

# Rust는 어떻게 안전한 프로그래밍을 이뤄내는가

옥찬호 (Chris Ohk)

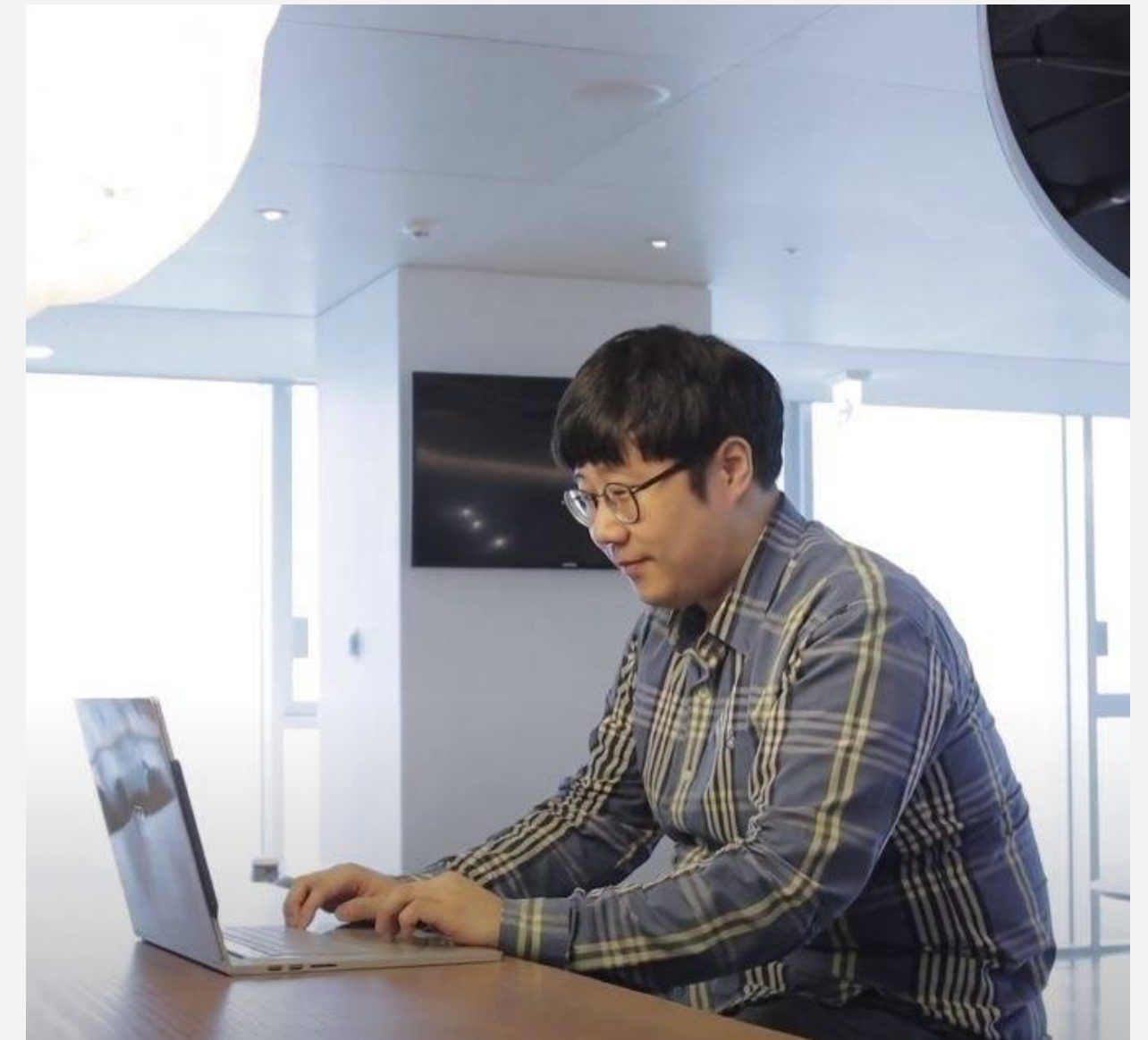
[utilforever@gmail.com](mailto:utilforever@gmail.com)

# 발표자 소개

- 옥찬호 (Chris Ohk)
  - (현) 42dot Embedded Software Engineer
  - (전) EJN Tech Lead
  - (전) Momenti Engine Engineer
  - (전) Nexon Korea Game Programmer
  - Microsoft Developer Technologies MVP
  - C++ Korea Founder & Administrator
  - Reinforcement Learning KR Administrator
  - IT 전문서 집필 및 번역 다수
  - 대학교 Rust 특강 및 강의 다수

utilForever@gmail.com

 utilForever



# 들어가며

- 프로그래밍을 해보지 않은 분에게는 어려운 주제일 수 있습니다.
- 오늘 발표들 중 코드(Rust 92% + C++ 7% + Python 1%)가 가장 많이 나옵니다.
- 메모리를 직접 관리하는 언어를 사용했던 경험이 있다면, 이해하는 데 도움이 됩니다.
- 발표 자료와 예제 코드는 다음 저장소에 올릴 예정입니다.

<https://github.com/utilForever/2024-DEVCON-Rust-Safety>

## Table of Contents

---

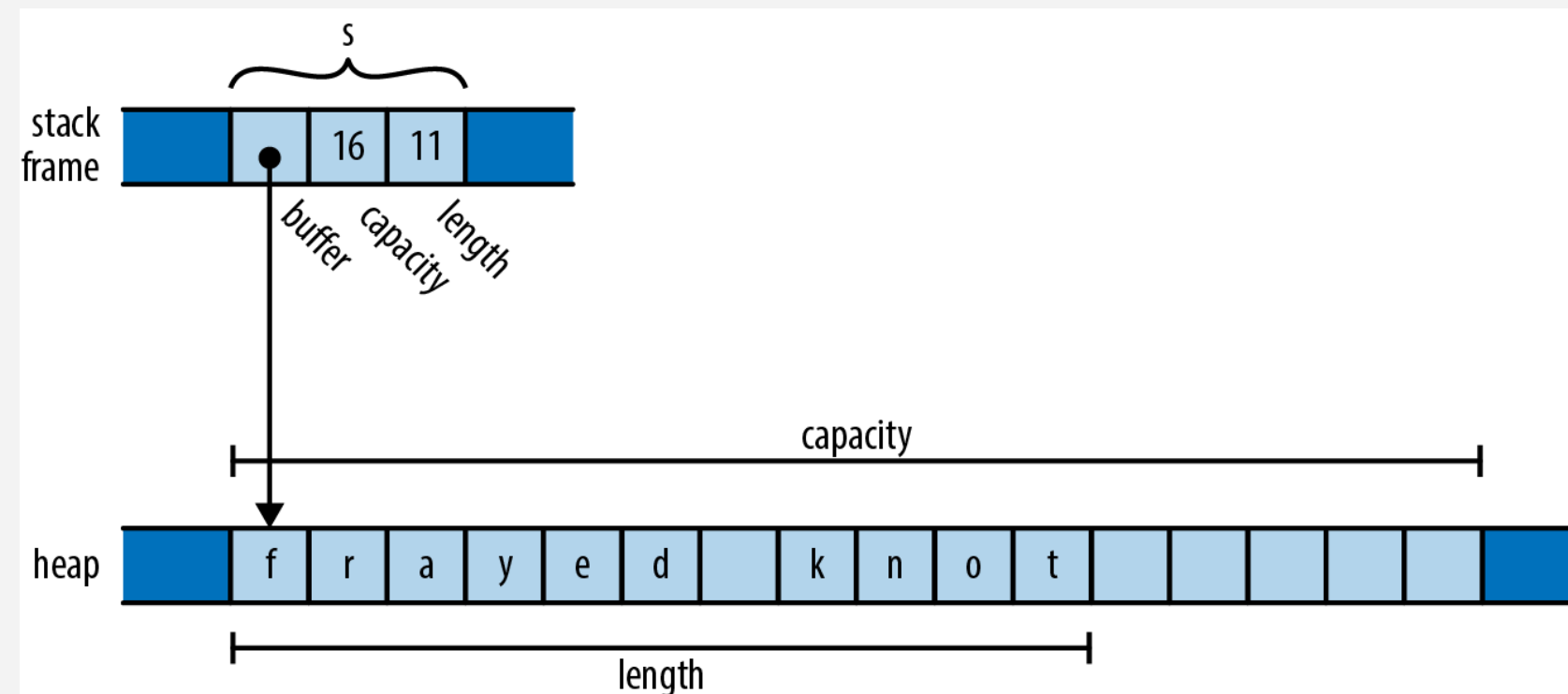
- 01** Rust의 메모리 관리 방식
- 02** Rust가 실수를 막는 방법
- 03** Rust의 메모리 관리 활용

