

HSPACE Rust 특강

Rust Basic #3 - 표현식

Chris Ohk

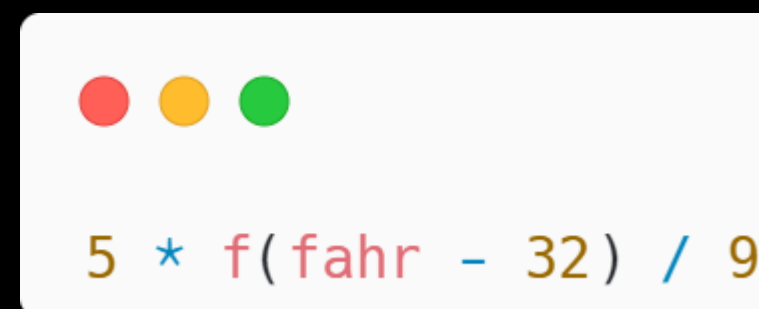
utilForever@gmail.com

- 표현식 언어
- 우선순위와 결합성
- 블록과 세미콜론
- 선언
- `if`와 `match`
- `if let`
- 루프
- 루프의 제어 흐름
- `return` 표현식

- Rust에 loop가 있는 이유
- 함수와 메소드 호출
- 필드와 요소
- 레퍼런스 연산자
- 산술, 리터럴, 비교, 논리 연산자
- 대입
- 타입 캐스팅
- 클로저

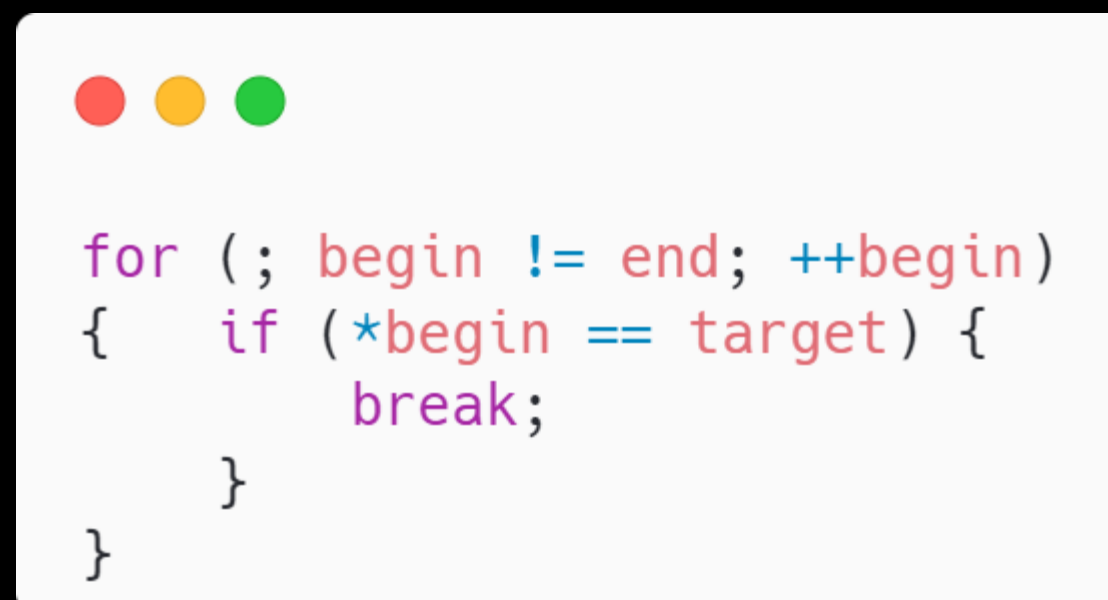
- C에서는 표현식(Expression)과 실행문(Statement) 간에 뚜렷한 차이가 있다.

- 표현식은 값을 갖는다.



```
5 * f(fahr - 32) / 9
```

- 반면 실행문은 값을 갖지 않는다.



```
for (; begin != end; ++begin)
{
    if (*begin == target) {
        break;
    }
}
```

- Rust는 표현식 언어(Expression Language)다.
- C에서는 if와 switch가 실행문이다. 이들은 값을 산출하지 않으며, 표현식 중간에 쓰일 수 없다.
- Rust에서는 if와 match가 값을 산출하는 것이 가능하다. Rust에 삼항 연산자가 없는 이유가 바로 여기에 있다.

```
let status = if cpu.temperature <= MAX_TEMP {  
    HttpStatus::Ok  
} else {  
    HttpStatus::ServerError  
};
```

```
println!(  
    "Inside the vat, you see {}.",  
    match var.contents {  
        Some(brain) => brain.desc(),  
        None => "nothing of interest",  
    }  
);
```

- Rust의 표현식 문법은 다음 사이트를 참고하자.

