Системне Програмування

З використанням мови програмування **Rust.**
Fundamentals. Memory

Пам’ять

Rust відрізняється від мов, таких як Java, Python чи Go, тим, що не використовує автоматичне збирання сміття (GC). Замість цього він застосовує систему власності (ownership system), яка дозволяє:

Уникати пауз під час виконання програми, спричинених GC.
Досягати високої продуктивності, порівнянної з C/C++.
Мати кращий контроль над пам’яттю, що важливо для системного програмування.

Безпечність роботи з пам’яттю

Rust забезпечує **безпечне** керування пам’яттю без необхідності ручного виділення/звільнення ресурсів:

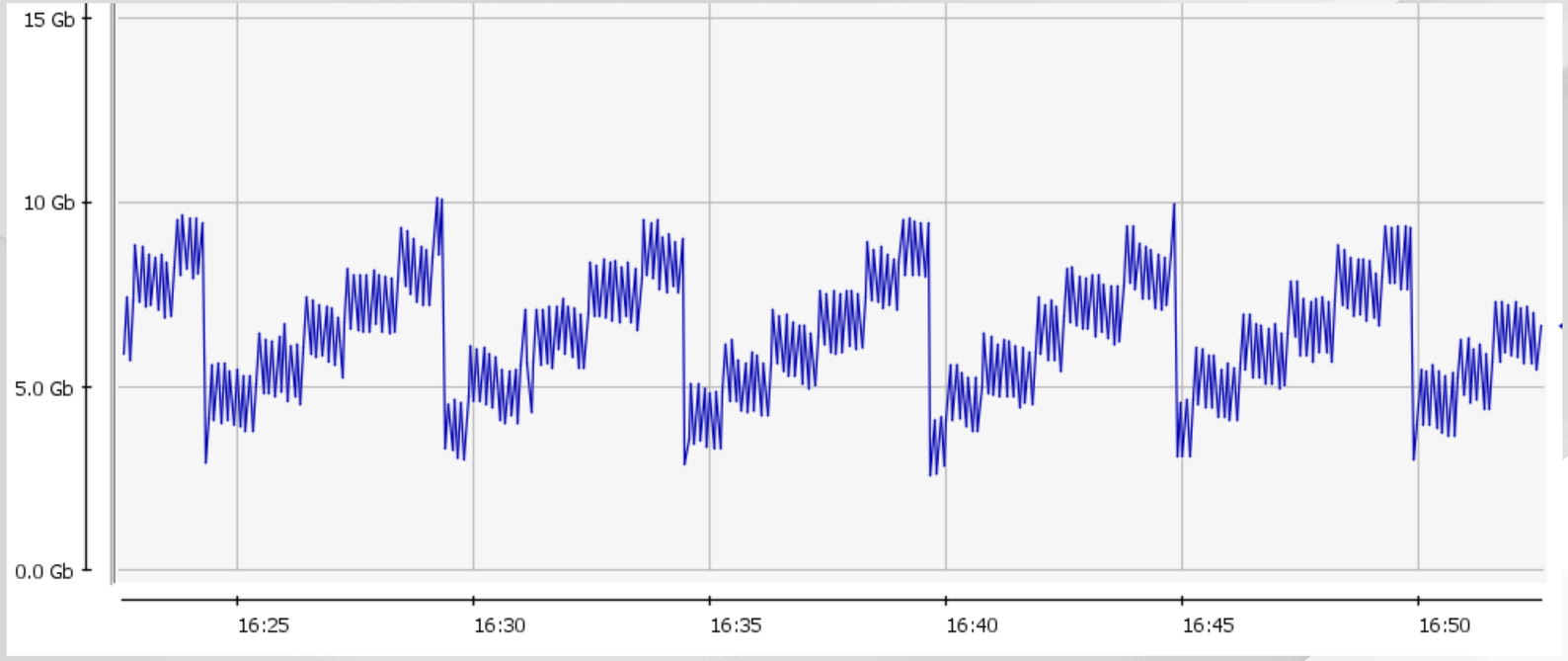
Ownership & Borrowing – система власності та запозичення гарантує, що немає витоків пам’яті або недійсних вказівників.

