HSPACE Rust 특강

Rust Basic #1 - 소유와 이동

Chris Ohk

utilForever@gmail.com

소유와 이동

- 소유 (Ownership)
- 이동 (Moves)
- `Copy` 타입 : 이동의 예외
- `Rc`와 `Arc` : 공유된 소유권

들어가며

- 우리가 프로그래밍 언어에 기대하는 메모리 관리에 관한 특성
  - 메모리가 원하는 시점에 지체 없이 해제되면 좋겠다. 그래야 프로그램의 메모리 소모량을 통제할 수 있다.
  - 해제된 오브젝트를 가리키는 포인터는 절대로 사용하고 싶지 않다.
    이런 상황은 미정의 동작으로 이어져 크래시와 보안 허점을 야기한다.

들어가며

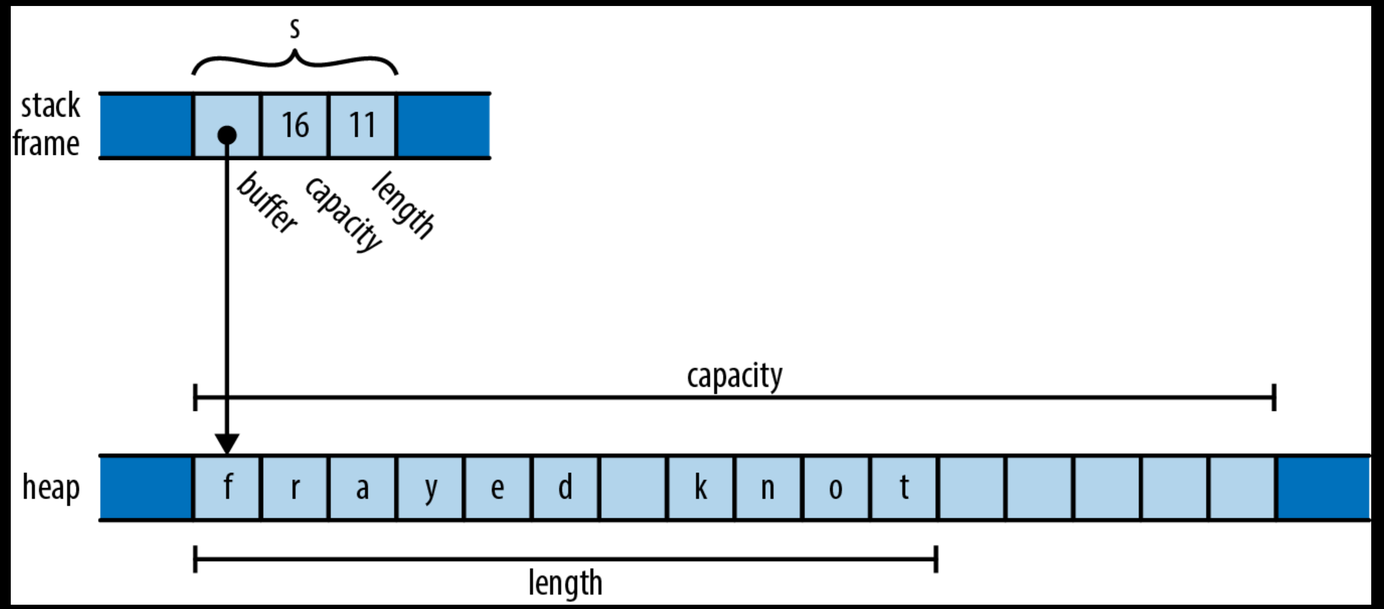
- 대부분의 프로그래밍 언어들은 크게 두 진영으로 나뉜다.
  - ‘안전 우선’ 진영은 가비지 컬렉션(Garbage Collection)을 써서 메모리를 관리한다.
    가비지 컬렉션은 어떤 오브젝트가 자신에게 도달 가능한 포인터를 모두 잃었을 때
    그 오브젝트를 자동으로 해제해 주는 기능으로, 포인터가 남아 있는 한 대상이 되는 오브젝트를
    계속 유지해 두는 단순한 방법을 써서 대상을 잃은 포인터가 생기는 걸 방지한다.
    Python, JavaScript, Ruby, Java, C#, Haskell 등 대부분의 최신 언어들이 이 진영에 속한다.
  - 그러나 가비지 컬렉션에 의존한다는 건 오브젝트가 해제되는 동안에 통제권이 가비지 컬렉터에게로 넘어간다는 걸 의미한다. 가비지 컬렉터는 보통 생각하는 것보다 덩치가 크고 무거우며, 메모리가 예상 시점에 해제되지 않는 이유를 이해하기도 여간 까다로운 게 아니다.

들어가며

- 대부분의 프로그래밍 언어들은 크게 두 진영으로 나뉜다.
  - ‘제어 우선’ 진영은 메모리 해제의 책임을 여러분에게 맡긴다.
    프로그램의 메모리 사용 전략을 전적으로 여러분이 알아서 구사할 수 있는 반면,
    대상을 잃은 포인터가 생기는 걸 방지하는 일 역시 여러분이 전적으로 책임져야 한다.
    C와 C++이 이 진영에 속하는 유일한 주류 언어다.
  - 그러나 이 방식은 절대로 실수하지 않는 존재에게나 어울릴 뿐, 모름지기 사람이라면 누구나 실수하기 마련이다.
    잘못된 포인터 사용은 알려진 보안 문제를 수집해 온 이래로 줄곧 문제의 공통된 원인으로 지목되어 왔다.