<https://doc.rust-lang.org/reference/expressions.html>

- Rust는 대부분의 프로그래밍 언어들처럼 연산자 우선순위(Operator Precedence)를 두고 있다.
연산자 우선순위는 표현식이 인접한 여러 인자를 가질 때 연산의 순서를 결정한다.
- 연산자 결합성(Operator Associativity)이란 연산자들의 우선순위가 같을 때 연산의 순서를 결정한다.

- 블록은 가장 일반적인 종류의 표현식이다.
- 값을 산출하며, 값이 필요한 모든 곳에 쓰일 수 있다.
- 블록의 마지막 줄이 세미콜론으로 끝날 경우 블록의 반환 값은 ()가 된다.
만약 마지막 줄의 세미콜론을 생략하면, 블록의 반환 값은 마지막 표현식의 값이 된다.

```
let display_name = match post.author() {  
    Some(author) => author.name(),  
    None => {  
        let network_info = post.get_network_metadata()?;  
        let ip = network_info.client_address();  
  
        ip.to_string()  
    }  
};
```

- 일반적으로 많이 사용하는 문장이다.

```
let x: i64 = 30;
```

- 변수 이름은 반드시 문자나 밑줄로 시작해야 하며, 숫자는 첫번째 문자 이후에만 올 수 있다.
(단, Rust는 '문자'의 정의를 폭넓게 가져간다. 그리스 문자, 악센트가 있는 라틴 문자, 일부 유니코드 등이 가능하다.)
- let 선언문은 초기화 없이 변수를 선언할 수 있다. 이렇게 선언한 변수는 나중에 대입을 통해 초기화될 수 있다.
변수를 제어 흐름 구문 요소 같은 곳의 중간에서 초기화해야 할 때 유용하다.

```
let name;

if user.has_nickname() {
    name = user.nickname();
} else {
    name = generate_unique_name();
    user.register(&name);
}
```


- 일반적으로 많이 사용하는 문장이다.
- 똑같은 이름의 변수를 선언하면 이전에 존재하던 변수 값을 가려버리게 되는데, 이를 새도잉(Shadowing)이라 한다. 새로 선언된 변수의 정의는 자신이 속한 블록이 끝날 때까지 기존 변수의 정의를 대체한다.

```
let x = 10;  
let x = x + 2;  
let x = x + 3;  
  
println!("x = {x}");
```

```
for line in file.lines() {  
    let line = line?;  
    // ...  
}
```

- 일반적으로 많이 사용하는 문장이다.
- 블록은 아이템 선언(Item Declaration)도 포함할 수 있다.
아이템이란 fn, struct, use와 같이 프로그램이나 모듈 전역에 등장할 수 있는 선언을 말한다.

```
use std::cmp::Ordering;
use std::io;

fn show_files() -> io::Result<()> {
    let mut v = vec![];

    // ...

    fn cmp_by_timestamp_then_name(a: &FileInfo, b: &FileInfo) -> Ordering {
        a.timestamp
            .cmp(&b.timestamp)           // First, compare timestamps
            .reverse()                   // newest file first
            .then(a.path.cmp(&b.path))    // compare paths to break ties
    }

    v.sort_by(cmp_by_timestamp_then_name);

    // ...
}
```

if와 match

- if 표현식의 형태는 친숙하다.
 - C와 달리 조건을 괄호 안에 둘 필요가 없다. 오히려 Rust는 불필요한 괄호가 있으면 경고를 내보낸다.
 - 각 조건은 반드시 bool 타입의 표현식이어야 한다. (Rust는 암시적으로 수나 포인터를 bool 값으로 변환하지 않는다.)

```
let x = 10;

if x < 10 {
    println!("x is less than 10");
} else if x == 10 {
    println!("x is equal to 10");
} else {
    println!("x is greater than 10");
}
```

if와 match

- match 표현식은 C의 switch 문과 비슷하지만 좀 더 유연하다.

```
match code {  
    0 => println!("OK"),  
    1 => println!("Wires Tangled"),  
    2 => println!("User Asleep"),  
    _ => println!("Unrecognized Error {code}"),  
}
```