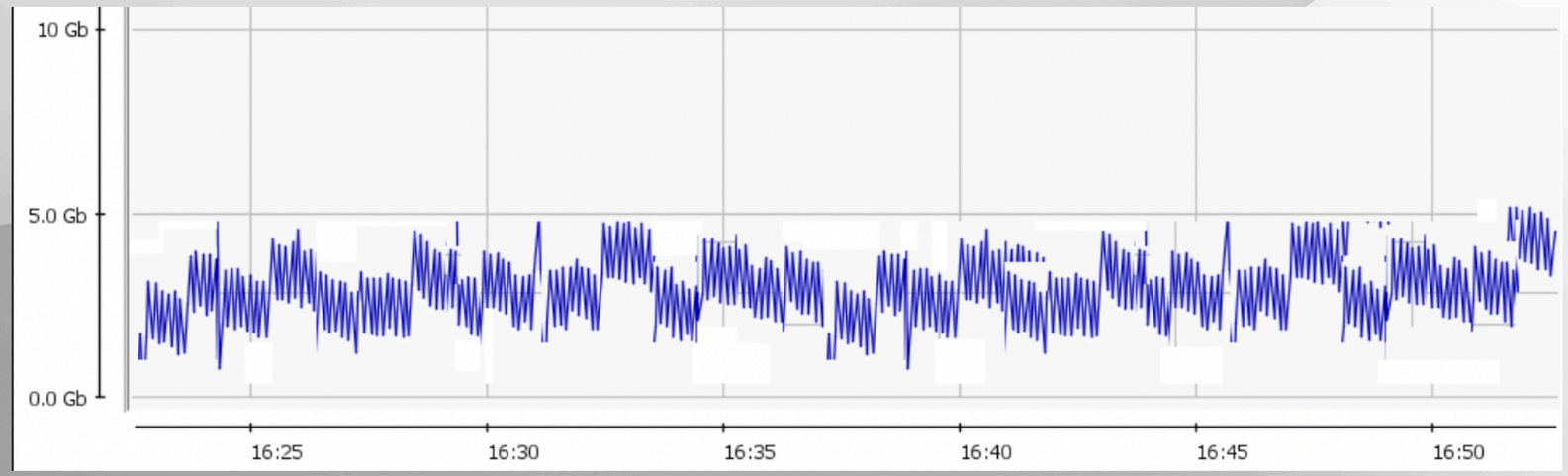
Mutable & Immutable Borrowing – запобігає змагальним умовам (race conditions) під час доступу до змінних у багатопотоковому середовищі.

Lifetime Annotations – допомагають компілятору визначати, як довго дані залишаються дійсними.

Ці механізми дозволяють Rust досягати рівня безпеки пам’яті, якого важко досягти в C/C++ без використання додаткових засобів аналізу.

Використання пам’яті: зі збирачем сміття та без





Пам’ять

Ми бачимо що для мов програмування зі збирачем сміття характерно:

- більше споживання пам’яті, і потім, кожний певний проміжок часу, “збірка сміття” (дані які вже не потрібні)

- існує момент часу (інколи досить великий, 0.5–10 секунд), Коли програма “нічого не робить”, коли вона має зібрати сміття та “почистити” пам’ять перед наступним використанням

Порівняння коду

Java, Scala, Python

```
case class Person(id: Int, name: String)

def whatever(p: Person): Unit = {
  // ...
  val p = new Person(33, "Jim")
  // ...
}
```

Пам’ять, виділена
Під змінну **p**
не звільняється одразу.
Вона звільняється,
коли пам’ять
закінчується

Rust

```
struct Person {
    id: u32,
    name: String,
}

fn whatever() {
    // ...
    let p = Person {
        id: 33,
        name: "Jim"
    };
    // ...
}
```