# Rust의 메모리 관리 방식

소유권(Ownership), 이동(Moves), 레퍼런스(References)와 빌림(Borrowing), 수명(Lifetimes)

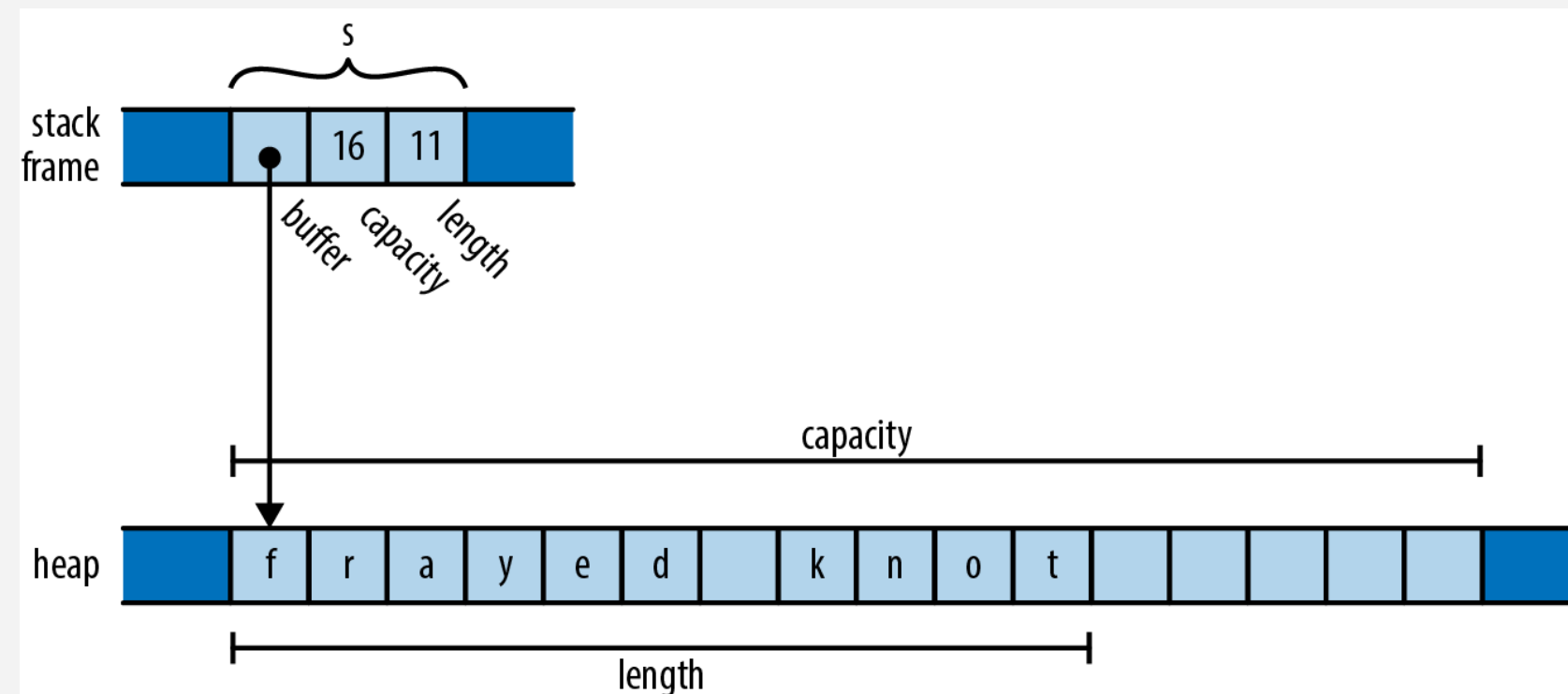
# 소유권(Ownership)

- C나 C++ 코드를 많이 봤다면 어떤 클래스의 인스턴스가 자신이 가리키는 다른 어떤 오브젝트를 소유한다고 표현하는 걸 많이 봤을 것이다.
- 소유주가 자신이 소유한 오브젝트를 소멸시키기 전에 그 코드에서 해당 포인터를 책임지고 없애야 한다. 소유자가 소유물의 수명을 결정하면 나머지는 그 결정을 존중해야 한다.



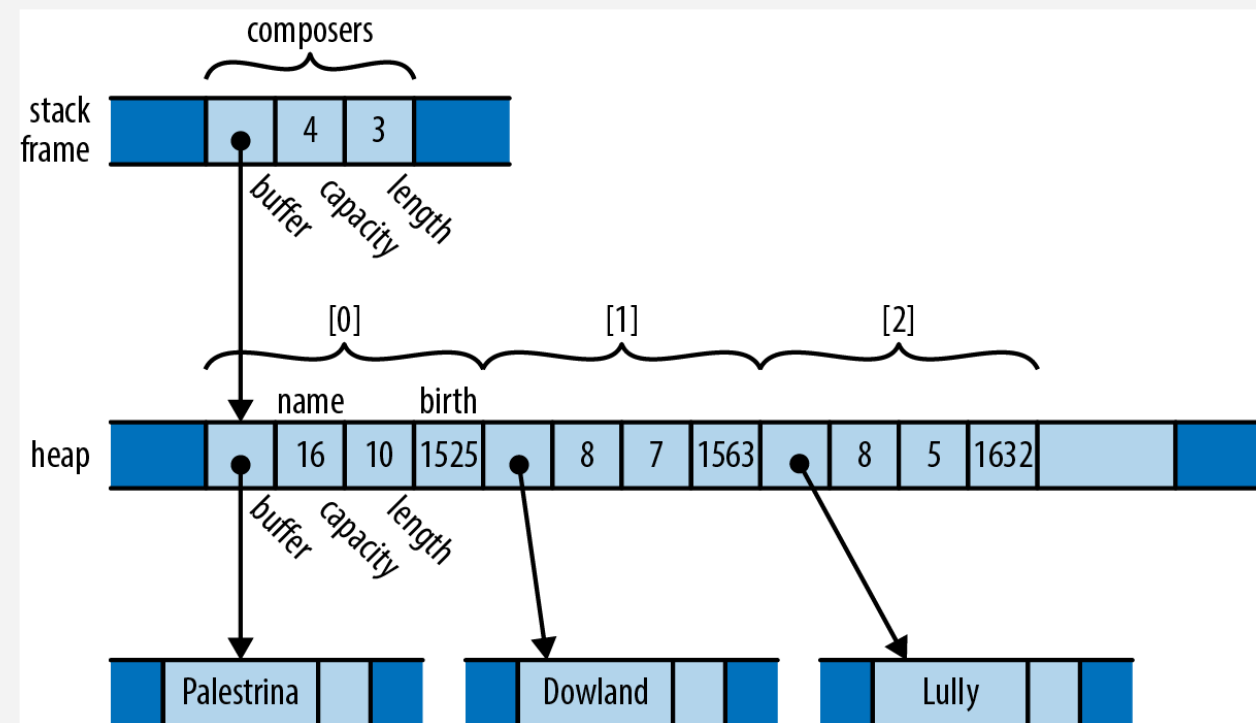
# 소유권(Ownership)

- Rust는 소유의 개념이 언어 자체에 내장되어 있으며 컴파일 타임 검사를 통해 검증된다.
  - 모든 값은 자신의 수명을 결정하는 소유자가 하나 뿐이다.
  - 소유자가 해제(Rust에서는 드롭(Drop)이라는 용어를 쓴다)될 때 그가 소유한 값도 함께 드롭된다.



# 소유권(Ownership)

- Rust는 소유의 개념이 언어 자체에 내장되어 있으며 컴파일 타임 검사를 통해 검증된다.
  - 변수가 자신의 값을 소유하듯이 구조체는 자신의 필드들을 소유한다.  
마찬가지로 튜플, 배열, 벡터 역시 자신의 요소들을 소유한다.
  - 소유자와 이들이 소유한 값은 트리(Tree) 구조를 이룬다.