- switch 문처럼 code의 값에 따라 네 갈래의 코드 중 하나를 실행한다. 와일드카드 패턴 _는 모든 것을 매칭하는데, 반드시 맨 끝에 와야 한다는 점만 제외하면 switch 문의 default: 케이스와 비슷하다.
- 컴파일러는 C++의 switch 문처럼 점프 테이블을 써서 이런 종류의 match를 최적화할 수 있다.

if와 match

- match의 다재다능함은 각 갈래의 => 왼편에 쓸 수 있는 다양한 지원 패턴에서 비롯된다.

```
match params.get("name") {  
    Some(name) => println!("Hello, {name}!"),  
    None => println!("Greetings, stranger.")  
}
```

- Rust는 주어진 가능한 값을 모두 다루지 않는 match 표현식을 금지한다.

```
let score = match card.rank {  
    Jack => 10,  
    Queen => 10,  
    Ace => 11  
}; // Error: Non-exhaustive patterns
```

if와 match

- if 표현식의 모든 블록은 반드시 같은 타입의 값을 산출해야 한다.



```
let suggested_pet = if with_wings { Pet::Buzzard } else { Pet::Hyena }; // OK
let favorite_number = if user.is_hobbit() { "eleventy-one" } else { 9 }; // Error
let best_sports_team = if is_hockey_season() { "Predators" }; // Error
```

- 마찬가지로 match 표현식의 모든 갈래는 반드시 같은 타입을 가져야 한다.



```
let suggested_pet = match favorites.element {
    Fire => Pet::RedPanda,
    Air => Pet::Buffalo,
    Water => Pet::Orca,
    _ => None, // Error: Incompatible types
};
```

if와 match

- Rust에는 if의 형태가 한 가지 더 있는데, if let 표현식이 바로 그것이다.
- 주어진 표현식이 패턴과 매칭되면 첫번째 블록이 실행되고, 매칭되지 않으면 두번째 블록이 실행된다.
- 가끔 Option이나 Result에서 데이터를 꺼낼 때 쓰기 좋다.

```
if let Some(cookie) = request.session_cookie {  
    return restore_session(cookie);  
}  
  
if let Err(err) = show_cheesy_anti_robot_task() {  
    log_robot_attempt(err);  
    politely_accuse_user_of_being_a_robot();  
} else {  
    session.mark_as_human();  
}
```

- 루프 표현식의 종류는 4가지다.

```
while condition {  
    block  
}  
  
while let pattern = expr {  
    block  
}  
  
loop {  
    block  
}  
  
for pattern in iterable {  
    block  
}
```


- 루프 표현식의 종류는 4가지다.
 - `while` 루프의 동작은 C와 동일하다. 단, `condition`은 반드시 `bool` 타입이어야 한다.
 - `while let` 루프는 `if let`과 비슷하다. 루프가 매번 반복을 시작할 때 `expr`의 값이 주어진 `pattern`과 매칭되면 `block`이 실행되고, 그렇지 않으면 루프가 종료된다.
 - `loop`는 무한 루프를 작성할 때 쓴다.
무한 루프는 `block`을 영원히 (또는 `break`나 `return`을 만나거나 스레드가 패닉에 빠질 때까지) 반복 실행한다.
 - `for` 루프는 `iterable` 표현식을 평가한 뒤에 그 결과로 얻은 이터레이터의 개별 값에 대해서 한 번씩 `block`을 평가한다. (`Vec`, `HashMap` 등 많은 타입들이 이런 식으로 반복 처리될 수 있다.)

- .. 연산자는 범위(Range)를 산출한다.
 - 범위는 start와 end 두 개의 필드로 된 간단한 구조체다.
(0..20은 `std::ops::Range { start: 0, end: 20 }`과 같다.)
 - Range는 반복 처리할 수 있는 타입이므로 for 루프에서 쓸 수 있다.
(Range는 `std::iter::Iterator` 트레이트를 구현한다.)

