникайте null-значень, використовуючи Option.
- Використовуйте Result для обробки помилок.

6

ЕРГОНОМІЧНЕ ВИКОРИСТАННЯ ФУНКЦІЙ ТА ЗАМИКАНЬ

- Використовуйте короткі функції з чіткими цілями.
- Замикання дозволяють спростувати код і знижувати дублювання.

Приклад замикання:

```
let add_one = |x: i32| x + 1;  
println!("{}", add_one(5));
```

ТЕСТУВАННЯ

- Використовуйте вбудовану систему тестування Rust.
- Пишіть юніт-тести для кожної функції.

Приклад юніт-тесту:

```
#[cfg(test)]  
mod tests {  
    #[test]  
    fn test_add() {  
        assert_eq!(2 + 2, 4);  
    }  
}
```



ВИСНОВКИ

- Дотримання рекомендацій Rust допомагає писати продуктивний, безпечний і легкий у підтримці код.
- Інструменти, як-от cargo і rustfmt, є невід'ємною частиною екосистеми Rust для підтримки кращих практик.

ДЯКУЮ ЗА УВАГУ!