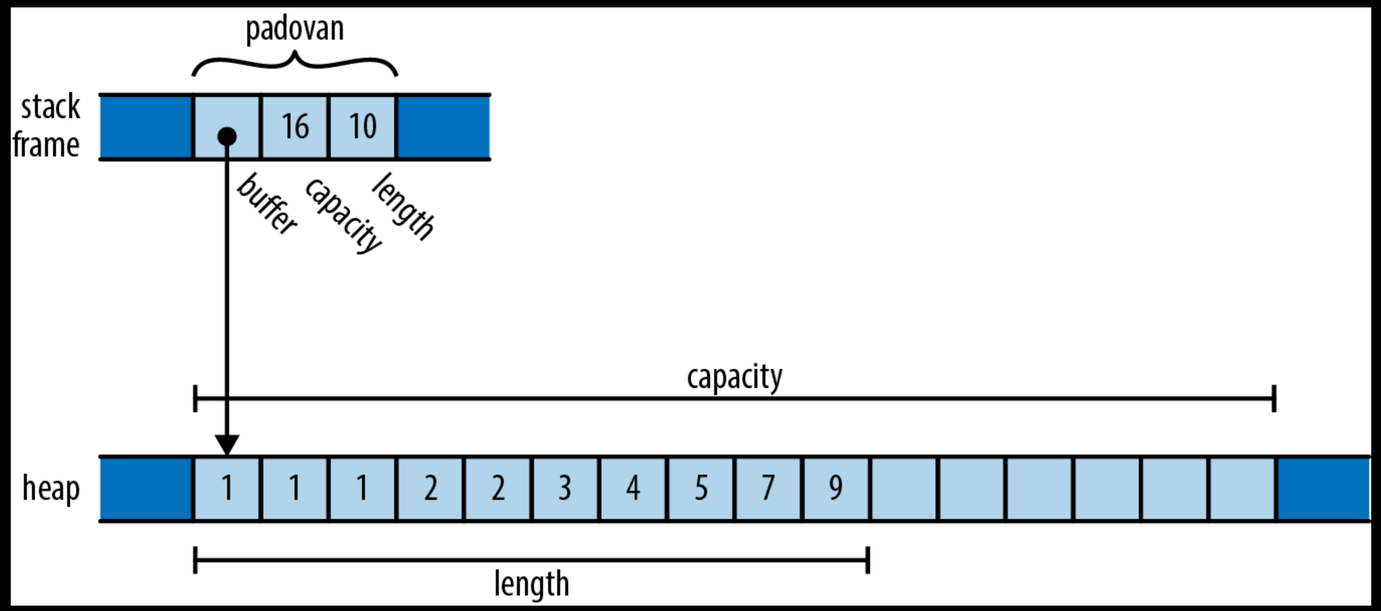
소유 (Ownership)

- 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 한다는 것이 일반적인 인식이다. 소유자가 소유물의 수명을 결정하면 나머지는 그 결정을 존중해야 한다.



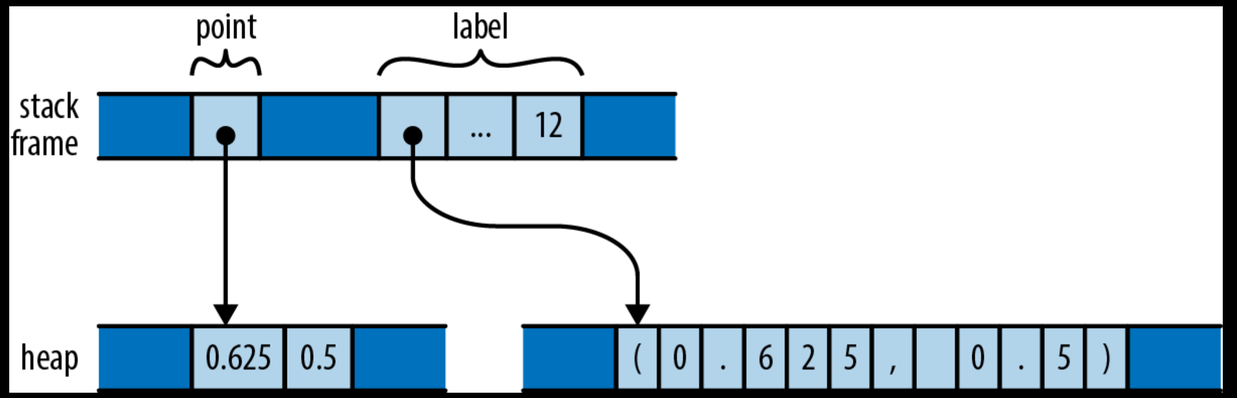
소유 (Ownership)

- Rust에서는 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.