Пам’ять, виділена
Під змінну **p**
звільняється одразу.
Після закриваючої дужки **}**

Робота з пам’яттю в Rust

1. Декларація НЕЗМІННОЇ комірки пам’яті

```
let x :i32 = 5;
x = x + 1;
```

2. Декларація ЗМІННОЇ комірки пам’яті

```
let mut k :i32 = 5;
k = k + 1;
```

Але:

```
let x :i32 = 5;

let x :i32 = 6;

let x :&str = "trick";
```

Це називається variable shadowing
Використовується тоді, коли попереднє значення вже не потрібно
І зазвичай, ніколи не використовується для того самого типу.

Поганий приклад

```
let x :i32 = 5;
let x :i32 = 6;
```

Негативно впливає на семантику,
Бо треба тримати в голові чому 6 а не 5…
І взагалі, на наступній сторінці треба пам’ятати 6 або 5 і чому.

Гарний приклад

```
let x :&str = "5";
let x :u32 = x
    .parse::<u32>()
    .unwrap();
```

Ми конвертували **String** в **u32**,
і String нам тепер не потрібен, тому має сенс зробити shadow,
тобто залишити нове значення для змінної **x**
Якщо такого функціонала не було б, то використання змінних типу xStr, xInt погано впливало б на розуміння коду.

Гарний приклад 2

```
let spaces : &str  = "    ";
let spaces : usize = spaces.len();
```

Перша змінна `spaces` має тип рядка,
а друга змінна `spaces` — числовий тип.
Таким чином, затемнення (shadowing) позбавляє нас необхідності вигадувати різні імена, наприклад, `spaces_str` і `spaces_num`; замість цього ми можемо повторно використовувати простіше ім'я `spaces`.

Mutable shadow

```
let mut spaces : &str = "   ";
spaces = spaces.len();
```

Однак, якщо ми спробуємо використати `mut` для цього, ми отримаємо помилку під час компіляції.

Значення та посилання

Rust — це нізькорівнева мова програмування (як і мова C),
тому операціям з пам’яттю приділяється багато нюансів.

Rust розділяє значення та посилання, та має відповідний синтаксис
для робити з ними (синтаксис такий самий як у мові C)

```
let x :i32 = 5;

let px :&i32 = &x;

let x2 :i32 = *px;
```

Значення та посилання

Rust має розумний компілятор і в багатьох випадках може конвертувати автоматично

```
let x :i8 = 5;

let px :&i8 = &x;

let x_abs :i8 = px.abs();
```

```
fn go(px: &i32) -> i32 {
    px.abs()
}
```

Константи

Константи займають особливе місце в Rust
- їм потрібно вказати тип

```
const x = 5;
```

- їм неможливо зробити shadow

```
const x: u8 = 5;
let x: u8 = 6;
```

Хоча цей код виглядає “правильно” в IDE,
але компілятор видасть помилку

Scope

Як в більшості мов програмування, вкладений блок {...} декларує іншу область, яка може робити тимчасовий shadow, якщо бути не дуже уважним

```
let x :i32 = 5;
let x :i32 = x + 1; // shadow
{
    let x :i32 = x * 2; // inner scope
    println!("inner: {x}");
}
println!("outer: {x}");
```

Scope

```rust
let x :i32 = 5;           // x = 5
let x :i32 = x + 1;       // x = 6, shadow
{
    let x :i32 = x * 2; // x = 12, temporary shadow
    println!("inner: {x}");
}
println!("outer: {x}"); // x = 6
```

Ownership & Borrowing

Що таке Ownership

Володіння (**Ownership**) – це набір правил, які визначають, як програма на Rust керує пам’яттю. Усі програми повинні управляти використанням пам’яті під час виконання. Деякі мови програмування використовують збирання сміття (garbage collection), яке регулярно знаходить і очищає невикористовувану пам’ять під час роботи програми. В інших мовах програміст має явно виділяти та звільняти пам’ять.