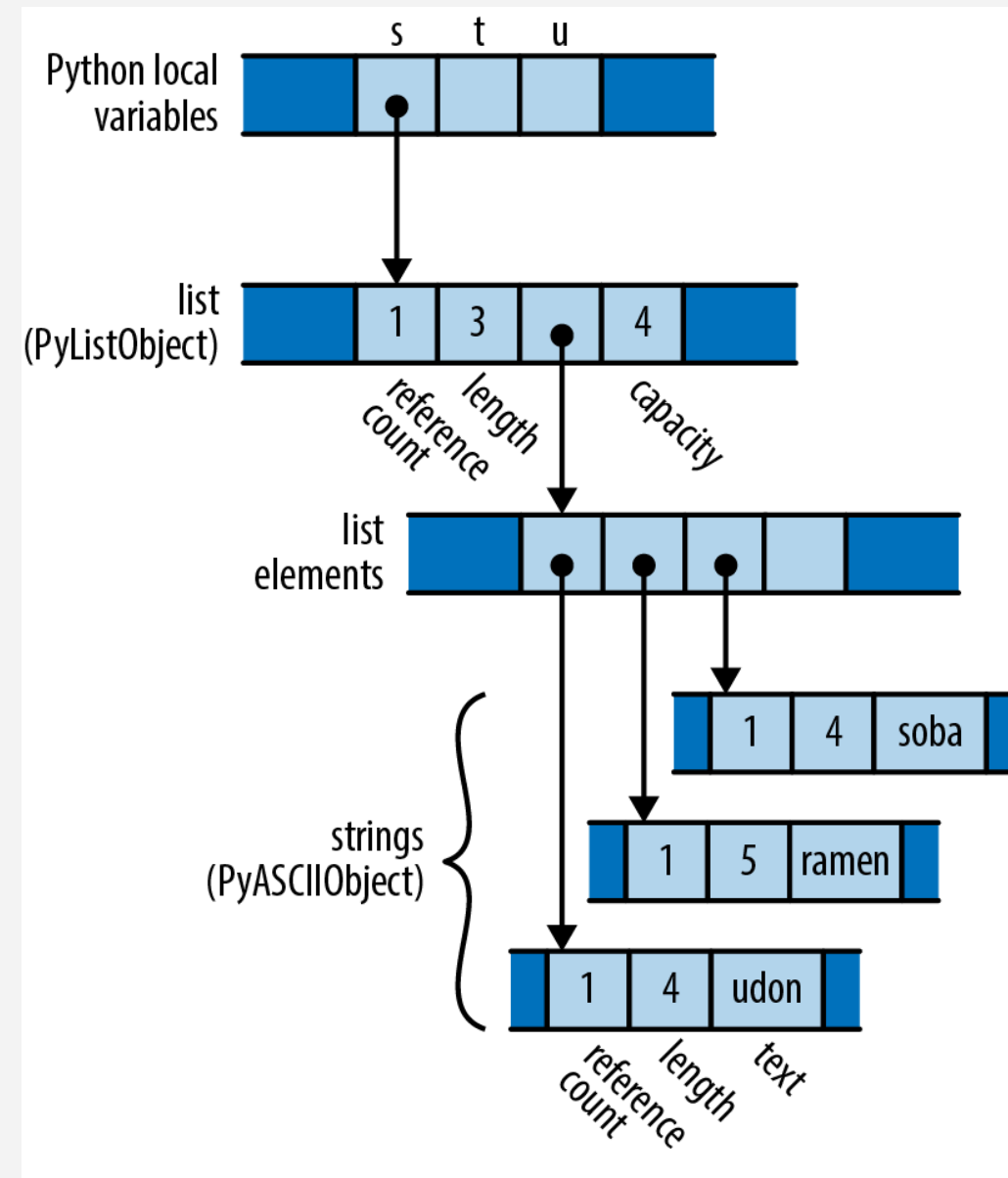


# 이동(Moves)

- Rust에서는 값을 변수에 대입하거나, 함수에 전달하거나, 함수에서 반환하거나 하는 식의 연산이 일어날 때 대부분 그 값이 복사되지 않고 이동(Move)된다.
  - 이때 원래 주인은 값의 소유권을 새 주인에게 양도하고 미초기화 상태가 되며, 이후 값의 수명은 새 주인이 통제한다.
  - Rust 프로그램은 값을 하나씩 쌓아서 복잡한 구조를 만들기도 하고 또 하나씩 옮겨서 이를 허물기도 한다.

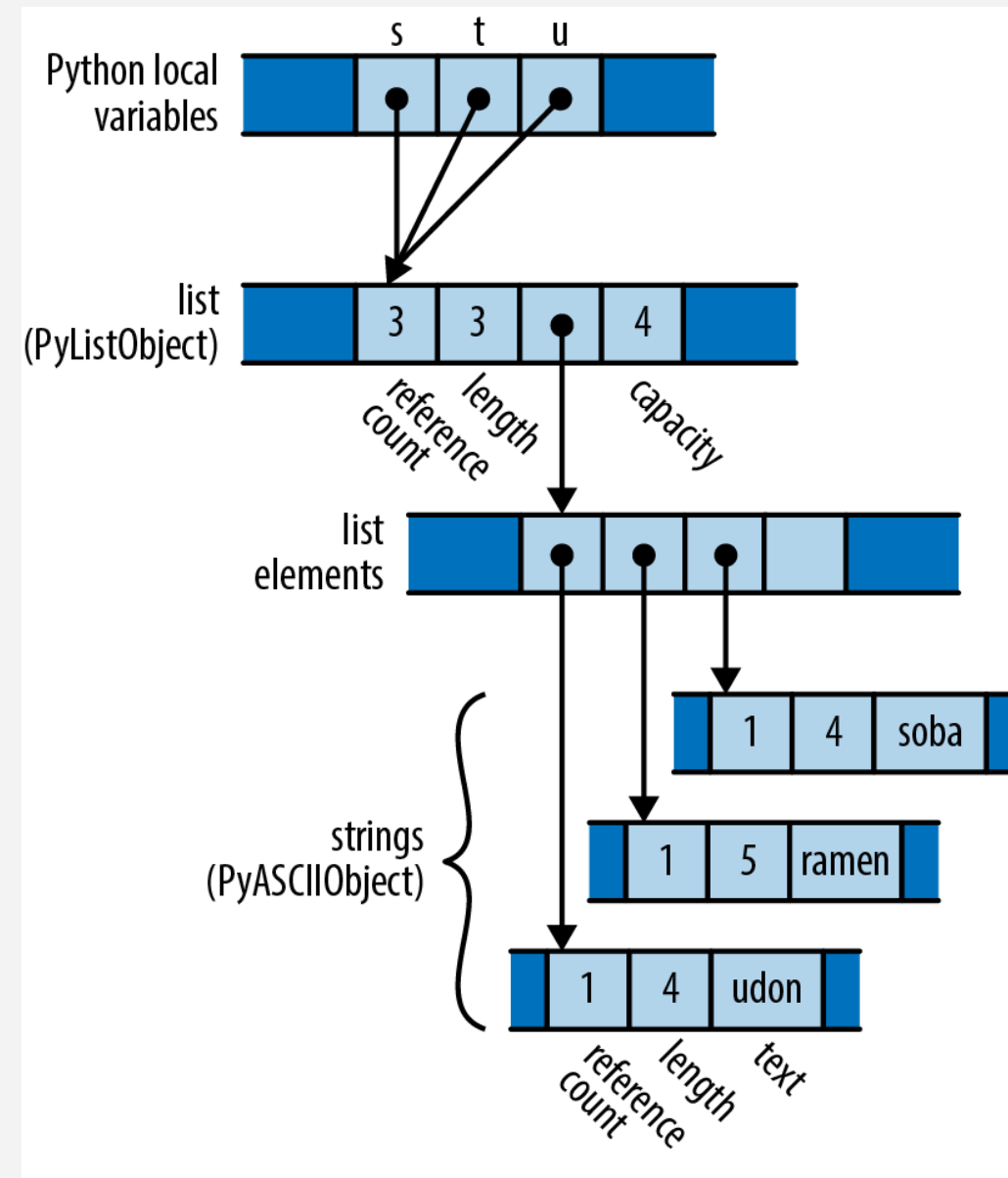
# 이동(Moves)