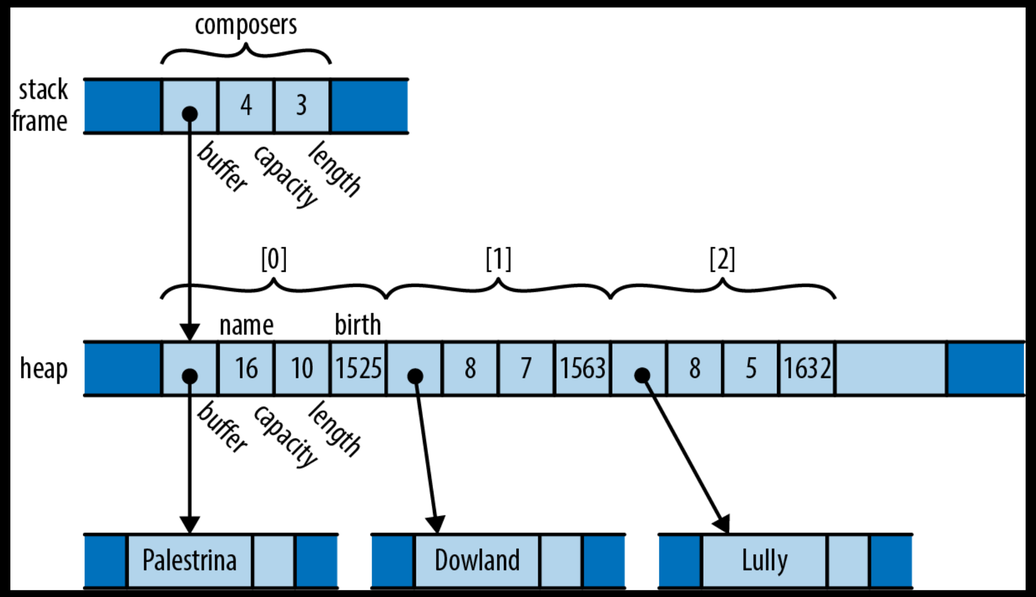
소유 (Ownership)

- Rust에서는 소유의 개념이 언어 자체에 내장되어 있으며, 컴파일 타임 검사를 통해 검증된다.
  - Box 타입은 소유의 또 다른 예다.



소유 (Ownership)

- Rust에서는 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.



소유 (Ownership)

- 소유의 개념은 여전히 너무 엄격해서 쓰기가 어렵다.
  따라서 Rust는 다음과 같은 방법을 통해서 이 단순한 아이디어를 확장한다.
  - 값을 한 소유자에게서 다른 소유자에게로 옮길 수 있다. 이를 통해서 트리로 만들고 바꾸고 허무는 것이 가능하다.
  - 정수, 부동소숫점 수, 문자 같은 아주 단순한 타입들은 소유 규칙의 적용 대상에서 제외한다.
    이런 타입을 `Copy` 타입이라고 한다.
  - 표준 라이브러리의 레퍼런스 카운트 기반의 포인터 타입인 `Rc`와 `Arc`를 제공한다.
    이들을 사용하면 약간의 제약이 따르긴 하지만 값이 여러 소유자를 가질 수 있다.
  - 값의 '레퍼런스를 빌려'올 수 있다. 레퍼런스는 한정된 수명을 가진 소유권이 없는 포인터다.

소유 (Ownership)

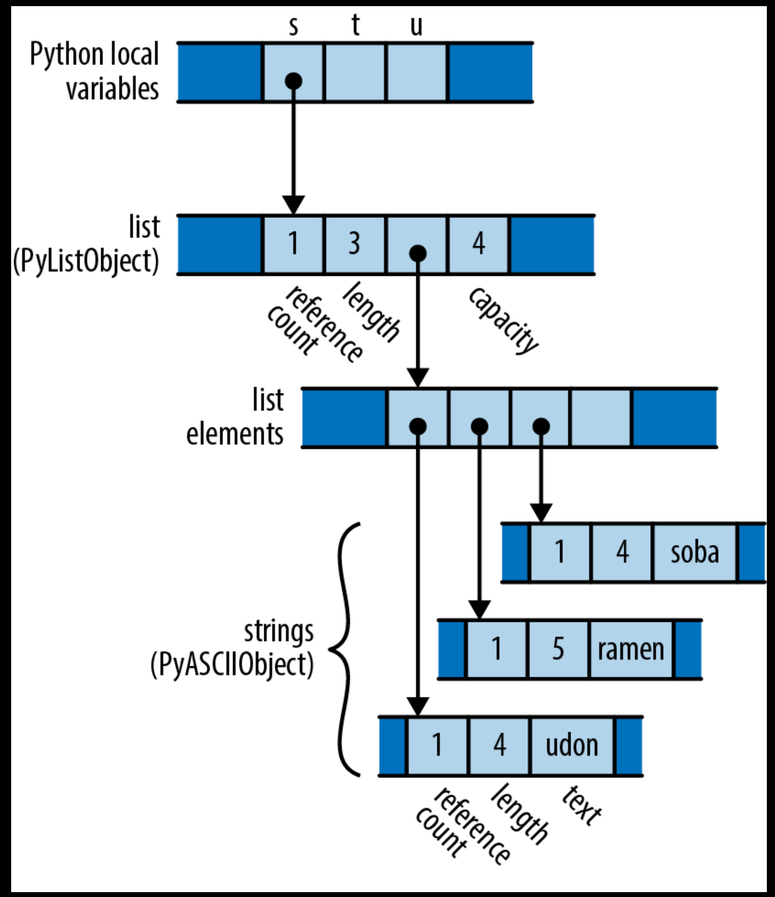
- 소유의 개념은 여전히 너무 엄격해서 쓰기가 어렵다.
  따라서 Rust는 다음과 같은 방법을 통해서 이 단순한 아이디어를 확장한다.
  - 값을 한 소유자에게서 다른 소유자에게로 옮길 수 있다. 이를 통해서 트리로 만들고 바꾸고 허무는 것이 가능하다.
  - 정수, 부동소숫점 수, 문자 같은 아주 단순한 타입들은 소유 규칙의 적용 대상에서 제외한다.
    이런 타입을 `Copy` 타입이라고 한다.
  - 표준 라이브러리의 레퍼런스 카운트 기반의 포인터 타입인 `Rc`와 `Arc`를 제공한다.
    이들을 사용하면 약간의 제약이 따르긴 하지만 값이 여러 소유자를 가질 수 있다.
  - 값의 '레퍼런스를 빌려'올 수 있다. 레퍼런스는 한정된 수명을 가진 소유권이 없는 포인터다.