Rust застосовує **третій підхід**: пам’ять керується через систему володіння, яка підкоряється певному набору правил, що перевіряються компілятором. Якщо будь-яке з цих правил буде порушене, програма не скомпілюється.

Важливо, що механізм володіння не уповільнює виконання програми, оскільки всі перевірки відбуваються під час компіляції.

Правила Ownership

- Кожне значення в Rust має власника.
- Одночасно може бути лише один власник.
- Коли власник виходить за межі області видимості, значення буде знищене (звільнене з пам’яті).

Scope

```
{                                   // s is not valid here
    let s : &str = "hello";         // s is valid
    // ...
}                                   // this scope is now over,
                                    // and s is no longer valid
```

Пам’ять звільняється відразу як змінна вийшла з області пам’яті

Move

Rust має цікаву концепцію **Move**:

```
let x :i32 = 5;
let y :i32 = x;
println!("{}", y); // 5
println!("{}", x); // 5
```

Виглядає зрозуміло і не відрізняється від інших мов програмування

Move

але…

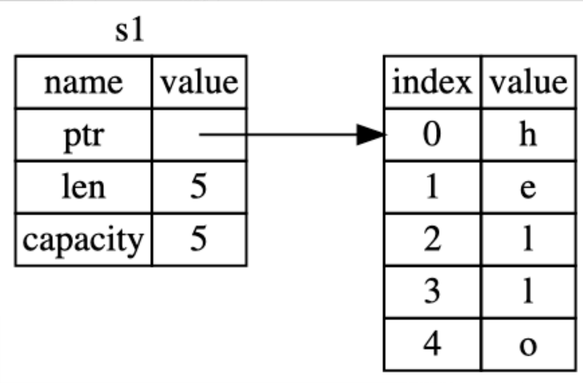
```
let s1 : String = String::from( s: "Rust!");
let s2 : String = s1;
println!("{}", s2);
println!("{}", s1);
```

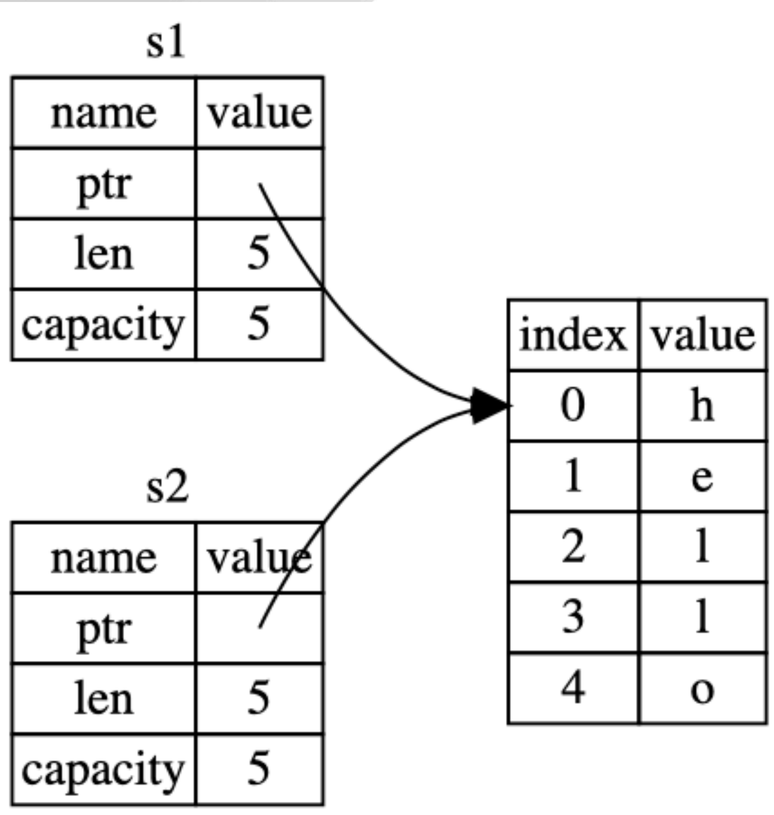
Не скомпілюється…

Move

```
let s1 : String = String::from( s: "hello");
let s2 : String = s1;
println!("{}", s2);
println!("{}", s1);
```