- 언어별 대입 처리 방식 비교 : Python



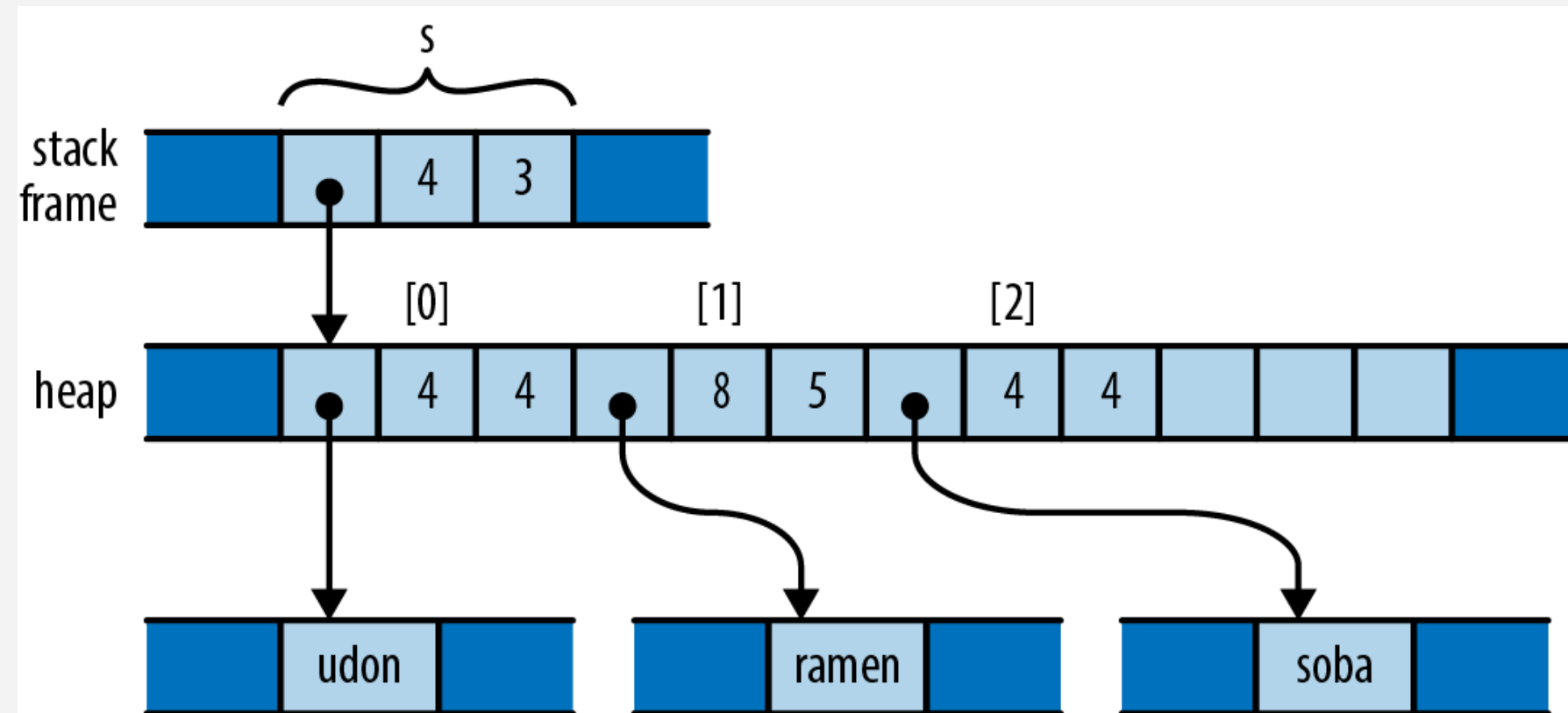
# 이동(Moves)

- 언어별 대입 처리 방식 비교 : Python



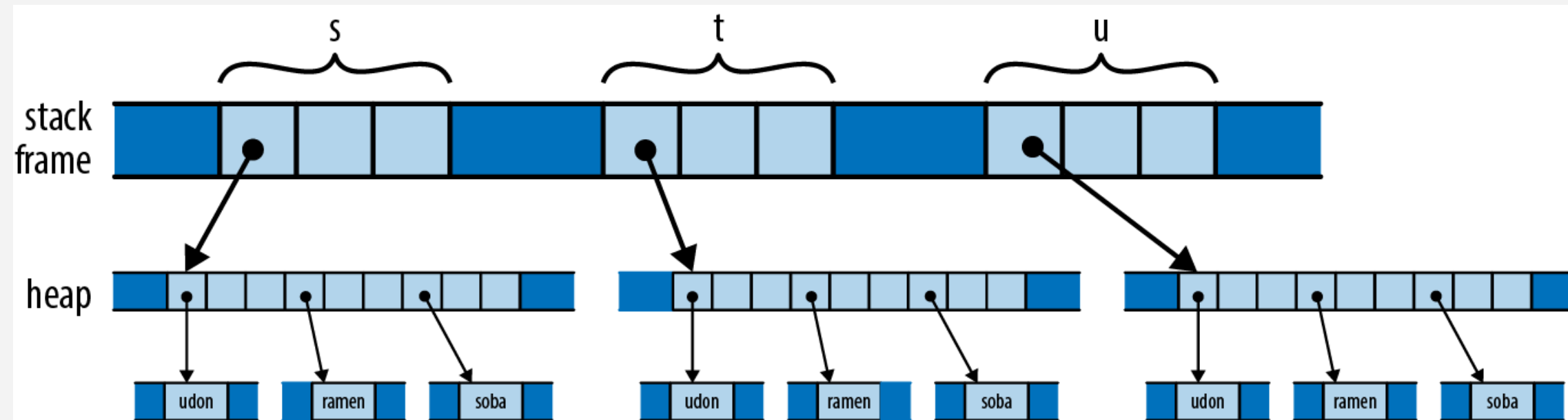
# 이동(Moves)

- 언어별 대입 처리 방식 비교 : C++



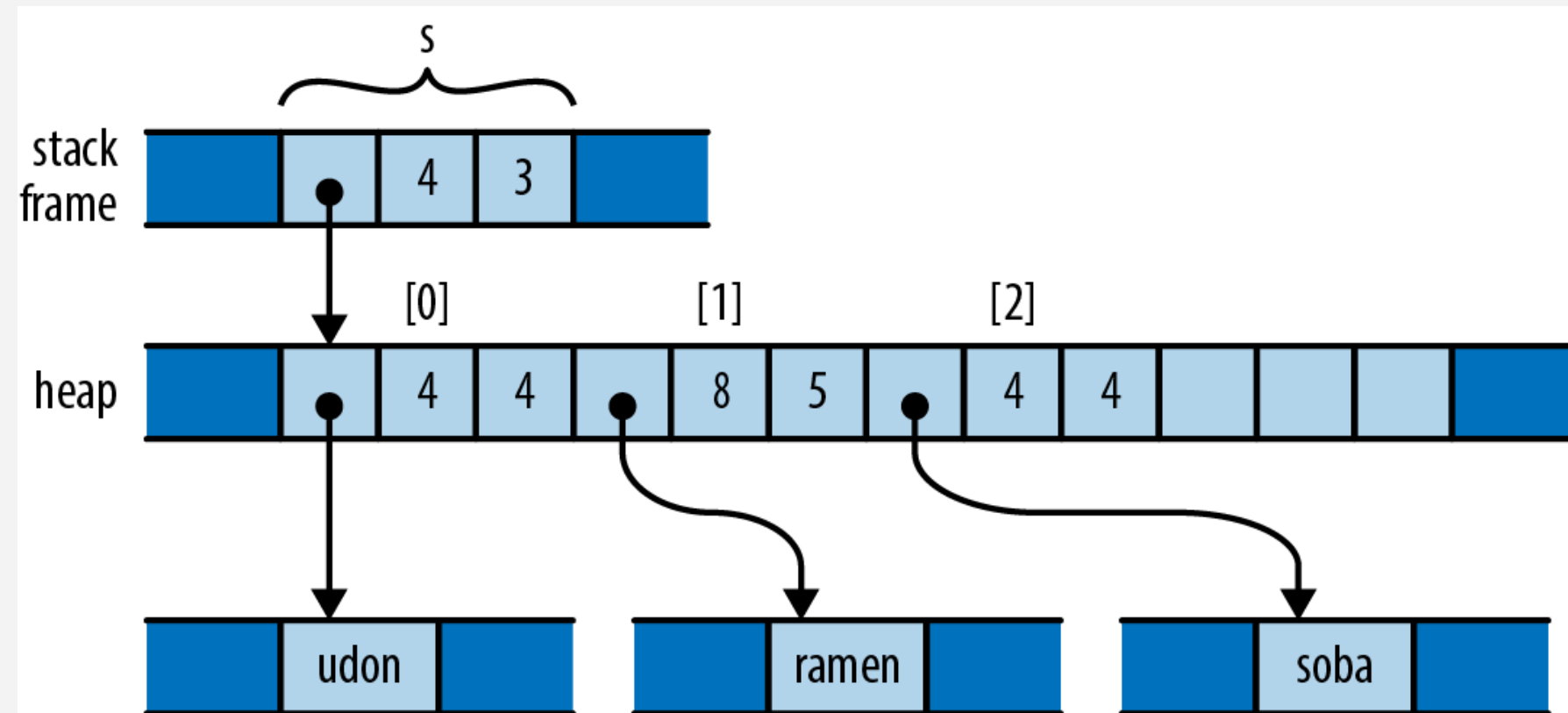
# 이동(Moves)