이동 (Moves)

- Rust에서는 값을 변수에 대입하거나, 함수에 전달하거나, 함수에서 반환하거나 하는 식의 연산이 일어날 때 배부분 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.

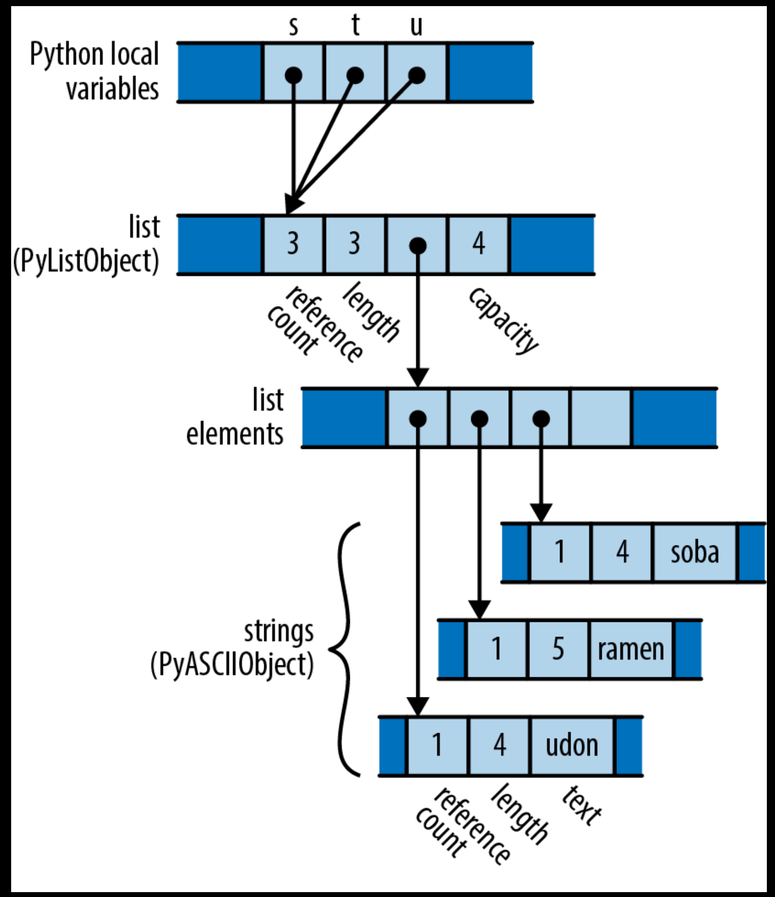
이동 (Moves)

- 각 언어가 대입을 처리하기 위해 선택한 방법을 자세히 살펴 보자.
  - Python



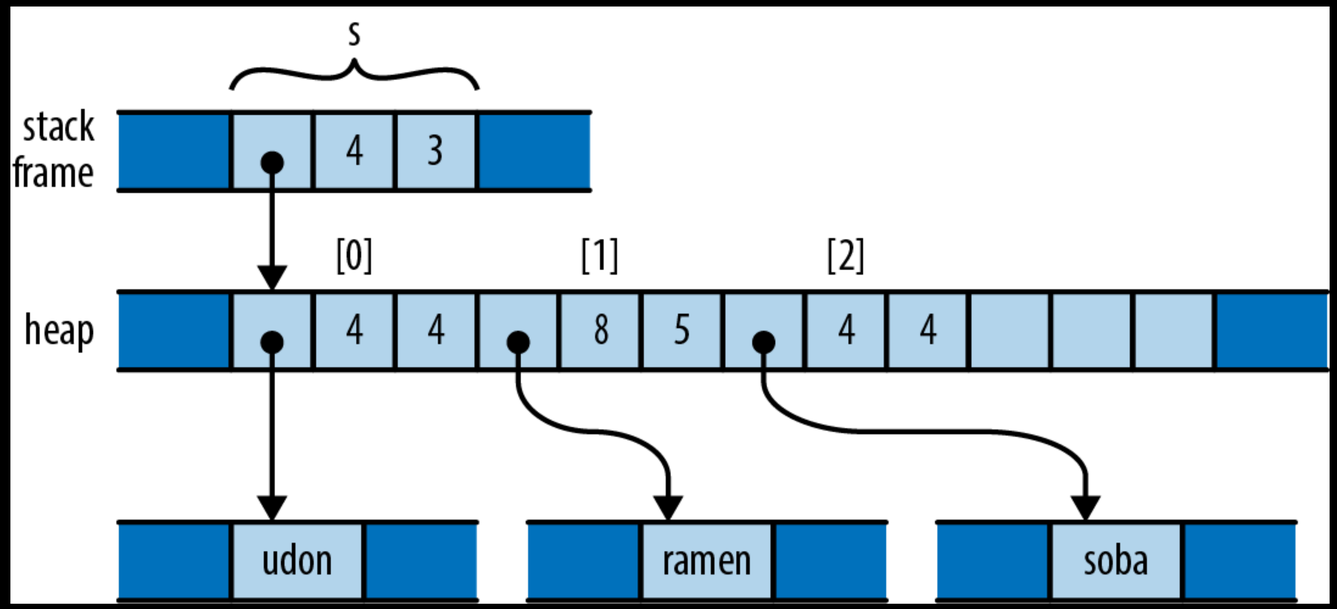
이동 (Moves)