Не скомпілюється



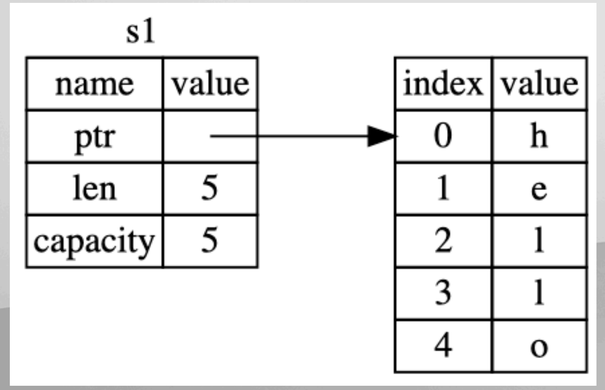


бо **Одночасно може бути лише один власник**

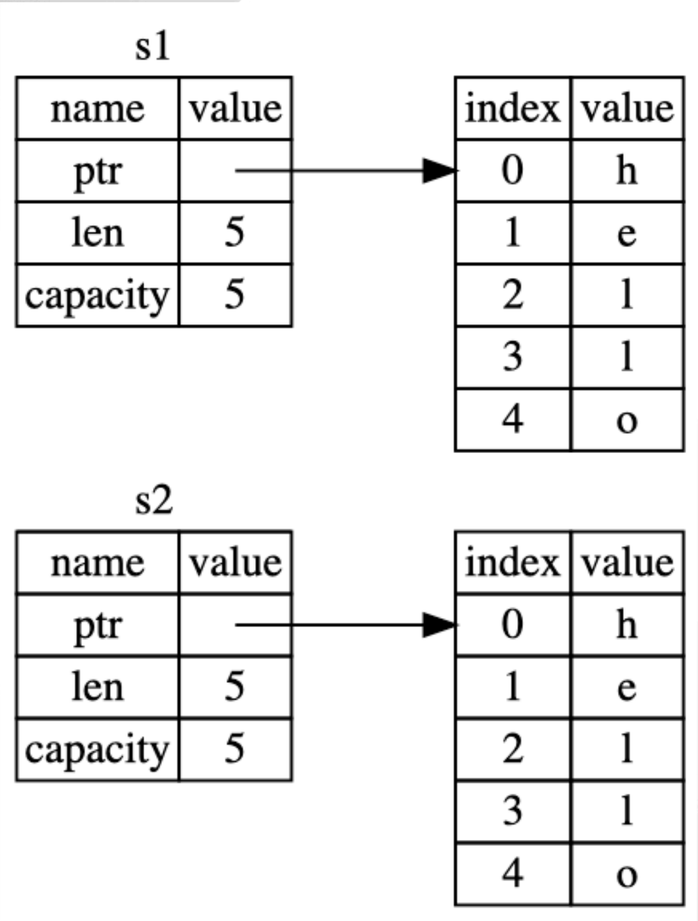
Move

Але інколи потрібно скопіювати

```
let s1 : String = String::from( s: "hello");
let s2 : String = s1.clone();
println!("{}", s2);
println!("{}", s1);
```