- 언어별 대입 처리 방식 비교 : C++



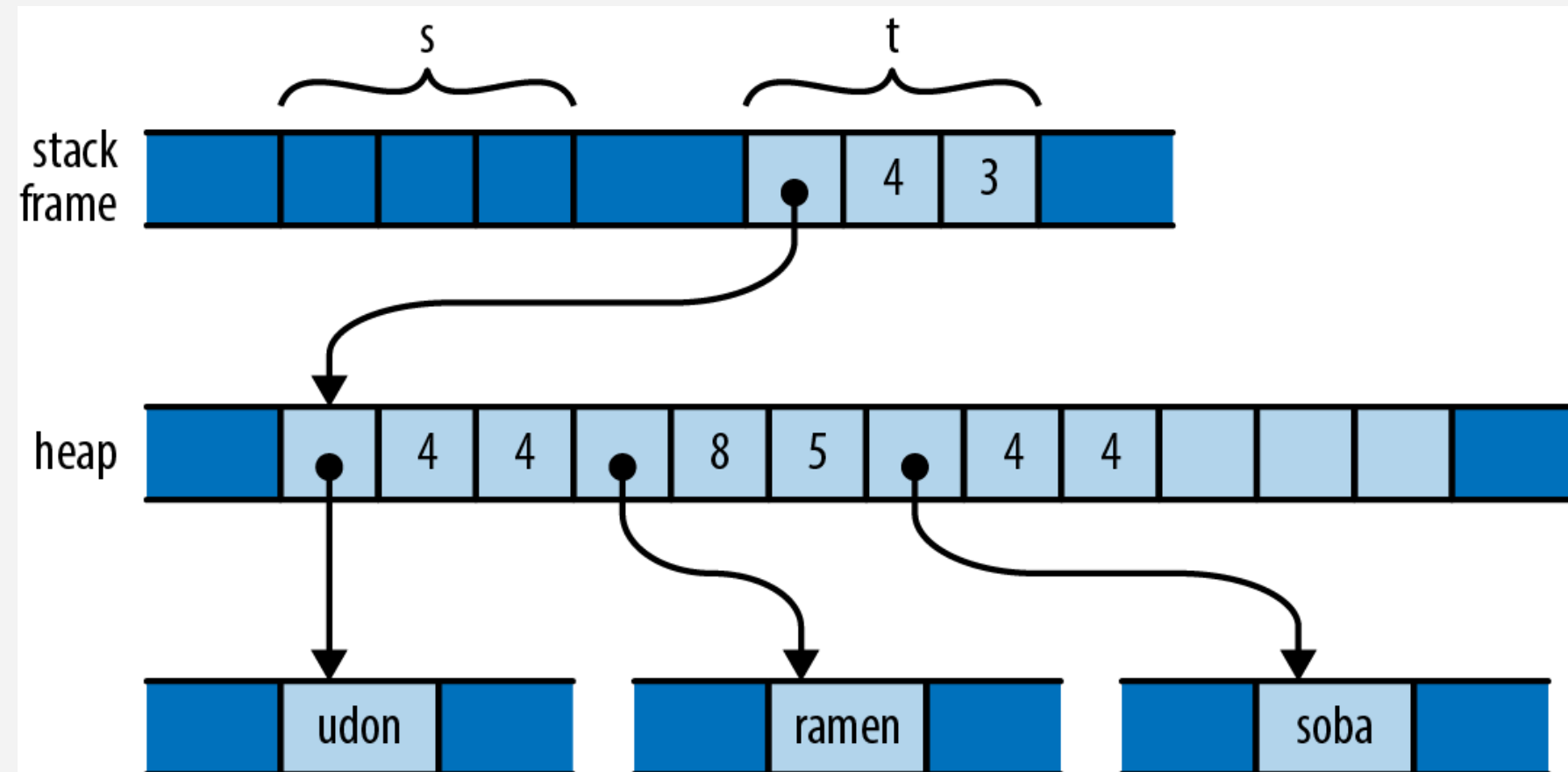
# 이동(Moves)

- 언어별 대입 처리 방식 비교 : Rust



# 이동(Moves)

- 언어별 대입 처리 방식 비교 : Rust

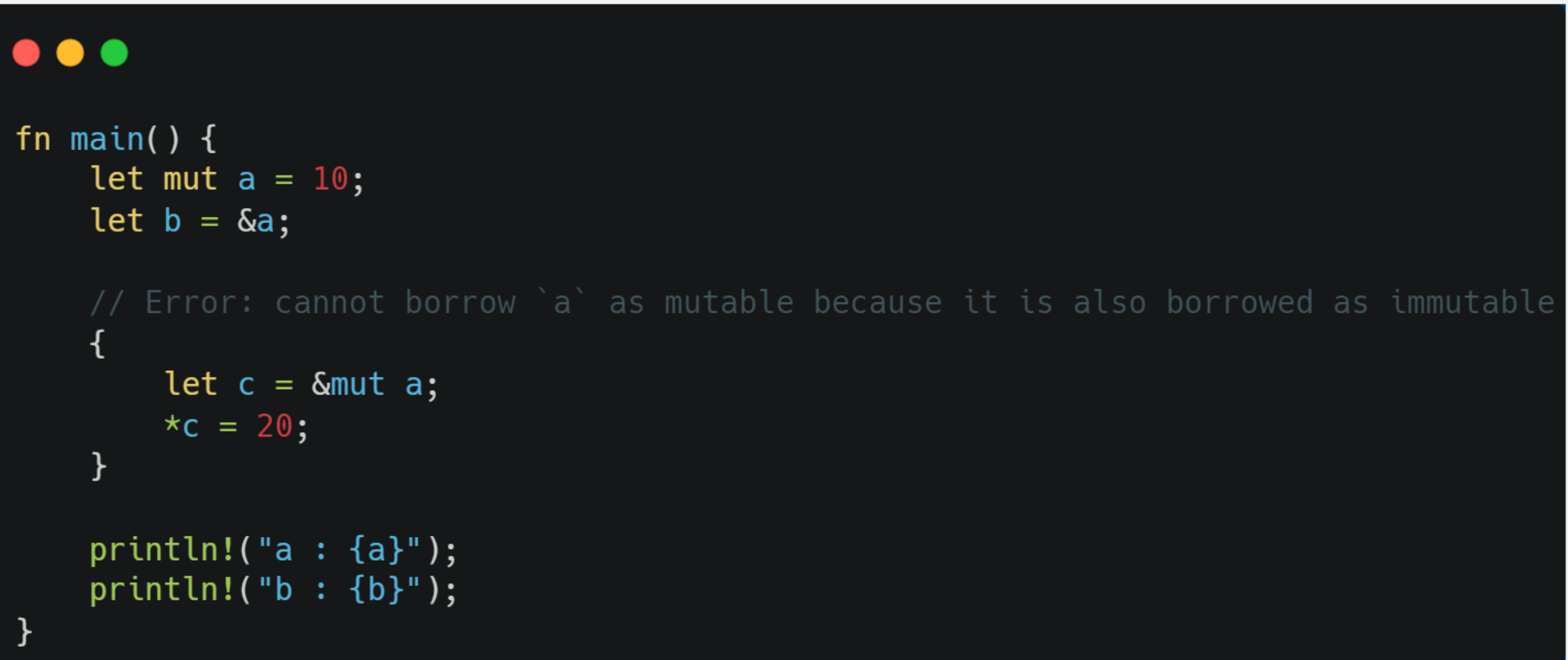


# 레퍼런스(Reference)

- 소유권이 매번 이동되는 게 번거롭다면, 레퍼런스를 통해 잠시 소유권을 빌려줄 수 있다.
- Rust는 이때 레퍼런스 규칙을 적용한다.
  - 단 하나의 변경 가능한 레퍼런스 또는 여러개의 변경 불가능한 레퍼런스만 허용하며, 둘 중 하나만 가능하다.
  - 레퍼런스는 그 소유자보다 더 오래 살 수 없다.