- for 루프로 값을 반복 처리하기
 - 기본적으로는 Rust의 이동 문법에 따라 값이 소비된다.

```
fn main() {  
    let strings: Vec<String> = error_messages();  
  
    for s in strings {  
        // Each String is moved into s here...  
        println!("{s}");  
    } // ...and dropped here  
  
    println!("{}", error(s), strings.len()); // Error: Use of moved value  
}
```

- for 루프로 값을 반복 처리하기
 - 만약 값이 소비되는 걸 원하지 않는다면, 컬렉션의 레퍼런스를 반복 처리하도록 수정하면 된다.

```
fn main() {  
    let strings: Vec<String> = error_messages();  
  
    for rs in &strings {  
        println!("String {:?} is at address {:p}.", *rs, rs);  
    }  
  
    println!("{}", error(s)", strings.len());  
}
```

- for 루프로 값을 반복 처리하기
 - mut 레퍼런스를 반복 처리할 때는 각 요소의 mut 레퍼런스가 제공된다.

```
fn main() {  
    let strings: Vec<String> = error_messages();  
  
    // The type of rs is &mut String  
    for rs in &mut strings {  
        rs.push('\n'); // Add a newline to each string  
    }  
  
    println!("{}", error(s), strings.len());  
}
```

- `break` 표현식은 자기가 속한 루프를 빠져나간다.
`continue` 표현식은 루프의 다음 반복 처리로 건너 뛴다.
- `loop`의 본문에서는 `break`에 루프의 값을 산출하는 표현식을 줄 수 있다.

```
fn main() {  
    // Each call to `next_line` returns either `Some(line)`,  
    // where `line` is a line of input, or `None`,  
    // if we've reached the end of the input.  
    // Return the first line that starts with "answer: ".  
    // Otherwise, return "answer: nothing".  
    let answer = loop {  
        if let Some(line) = next_line() {  
            if line.starts_with("answer: ") {  
                break line;  
            }  
        } else {  
            break "answer: nothing";  
        }  
    };  
}
```

- 루프는 수명과 함께 레이블(Label)을 가질 수 있다. 중첩 루프를 빠져나갈 때 유용하다.



```
'search: for room in apartment {  
    for spot in room.hiding_spots() {  
        if spot.contains(keys) {  
            println!("Your keys are {spot} in the {room}.");  
            break 'search;  
        }  
    }  
}
```

- `break`는 레이블과 값 표현식 모두 가질 수 있다.

```

// Find the square root of the first perfect square in the series.
let sqrt = 'outer: loop {
    let n = next_number();

    for i in 1.. {
        let square = i * i;

        if square == n {
            // Found a square root.
            break 'outer i;
        }

        if square > n {
            // `n` isn't a perfect square, try the next
            break;
        }
    }
};
```


return 표현식

- return 표현식은 현재 함수를 빠져나와 호출부에 값을 반환한다.
값이 없는 return은 return ()의 축약 표기다.



```
fn f() {    // Return type omitted: Defaults to ()  
    return; // Return value omitted: Defaults to ()  
}
```

Rust에 loop가 있는 이유

- Rust 컴파일러가 여러 측면에서 프로그램 전반의 제어 흐름을 분석하는 방식
 - 함수의 모든 경로가 예정된 리턴 타입의 값을 반환하는지 검사한다.
이를 제대로 수행하기 위해서는 함수의 끝에 도달하는 것이 가능한지 아닌지 알아야 한다.
 - 지역 변수가 초기화되지 않은 채로 쓰이는 일이 없는지 검사한다. 여기에는 함수의 모든 경로를 검사해서 초기화 코드를 거치지 않고서는 변수가 쓰이는 곳에 도달할 길이 없음을 확인하는 작업이 수반된다.
 - 도달할 수 없는 코드에 대해서 경고를 내보낸다.
함수에서 어떤 코드가 자신에게 이르는 경로를 전혀 갖지 못하면 그 코드는 도달할 수 없는 코드다.
- 이를 가리켜 흐름을 고려한(Flow-sensitive) 분석이라고 한다.

Rust에 loop가 있는 이유

- 다음 코드를 살펴 보자.
 - 해당 함수는 return 문을 통해서만 종료되기 때문에 while 루프가 i32를 산출하지 않는다는 건 사실이 아니다.
 - Rust의 타입 시스템도 제어 흐름의 영향을 받는다. 하지만 break나 return 표현식, 무한 루프인 loop, panic!(), std::process::exit() 호출로 끝나는 블록에 이 규칙을 적용하는 건 아무래도 좀 이상하다.
 - 따라서 Rust에서는 이들 표현식이 평범한 타입을 갖지 않는다. 정상적으로 끝나지 않는 표현식에서는 !라고 하는 특수한 타입이 지정되고 타입 일치에 관한 규칙에서 면제된다. std::process::exit()의 함수 시그니처를 보면 리턴 타입이 !임을 알 수 있는데, 이런 함수를 일탈 함수(Divergent Function)라고 한다.

```
fn wait_for_process(process: &mut Process) -> i32 {  
    while true {  
        if process.wait() {  
            return process.exit_code();  
        }  
    }  
} // Error: Mismatched types: expected i32, found ()
```