- 각 언어가 대입을 처리하기 위해 선택한 방법을 자세히 살펴 보자.
  - Python



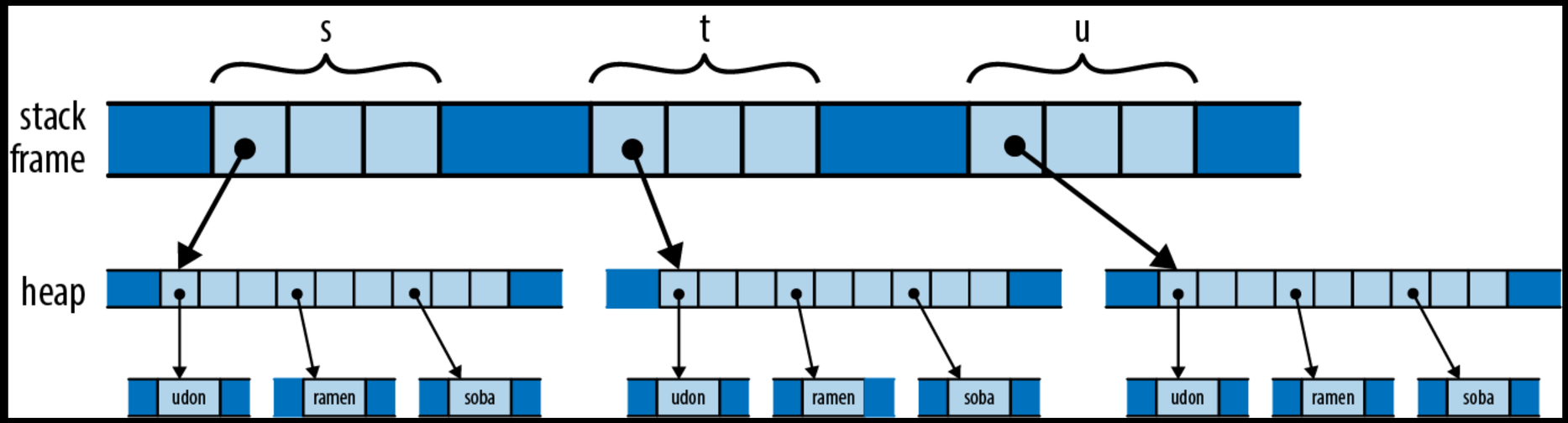
이동 (Moves)

- 각 언어가 대입을 처리하기 위해 선택한 방법을 자세히 살펴 보자.
  - C++



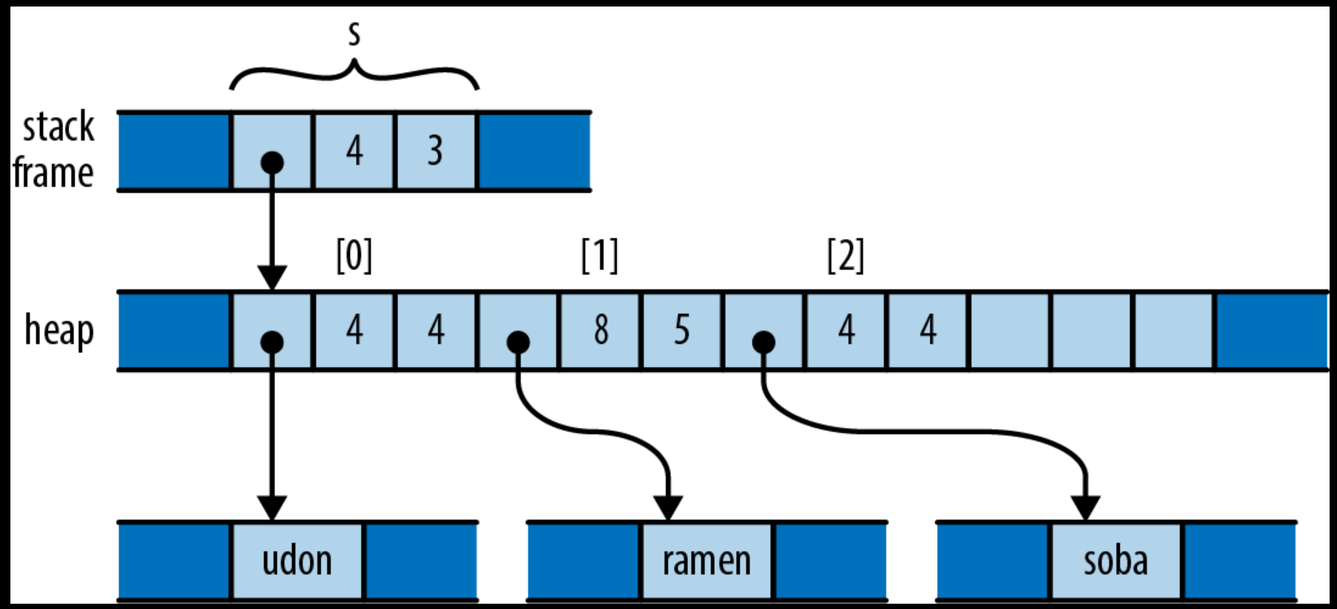
이동 (Moves)