# 레퍼런스(Reference)



```
fn main() {  
    let mut a = 10;  
    let b = &a;  
  
    // Error: cannot borrow `a` as mutable because it is also borrowed as immutable  
    {  
        let c = &mut a;  
        *c = 20;  
    }  
  
    println!("a : {a}");  
    println!("b : {b}");  
}
```

# 수명(Lifetime)

- 레퍼런스는 C / C++에 있는 포인터와 비슷하다. 하지만 포인터는 안전하지 않다.  
Rust는 과연 어떤 식으로 레퍼런스를 통제하고 있는 걸까?

# 수명(Lifetime)

- 지역 변수 빌려오기
  - 지역 변수의 레퍼런스를 빌려올 때는 레퍼런스를 그 변수의 범위 밖으로 가지고 나갈 수 없다.

```
fn main() {  
    {  
        let r;  
  
        {  
            let x = 1;  
            r = &x;           // `x` does not live long enough  
        }  
  
        assert_eq!(*r, 1);    // Bad: Reads memory `x` used to occupy  
    }  
}
```

# 수명(Lifetime)

- 지역 변수 빌려오기
  - 지역 변수의 레퍼런스를 빌려올 때는 레퍼런스를 그 변수의 범위 밖으로 가지고 나갈 수 없다.

```
error: `x` does not live long enough
--> references_dangling.rs:8:5
   7 |         r = &x;
      |           ^^ borrowed value does not live long enough
   8 |     }
      |     - `x` dropped here while still borrowed
   9 |     assert_eq!(*r, 1); // bad: reads memory `x` used to occupy
  10 | }
```

# 수명(Lifetime)

- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 수명(Lifetime)을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
  - 수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다. 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 포함하거나 에워싸야 한다.

```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```

lifetime of &x must not exceed this range

# 수명(Lifetime)

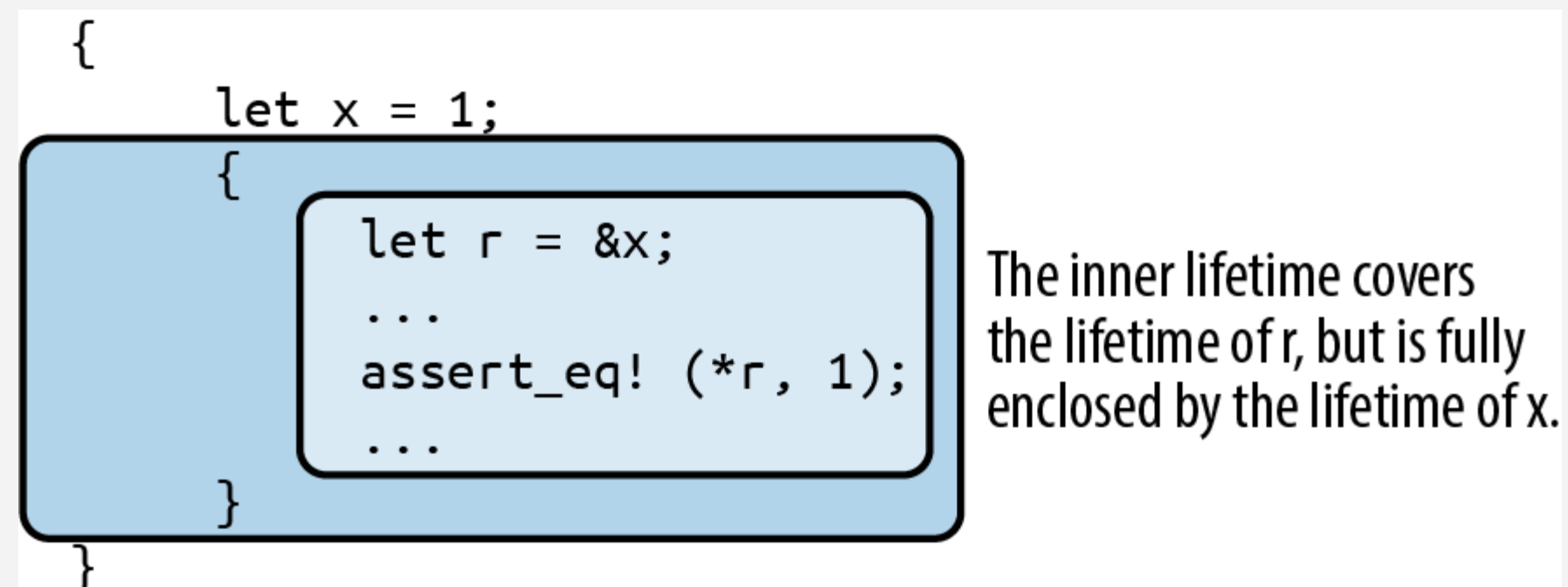
- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 수명(Lifetime)을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
  - 수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다. 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 포함하거나 에워싸야 한다.

```
{
    let r;
    {
        let x = 1;
        ...
        r = &x;
        ...
    }
    assert_eq!(*r, 1);
}
```

lifetime of anything stored in  
r must cover at least this range

# 수명(Lifetime)

- 지역 변수 빌려오기
  - Rust는 프로그램에 있는 모든 레퍼런스 타입을 대상으로 각 타입의 쓰임새에 맞는 제약 조건이 반영된 수명(Lifetime)을 부여하려고 한다. (컴파일 시점에만 존재하는 가상의 개념이다.)
  - 수명이란 실행문, 표현식, 변수 범위 등 프로그램에서 레퍼런스가 안전하게 쓰일 수 있는 구간을 말한다. 변수의 수명은 자신에게서 차용된 레퍼런스의 수명을 반드시 **포함**하거나 **에워싸야** 한다.



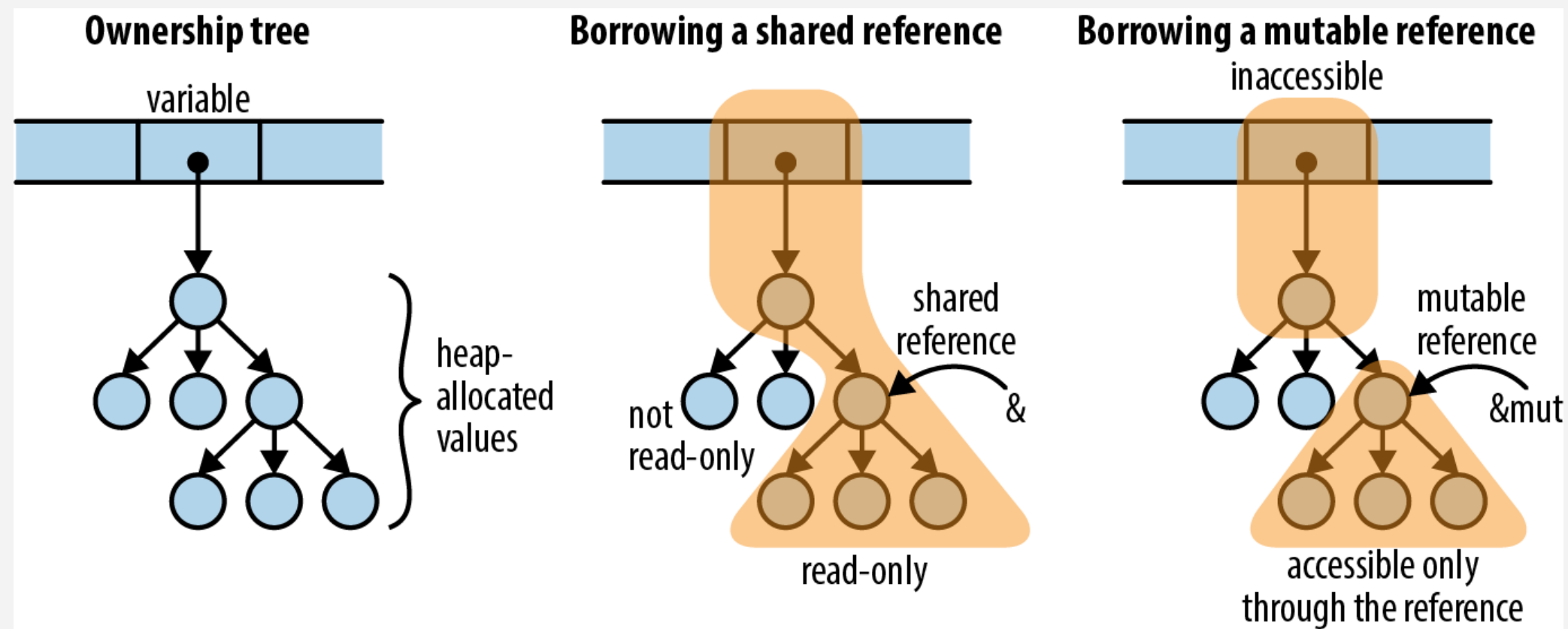
# 수명(Lifetime)

- Rust의 변경과 공유에 관한 규칙
  - **공유된 접근은 읽기 전용 접근이다.**
    - 공유된 레퍼런스로 빌려온 값은 읽을 수만 있다.  
공유된 레퍼런스가 살아 있는 동안에는 **그 무엇도** 참조 대상이나 참조 대상을 통해 도달할 수 있는 다른 대상을 변경할 수 없다.  
소유자 역시 읽기 전용으로 설정되기 때문에 이 구조에 관여된 대상의 변경할 수 있는 레퍼런스가 아예 존재할 수 없다.  
한마디로 동결 상태라고 보면 된다.
  - **변경할 수 있는 접근은 배타적인 접근이다.**
    - 변경할 수 있는 레퍼런스로 빌려온 값은 그 레퍼런스를 통해서만 접근할 수 있다.  
변경할 수 있는 레퍼런스가 살아 있는 동안에는 참조 대상이나 참조 대상을 통해 도달할 수 있는 다른 대상에 접근할 수 있는 경로가 없다.  
변경할 수 있는 레퍼런스와 수명이 겹칠 수 있는 유일한 레퍼런스는 변경할 수 있는 레퍼런스 그 자체에서 빌려온 것들 뿐이다.



# 수명(Lifetime)

- 레퍼런스는 유형에 따라서 참조 대상에 이르는 소유 경로상의 값들과 참조 대상을 통해 도달할 수 있는 값들을 가지고 할 수 있는 일이 다르다.



# Rust가 실수를 막는 방법

타입 변환, 열거체와 Option / Result, 패턴 매칭(Pattern Matching), Copy와 Clone, 부동소숫점과 정렬

# 타입 변환

- Rust는 암시적인 타입 변환이 없고, `as` 키워드를 통해 명시적인 타입 변환만 가능하다.