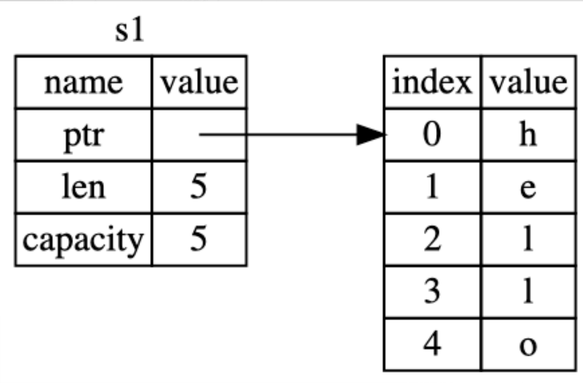
Rust надає 100% контроль

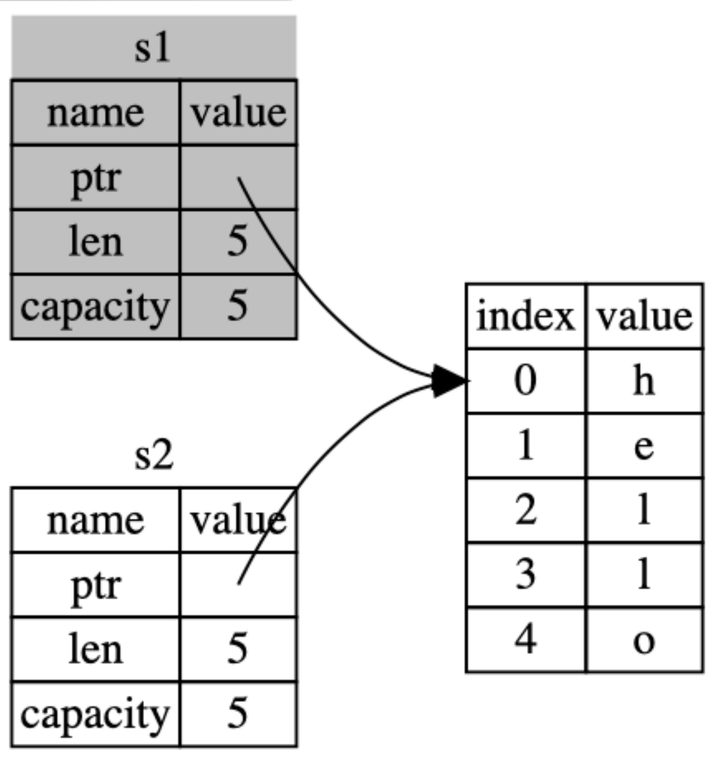


Move

```
let s1 : String = String::from( s: "hello");
let s2 : String = s1;
println!("{}", s2);
println!("{}", s1);
```

Не скомпілюється





бо **Одночасно може бути лише один власник**

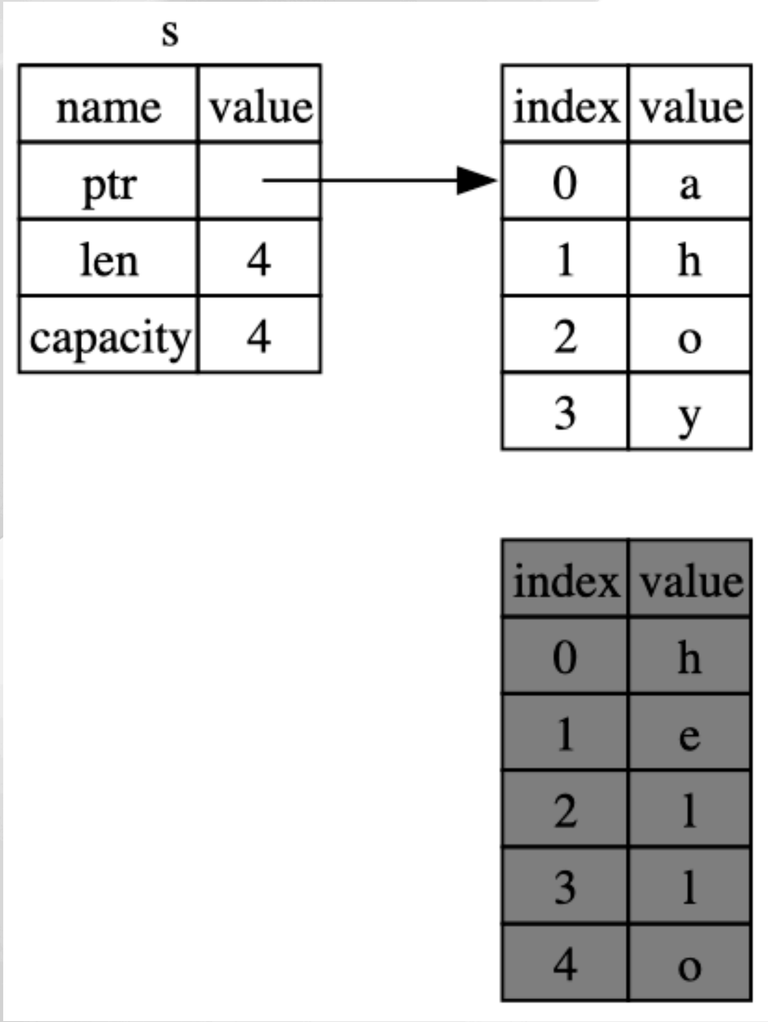
Моментальне звільнення

```rust
let mut s : String = String::from( s: "hello");
s = String::from( s: "ahoy");
println!("{s}, world!");
```