- 각 언어가 대입을 처리하기 위해 선택한 방법을 자세히 살펴 보자.
  - Rust



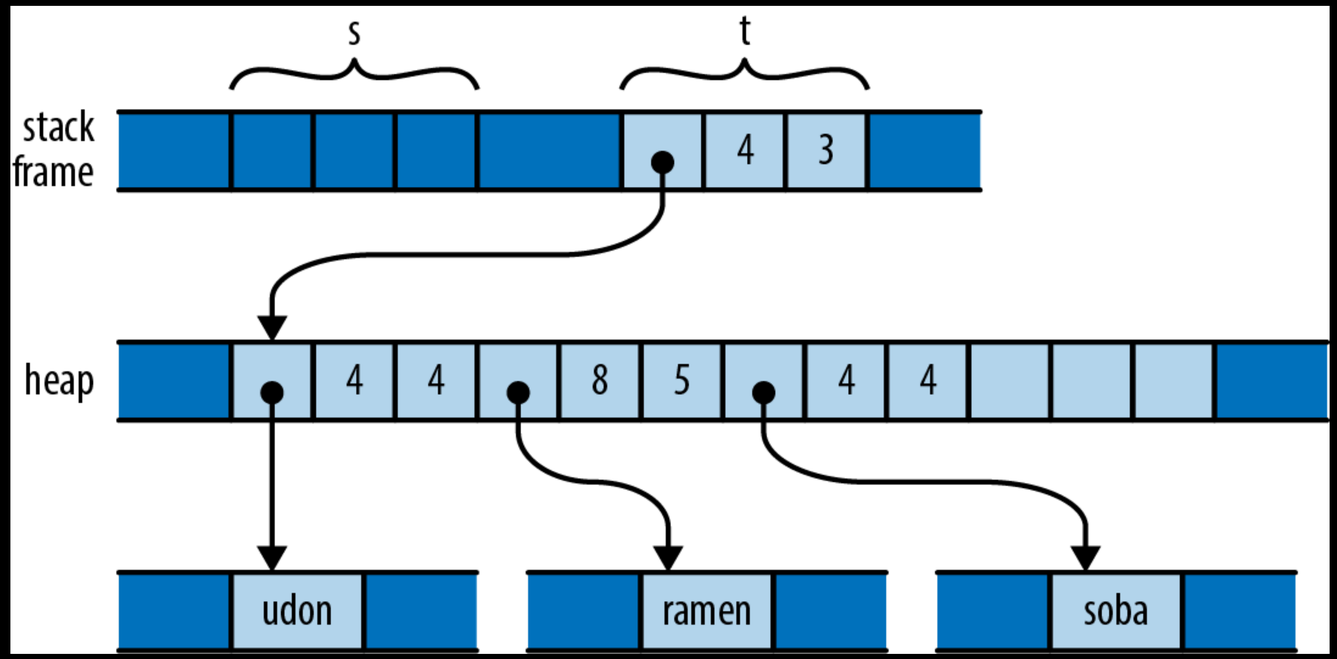
이동 (Moves)

- 각 언어가 대입을 처리하기 위해 선택한 방법을 자세히 살펴 보자.
  - Rust



이동 (Moves)

- 각 언어가 대입을 처리하기 위해 선택한 방법을 자세히 살펴 보자.
  - Rust



이동 (Moves)

- 이동으로 처리되는 그 밖의 연산들
  - 값을 이미 초기화된 변수로 옮기는 경우에는 Rust가 그 변수의 이전 값을 드롭한다.
  - Rust는 값이 쓰이는 거의 모든 곳에 이동 문법(Move Semantics)을 적용한다.
    - 함수에 인수를 전달하면 소유권이 함수의 매개 변수로 이동하고, 함수에서 값을 반환하면 소유권이 호출부로 이동하고, 튜플을 만들면 값이 튜플 안으로 이동하는 식이다.

이동 (Moves)

- 이동으로 처리되는 그 밖의 연산들

```
struct Person { name: String, birth: i32 }

let mut composers = Vec::new();
composers.push(Person { name: "Palestrina".to_string(),
                        birth: 1525 });
```

  - 함수에서 값을 반환할 때
    - `Vec::new()` 호출은 새 벡터를 생성하고 벡터의 포인터가 아니라 벡터 그 자체를 반환한다.
      이때 벡터의 소유권은 `Vec::new`에서 `composers` 변수로 이동한다. 마찬가지로 `to_string` 호출은 새 `String` 인스턴스를 반환한다.
  - 새 값을 생성할 때
    - 새 `Person` 구조체의 `name` 필드는 `to_string`의 반환 값으로 초기화된다. 이때 문자열의 소유권은 구조체가 갖는다.
  - 함수로 값을 전달할 때
    - 벡터의 `push` 메소드에 전달되는 것은 `Person` 구조체의 포인터가 아니라 구조체 그 자체이며, 전달된 구조체는 벡터 끝으로 이동된다.
    - 벡터는 이 `Person`의 소유권을 가지며, 따라서 `name`이 쥐고 있는 `String`의 간접 소유자가 된다.