```
fn main() {  
    let a = 10;  
    let b = 30.4;  
  
    // Error: mismatched types  
    // let c = a + b;  
  
    // Use explicit type conversion  
    let c = a as f64 + b;  
  
    println!("{c}");  
}
```

# 열거체

- Rust의 열거체는 데이터를 가질 수도 있고, 타입이 꼭 같을 필요도 없다.

```
enum Status {  
    Idle,  
    Run(i32),  
    Attack { damage: i32 },  
    Defend { defense: i32 },  
}  
  
fn main() {  
    let mut status = Status::Idle;  
    status = Status::Run(10);  
    status = Status::Attack { damage: 20 };  
    status = Status::Defend { defense: 5 };  
};
```

# Option

- Rust에는 NULL이 존재하지 않는다.  
하지만 개발을 하다 보면 NULL이 필요할 때가 있는데, 이를 위해 만들어진 타입이다.

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

```
fn main() {  
    let x: Option<i32> = Some(5);  
    let y: Option<i32> = None;  
  
    println!("{:?}", x);  
    println!("{:?}", y);  
  
    println!("{}", x.unwrap_or(0));  
  
    match y {  
        Option::Some(v) => println!("Value: {}", v),  
        Option::None => println!("No value"),  
    }  
}
```

# Result

- 특정 함수의 동작 결과를 성공, 실패로 나타내기 위한 열거체 타입이다.

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

```
fn square_if_even(num: i32) -> Result<i32, String> {  
    if num % 2 == 0 {  
        Ok(num * num)  
    } else {  
        Err(String::from("Not even"))  
    }  
}  
  
fn main() {  
    let num1 = 4;  
    let num2 = 5;  
  
    match square_if_even(num1) {  
        Ok(v) => println!("Result: {v}"),  
        Err(e) => println!("Error: {e}"),  
    }  
  
    match square_if_even(num2) {  
        Ok(v) => println!("Result: {v}"),  
        Err(e) => println!("Error: {e}"),  
    }  
}
```

# 패턴 매칭(Pattern Matching)

- C/C++의 switch-case 문과 유사한 동작을 한다.
- Rust의 match 표현식은 값을 표현할 수 있는 모든 범위를 처리해야 한다.

```
fn main() {  
    let num = 100;  
  
    match num {  
        0 => println!("Zero");  
        1 | 3 | 5 | 7 | 9 => println!("1-digit Odd");  
        2..=8 => println!("1-digit Even");  
        num @ 10..=99 => println!("2-digit: {num}");  
        _ => println!("3-digit or more");  
    }  
}
```

# Copy와 Clone

- C++에서는 얇은 복사(Shallow Copy)로 인해 댕글링 포인터 문제가 발생할 때가 있다.
- 이 문제를 해결하려면 깊은 복사(Deep Copy)를 해야 한다.

```
class SpreadSheet
{
public:
    SpreadSheet(int id, int rows, int cols)
        : m_id(id), m_rows(rows), m_cols(cols)
    {
        m_data = new int *[rows];

        for (int i = 0; i < rows; ++i)
        {
            m_data[i] = new int[cols];
        }
    }

    ~SpreadSheet()
    {
        for (int i = 0; i < m_rows; ++i)
        {
            delete[] m_data[i];
        }

        delete[] m_data;
    }
};
```

```
void SetCell(int row, int col, int value)
{
    m_data[row][col] = value;
}

int GetCell(int row, int col) const
{
    return m_data[row][col];
}

private:
    int m_id;
    int m_rows;
    int m_cols;
    int **m_data;
};
```



# Copy와 Clone

- C++에서는 얇은 복사(Shallow Copy)로 인해 댕글링 포인터 문제가 발생할 때가 있다.
- 이 문제를 해결하려면 깊은 복사(Deep Copy)를 해야 한다.

```
int main()
{
    Spreadsheet sheet1(1, 10, 5);

    {
        Spreadsheet sheet2(sheet1);
        sheet1.SetCell(0, 0, 42);

        // sheet2.GetCell(0, 0) == 42
        std::cout << sheet2.GetCell(0, 0) << '\n';
    }

    // sheet1.GetCell(0, 0) == 42
    std::cout << sheet1.GetCell(0, 0) << '\n';

    return 0;
}
```

# Copy와 Clone

- C++에서는 얇은 복사(Shallow Copy)로 인해 댕글링 포인터 문제가 발생할 때가 있다.
- 이 문제를 해결하려면 깊은 복사(Deep Copy)를 해야 한다.

```
SpreadSheet(const SpreadSheet &src)
: SpreadSheet(src.m_id, src.m_rows, src.m_cols)
{
    for (int i = 0; i < m_rows; ++i)
    {
        for (int j = 0; j < m_cols; ++j)
        {
            m_data[i][j] = src.m_data[i][j];
        }
    }
}
```

# Copy와 Clone

- Rust는 얇은 복사와 깊은 복사를 위한 트레이트 Copy와 Clone을 구분하고,
- 얇은 복사를 했을 때 문제가 생길 수 있는 타입에 대해 Copy 트레이트를 구현하지 않고 Clone 트레이트만 구현해 깊은 복사만 할 수 있도록 강제한다.

```
fn hello(name: String) {  
    println!("Hello, {name}");  
}  
  
fn square(num: i32) -> i32 {  
    num * num  
}  
  
fn main() {  
    // Only implement Clone trait, not Copy  
    let name = String::from("Chris");  
    hello(name.clone());  
    hello(name);  
  
    // Implement Copy and Clone traits  
    let num = 5;  
    println!("square({num}): {}", square(num));  
    println!("square({}) : {}", num + 5, square(num + 5));  
}
```

# 부동소숫점과 정렬

- 다음 코드는 컴파일이 되지 않는다. 왜 그럴까?