Навіть більше!
коли ми запускаємо компілятор
в режимі оптимізації:

```
~ cargo build --release
```

то навіть не буде аллокації
зі значенням “hello” тому що
не було жодного використання!



Пам’ять

Також поведінка компілятора частково може бути змінена, якщо ми для наших типів будемо імплементувати трейти:

- Copy
- Clone
- Drop
- Deref

Володіння (Ownership)

```rust
fn main() {
    let s : String =
        String::from( s: "hello"); // s comes into scope
    takes_ownership(s);           // s's value moves into the function...
    println!("{}", s);            // ... and so is no longer valid here

    let x : i32 = 5; // x comes into scope
    makes_copy(x); // because i32 implements the Copy trait,
                   // x does NOT move into the function,

    println!("{}", x); // so it's okay to use x afterward
} // Here, x goes out of scope, then s.
  // But because s's value was moved, nothing special happens.

fn takes_ownership(s1: String) { // s1 comes into scope
    println!("{s1}");
} // Here, some_string goes out of scope and `drop` is called.
  // The backing memory is freed.

fn makes_copy(i1: i32) { // i1 comes into scope
    println!("{i1}");
} // Here, i1 goes out of scope. Nothing special happens.
```

Володіння (Ownership)

Володіння можна передавати

```rust
fn gives_ownership() -> String {
    let s : String = String::from( s: "yours");
    // s comes into scope
    s // s is returned and moves out to the calling function
}

fn takes_and_gives_back(s: String) -> String {
    // s comes into scope
    s // s is returned and moves out to the calling function
}
```

References and Borrowing

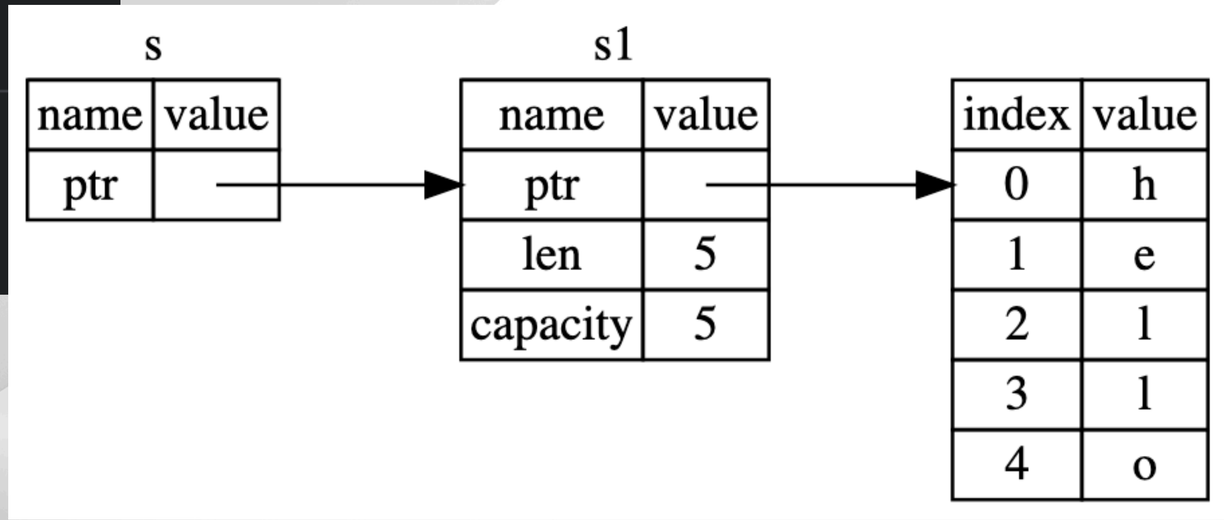
References and Borrowing

```rust
let x :i8 = 5;

let px :&i8 = &x;
```