이동 (Moves)

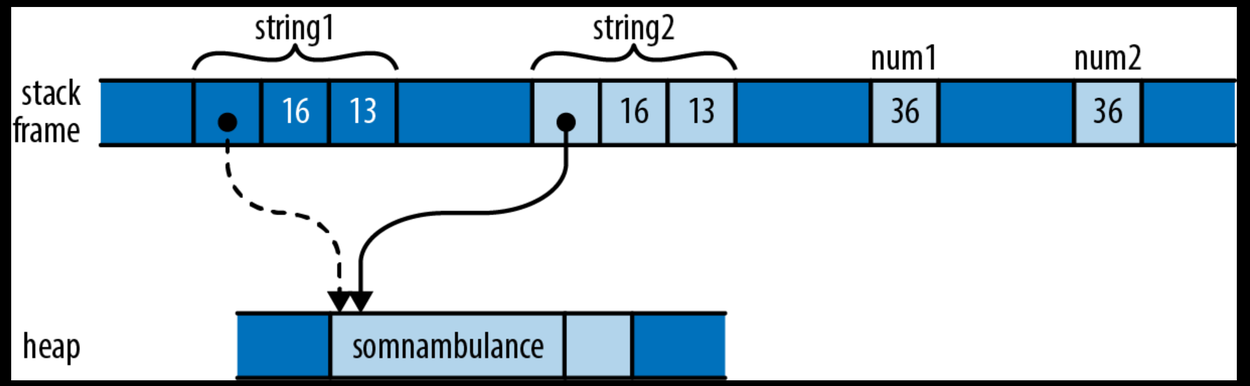
- 이동과 제어 흐름
  - 어떤 변수가 있을 때 그의 값이 이동될 가능성이 있고, 이동된 이후에 확실하게 새 값이 주어지지 않았다면,
    그 변수는 미초기화 상태로 간주된다.
  - 비슷한 이유로 반복문 안에서 변수를 이동하는 것은 금지되어 있다.
  - 단, 다음 반복에 들어가기 전까지 확실하게 새 값을 부여해 두면 문제가 없다.

이동 (Moves)

- 색인을 써서 접근하는 값의 이동
  - 모든 종류의 값 소유자가 미초기화 상태가 될 수 있는 건 아니다.
  - 예를 들어, 벡터의 일부 요소가 밖으로 이동되고 나면
    Rust는 어떻게든 그 요소들이 미초기화 상태라는 걸 기억해 두고 벡터가 드롭될 때까지 그 정보를 추적해야 한다.
  - 벡터는 그냥 벡터여야 하며, 실제로 Rust는 해당 코드에 대해 오류를 내며 거부한다.
  - 정리하자면, 벡터의 일부 요소를 이동하는 건 불가능하고 벡터 전체를 이동하는 건 가능하다.

Copy 타입 : 이동의 예외

- 이동되는 값은 주로 벡터와 문자열처럼 잠재적인 메모리 사용량이 많고, 복사하는데 비용이 많이 드는 타입인 경우가 많다.
  - 이동은 이런 타입의 소유권을 파악하기 쉽게 해주고, 대입하는데 드는 비용을 크게 줄여 준다.
  - 그러나 정수나 문자 같은 단순한 타입의 경우에는 이런 세심한 처리가 사실 불필요하다.



Copy 타입 : 이동의 예외

- 대부분의 타입이 이동되지, 모든 타입이 이동되지는 않는다.
  - 여기에는 예외가 있는데 Rust가 Copy 타입으로 지정해 둔 타입들이 바로 그것이다.
  - Copy 타입의 값을 대입하면 이동되지 않고 복사된다.
    대입이 끝나도 원본은 가지고 있던 값을 그대로 유지하고 있어서 계속 사용할 수 있다.
  - 표준 Copy 타입에는 모든 정수와 부동소숫점 수 타입, char와 bool 타입, 그리고 기타 몇 가지가 포함된다.
    Copy 타입으로 된 튜플이나 고정 길이 배열도 Copy 타입이다.
  - 간단한 비트 단위 복사로 복사본을 만들 수 있는 타입만 Copy가 될 수 있다.
    앞에서 이미 설명한 것처럼 String은 힙에 할당된 버퍼를 소유하므로 Copy 타입이 아니다.
    비슷한 이유로 Box<T>는 힙에 할당된 자신의 지칭 대상을 소유하므로 Copy가 아니다.
  - 대충 값을 드롭할 때 특별한 처리가 필요한 타입은 Copy가 될 수 없다고 보면 된다.

Copy 타입 : 이동의 예외

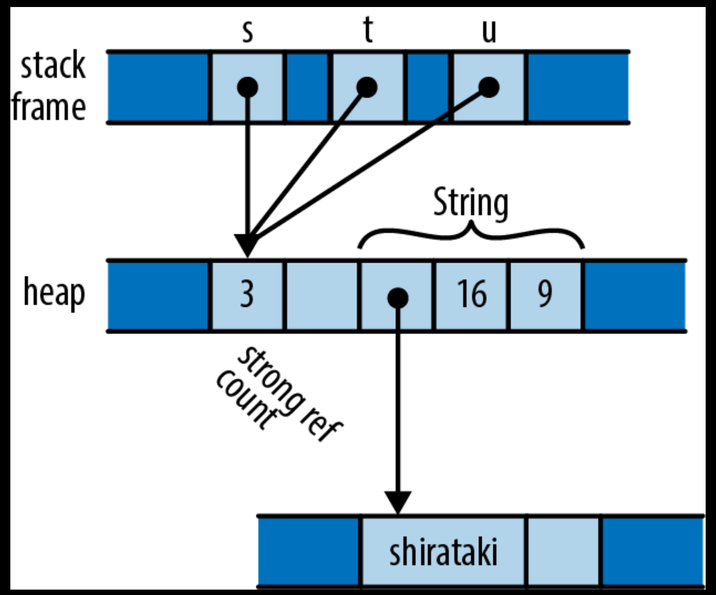
- 사용자 정의 타입의 경우는 어떨까? 기본적으로 `struct`와 `enum` 타입은 `Copy`가 아니다.
  - 사용자 정의 타입이 `Copy`가 아니라고 말하는 건 단지 기본 동작이 그렇다는 얘기다.
  - 만약 구조체의 모든 필드가 `Copy`라면, 정의 위에 `#[derive(Copy, Clone)]` 애트리뷰트를 둬서 타입을 `Copy`로 만들 수 있다.
  - 하지만 `Copy`가 아닌 타입으로 된 필드를 갖고 있는 경우에는 이 방법이 통하지 않는다.