```
fn main() {  
    let mut arr = vec![1.2, 4.5, 3.1, -5.7, 6.3];  
  
    arr.sort();  
  
    println!("{:?}", arr);  
}
```

# 부동소숫점과 정렬

- Rust에서 `sort()`를 사용하기 위해선 `Ord` 트레이트가 구현된 타입이어야 한다.
- 하지만 부동소숫점 타입인 `f32`와 `f64`는 `Ord` 트레이트가 구현되어 있지 않다. 왜 그럴까?

```
error[E0277]: the trait bound `{float}: Ord` is not satisfied
--> .\2 - Examples\f64_sort.rs:5:9
   |
5  |     arr.sort();
   |           ^^^^ the trait `Ord` is not implemented for `{float}`
   |
= help: the following other types implement trait `Ord`:
        isize
        i8
        i16
        i32
        i64
        i128
        usize
        u8
        and 4 others
note: required by a bound in `slice::<impl [T]>::sort`
--> C:\Users\utilForever\.rustup\...\rust\library\alloc\src\slice.rs:209:12
207 |     pub fn sort(&mut self)
    |               ----- required by a bound in this associated function
208 |     where
209 |         T: Ord,
    |           ^^^ required by this bound in `slice::<impl [T]>::sort`
```

# 부동소숫점과 정렬

- Rust는 각 연산마다 대부분 트레잇이 하나 존재한다. (예 : Add, Sub 등)
  - 하지만 비교 연산은 트레잇이 2개 있다.
    - 일치 연산 : Eq, PartialEq
    - 비교 연산 : Ord, PartialOrd
  - 이렇게 만든 이유는 이산수학 때 배웠던 **동치 관계(Equivalence Relation)** 때문이다.
    - 반사 관계(Reflexive) : 임의의  $x \in X$ 에 대하여,  $x \sim x$
    - 대칭 관계(Symmetric) : 임의의  $x, y \in X$ 에 대하여, 만약  $x \sim y$ 라면  $y \sim x$
    - 추이적 관계(Transitive) : 임의의  $x, y, z \in X$ 에 대하여, 만약  $x \sim y$ 이고  $y \sim z$ 라면  $x \sim z$

# 부동소숫점과 정렬

- 대부분의 기본 타입은 동치 관계 조건을 모두 만족한다.  
(완전 동치 관계 : Full Equivalence Relation)
- 하지만 부동소숫점은 동치 관계 조건 중 반사 관계를 만족하지 않는다.  
(부분 동치 관계 : Partial Equivalence Relation)
- 그 이유는 부동소숫점 연산 과정에서 발생할 수 있는 NaN 때문이다.  
이로 인해 Rust는 부동소숫점인 f32, f64 타입은 Eq, Ord 트레이트를 구현하지 않았다.

# 부동소숫점과 정렬

- Rust에서 부동소숫점 타입이 저장된 컨테이너를 정렬하려면 다음과 같이 해야 한다.

```
fn main() {  
    let mut arr = vec![1.2, 4.5, 3.1, -5.7, 6.3];  
  
    // Can't use arr.sort() because f64 doesn't implement Ord  
    // arr.sort();  
  
    // Instead, use sort_by  
    arr.sort_by(|a, b| a.partial_cmp(b).unwrap());  
  
    println!("{:?}", arr);  
}
```



# Rust의 메모리 관리 활용

클로저(Closure), 동시성(Concurrency)

# 클로저(Closure)

- 익명 함수로, 람다 표현식(Lambda Expression)이라고 말하기도 한다.
- Rust는 클로저에 작성하는 코드에 따라 클로저의 트레잇 구현을 달리 한다.

```
fn call_twice<F>(closure: F) where F: Fn() {  
    closure();  
    closure();  
}  
  
fn main() {  
    let name = String::from("Chris");  
    let hello = || {  
        println!("Hello, {name}");  
        drop(name);  
    };  
  
    call_twice(hello);  
}
```

# 클로저(Closure)

- 익명 함수로, 람다 표현식(Lambda Expression)이라고 말하기도 한다.
- Rust는 클로저에 작성하는 코드에 따라 클로저의 트레잇 구현을 달리 한다.

```
error[E0525]: expected a closure that implements the `Fn` trait, but this closure only implements `FnOnce`
--> .\14_closure.rs:8:17
   |
 8 |     let hello = || {
   |                  ^^ this closure implements `FnOnce`, not `Fn`
 9 |         println!("Hello, {name}");
10 |         drop(name);
   |         ---- closure is `FnOnce` because it moves the variable `name` out of its environment
...
13 |     call_twice(hello);
   |     ----- the requirement to implement `Fn` derives from here
   |     |
   |     required by a bound introduced by this call
note: required by a bound in `call_twice`
--> .\14_closure.rs:1:39
   |
 1 | fn call_twice<F>(closure: F) where F: Fn() {
   |                                     ^^^^ required by this bound in `call_twice`

error: aborting due to 1 previous error

For more information about this error, try `rustc --explain E0525`.
```

# 동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.
  - 멀티 스레드에서는 스레드 세이프한 타입만 사용할 수 있게 제한한다.

```
use std::rc::Rc;
use std::thread;

fn main() {
    let rc1 = Rc::new("Hyundai".to_string());
    let rc2 = rc1.clone();

    thread::spawn(move || {
        // Error
        rc2.clone();
    });

    rc1.clone();
}
```

# 동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.
  - 멀티 스레드에서는 스레드 세이프한 타입만 사용할 수 있게 제한한다.

```
error[E0277]: `Rc<String>` cannot be sent between threads safely
--> .\15_concurrency.rs:8:19
8   |         thread::spawn(move || {
    |         ^^^^^^^^^^^^^^^^^^^^^
    |         |
    |         within this `{closure@.\15_concurrency.rs:8:19: 8:26}`
    |         required by a bound introduced by this call
9   |             // Error
10  |             rc2.clone();
11  |         });
    |         ^ `Rc<String>` cannot be sent between threads safely

= help: within `{closure@.\15_concurrency.rs:8:19: 8:26}`, the trait `Send` is not implemented for `Rc<String>`,
which is required by `{closure@.\15_concurrency.rs:8:19: 8:26}: Send`
```

# 동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.
  - 멀티 스레드에서는 스레드 세이프한 타입만 사용할 수 있게 제한한다.

```
use std::sync::Arc;
use std::thread;

fn main() {
    let arc1 = Arc::new("Hyundai".to_string());
    let arc2 = arc1.clone();

    thread::spawn(move || {
        // No error
        let _ = arc2.clone();
    });

    let _ = arc1.clone();
}
```

# 동시성(Concurrency)

- Rust의 안전성은 멀티 스레드 프로그래밍에서 빛을 발한다.
  - 자식 스레드가 패닉에 빠졌을 때 다른 스레드로 전파할 것인가 여부를 제어할 수 있다.

```
use std::{thread, time::Duration};

fn main() {
    let handle1 = thread::spawn(|| {
        for i in 1..=5 {
            println!("Thread 1: Running - {i}");
            thread::sleep(Duration::from_millis(500));
        }
        42
    });
    let handle2 = thread::spawn(|| {
        println!("Thread 2: Running");
        panic!("Thread 2: Panic!");
    });

    match handle1.join() {
        Ok(result) => println!("Thread 1 is completed: {result}"),
        Err(e) => println!("Panic occurs in Thread 1: {e:?}"),
    }
    match handle2.join() {
        Ok(_) => println!("Thread 2 is completed"),
        Err(e) => println!("Panic occurs in Thread 2: {e:?}"),
    }
}
```

# 정리

- Rust의 안전성은 보수적으로 설계되어 있어 이해하기에 많이 어렵습니다.
- Rust를 배워 보고 싶으신 분들은 Rust 공식 문서 또는 서점에 좋은 책들이 많이 나와 있으니 참고하시기 바랍니다.
- 또는 제 강의 자료를 참고하시면 감사합니다.
  - 백엔드 프로그래밍 : <https://github.com/utilForever/2022-Korea-Rust-Backend>
  - 인터프리터 만들기 : <https://github.com/utilForever/2023-MatKor-Rust-Interpreter>
  - 마인크래프트 만들기 : <https://github.com/utilForever/2023-UNIST-Rust-Minecraft>
  - 크로스 플랫폼 프로그래밍 : <https://github.com/utilForever/2024-SNU-Rust-Application>
  - 리눅스 커널 프로그래밍 : <https://github.com/utilForever/2024-HSPACE-Rust-LinuxKernel>



# Thank You



옥찬호

42dot / Embedded Software Engineer

utilforever@gmail.com / @utilforever