Виглядає стандартно (як у мові C), але в Rust є “гарантія”, що значення за цим посилання ніхто не змінить.

References and Borrowing

```rust
fn main1() {
    let s1 : String = String::from( s: "hello");
    let len : usize = calc_length(&s1);
    println!("The length of '{s1}' is {len}.");
}

fn calc_length(s: &String) -> usize {
    s.len()
}
```



Синтаксис **&s1**
дозволяє нам створити посилання,
яке вказує на значення **s1**,
але не володіє ним. **Оскільки посилання не є власником**,
значення, на яке воно вказує, не буде видалене після того,
як посилання перестане використовуватися.

References and Borrowing

```rust
fn main() {
    let s : String = String::from( s: "hello");
    change(&s);
}

fn change(s: &String) {
    s.push_str( string: ", world");
}
```

Але цей код не скомпілюється

References and Borrowing

```rust
fn main() {
    let mut s : String = String::from( s: "hello");
    change2(&mut s);
}

fn change2(some_string: &mut String) {
    some_string.push_str( string: ", world");
}
```

Потрібно явно (`mut`) указати, що ми збираємося змінювати посилання

Але є нюанс

```
let mut s : String = String::from( s: "hello");

let r1 : &mut String = &mut s;
let r2 : &mut String = &mut s;

println!("{}, {}", r1, r2);
```

Не скомпілюється, бо в один момент не може існувати більше ніж одне посилання, яке дозволяє зміну

References and Borrowing

Зверніть увагу, що це абсолютно інший приклад

```
let mut s : String = String::from( s: "hello");

{
    let r1 : &mut String = &mut s;
} // r1 goes out of scope here,
  // so we can make a new reference with no problems.

let r2 : &mut String = &mut s;
```

References and Borrowing

Також не можливо існування mutable & immutable посилань

```
let mut s : String = String::from( s: "hello");

let r1 : &String = &s;          // no problem
let r2 : &String = &s;          // no problem
let r3 : &mut String = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

Не скомпілюється

Посилання в “нікуди”

```
fn main() {
    let nothing : &String = dangle();
}

fn dangle() -> &String {
    let s : String = String::from( s: "hello");
    &s
}
```

Не скомпілюється, оскільки нема “володаря”

Посилання в “нікуди”

```rust
fn main() {
    let owns : String = makes();
}

fn makes() -> String {
    let s : String = String::from( s: "hello");
    s
}
```

Якщо ми “створюємо” дані, то завжди повертаємо дані (не ref).
Це валідний код

Висновки

Rust – це потужна мова, яка дозволяє писати швидкі, безпечні та гнучкі програми без накладних витрат на збирання сміття.

Завдяки строгій системі власності, перевірці часу життя змінних та парадигмам функціонального програмування, Rust відкриває нові можливості для розробників у багатьох сферах –
- від низькорівневих системних програм
- до високопродуктивних веб-додатків.

Більше про роботу з пам’яттю на Rust:

https://doc.rust-lang.org/book/ch03-01-variables-and-mutability.html
https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html
https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Код з лекцій,
презентації Keynote,
PDF-файли
знаходяться на GitHub:

https://github.com/djnzx/rust-course
git@github.com:djnzx/rust-course.git