Rc와 Arc : 공유된 소유권

- 일반적인 Rust 코드에서는 대부분의 값이 하나의 소유자를 갖지만,
  모든 값에 대해서 원하는 수명을 갖는 소유자를 하나만 찾기가 어려울 때도 있다.
  - 예를 들어, 어떤 값이 하나 있고 그 값의 사용자가 여럿일 때
    모든 사용자가 작업을 마칠 때까지 그 값을 살려 두고 싶은 경우가 그렇다.
  - Rust는 이런 경우를 위해서 레퍼런스 카운트 기반의 포인터 타입 `Rc`와 `Arc`를 제공한다.
  - `Rc`와 `Arc` 타입은 거의 비슷하다. 유일한 차이점은 `Arc`의 경우 스레드 간에 직접 공유해도 안전한 반면,
    `Rc`는 스레드 안전성을 갖지 않는 더 빠른 코드를 써서 레퍼런스 카운트를 갱신한다는 점이다.
    (`Arc`는 원자적인 레퍼런스 카운트(Atomic Reference Count)의 줄임말이다.)
  - 스레드 간에 포인터를 공유할 일이 없다면 `Arc`가 갖는 성능상의 불이익을 감수할 이유가 없으므로 `Rc`를 쓰면 된다.

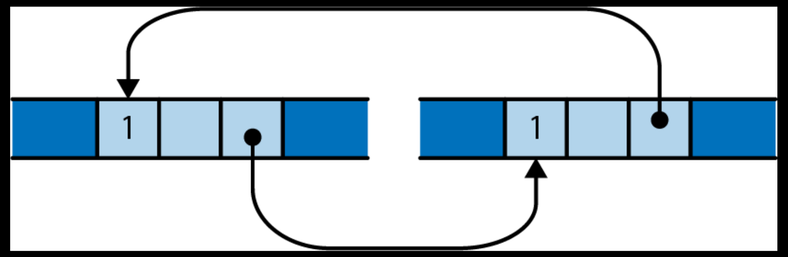
Rc와 Arc : 공유된 소유권

- 임의의 타입 T에 대해서 Rc<T> 값은 레퍼런스 카운트를 내장한 힙에 할당된 T를 가리키는 포인터다.
  - Rc<T> 값을 복제하면 T가 복사되는 게 아니라 이를 가리키는 또 다른 포인터가 만들어지고 레퍼런스 카운트가 증가된다.



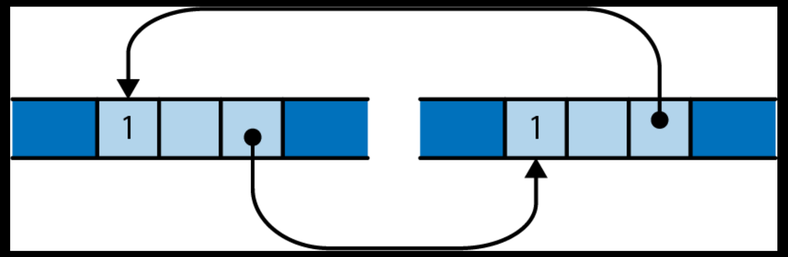
Rc와 Arc : 공유된 소유권

- 레퍼런스 카운트로 메모리를 관리할 때 겪을 수 있는 잘 알려진 문제
  - 레퍼런스 카운트를 쓰는 두 값이 서로 상대방을 가리키는 경우에는
    둘 다 레퍼런스 카운트가 0이 될 수 없어서 값이 영영 해제되지 않는다.
  - 이런 상황에 맞닥뜨리면 Rust에서도 메모리 누수가 생길 수 있지만, 그럴 일은 거의 없다.
    Rc 포인터는 자신이 가리키는 대상을 변경할 수 없도록 쥐고 있기 때문이다.



Rc와 Arc : 공유된 소유권

- 레퍼런스 카운트로 메모리를 관리할 때 겪을 수 있는 잘 알려진 문제
  - 하지만 Rust는 변경할 수 없는 값의 일부분을 변경할 수 있도록 만드는 방법을 제공한다.
    이를 내부 가변성(Interior Mutability)라고 한다.
  - 연결 일부를 약한 포인터(Weak Pointer)인 `std::rc::Weak`로 만들어서
    `Rc` 포인터의 순환 구조가 형성되는 걸 피할 수도 있다.



감사합니다.

utilForever@gmail.com

https://github.com/utilForever

Facebook, Twitter: @utilForever