함수와 메소드 호출

- Rust는 함수 호출 문법과 메소드 호출 문법이 같다.



```
let x = gcd(1302, 462);      // Function call  
let room = player.location(); // Method call
```

- 구조체나 튜플의 필드에 접근할 때는 `.`을 쓴다.

```
game.black_pawns // Struct field  
coords.1         // Tuple element
```

- 배열, 슬라이스, 벡터의 요소에 접근할 때는 대괄호를 쓴다.

```
pieces[i]         // Array element
```

- 이 세 가지 표현식은 대입문 왼쪽에 올 수 있다고 해서 **L값(Lvalue)**이라고 부른다.

```
game.black_pawns = 0x00ff0000_00000000_u64;  
coords.1 = 0;  
pieces[2] = Some(Piece::new(Black, Knight, coords));
```

- 배열이나 벡터에서 슬라이스를 추출하는 일은 간단하다.
- 여기서 `game_moves`는 배열일 수도 있고, 슬라이스일 수도 있고, 벡터일 수도 있다.
하지만 결과는 이와 무관하게 항상 `end - midpoint` 길이의 차용된 슬라이스다.



```
let second_half = &game_moves[midpoint .. end];
```

- .. 연산자의 양옆에 오는 피연산자들은 생략할 수 있다.

그 결과 피연산자 유무에 따라 최대 4가지 타입의 서로 다른 오브젝트가 만들어진다.

- 아래쪽에 있는 두 가지 형태는 끝을 포함하지 않는 범위(End-exclusive Range)다.
이를 반개구간(Half-open Interval)이라고도 하는데, 산출되는 범위에 끝 값이 포함되지 않는다.

```
..           // RangeFull
a ..         // RangeFrom { start: a }
.. b        // RangeTo { end: b }
a .. b      // Range { start: a, end: b }
```

- ..= 연산자는 끝을 포함하는 범위(End-inclusive Range)를 산출한다.
이를 폐구간(Closed Interval)이라고도 하는데, 산출되는 범위에 끝 값이 포함된다.

```
..= b       // RangeToInclusive { end: b }
a ..= b     // RangeInclusive::new(a, b)
```

- 주소 연산자 &와 &mut는 이미 다뤘었다.
- 단항 연산자 *는 레퍼런스가 가리키는 값에 접근하기 위한 용도로 쓰인다.
- Rust는 . 연산자로 필드나 메소드를 접근할 때 레퍼런스를 자동으로 따라가기 때문에, * 연산자는 레퍼런스가 가리키는 값 자체를 읽거나 쓰고 싶을 때만 필요하다.



```
let padovan: Vec<u64> = compute_padovan_sequence(n);

for elem in &padovan {
    draw_triangle(turtle, *elem);
}
```


산술, 비트별, 비교, 논리 연산자

- Rust의 이항 연산자는 대부분 다른 언어들과 비슷하다.

여기서는 Rust가 기존의 틀에서 벗어나 다르게 가져가는 몇 가지만 살펴 보자.

- 디버그 빌드에서 정수 오버플로가 탐지되면 패닉에 빠진다. 이를 위해 다양한 메소드를 제공한다. (이미 살펴봤었다.)
- 정수 나눗셈은 0에 가까운 쪽으로 반올림하며, 정수를 0으로 나누면 Release 빌드에서도 패닉에 빠진다.
- $a \% b$ 는 0에 가까운 쪽으로 반올림하는 나눗셈의 부호 있는 나머지를 계산한다.
- 비트별 논리 부정 연산자로 ~가 아니라 !를 쓴다.
- 비트별 연산은 C언어와 달리 비교 연산보다 우선 순위가 높아서,
 $x \& \text{BIT} != 0$ 이라고 쓰면 본래 의도대로 $(x \& \text{BIT}) != 0$ 과 같은 뜻이 된다.

- = 연산자는 `mut` 변수와 그가 가진 필드 또는 요소에 뭔가를 대입하기 위한 용도로 쓰인다.
- `Copy`가 아닌 타입의 값을 대입하는 경우에는 값이 목적지로 이동된다.
이때 값의 소유권은 원래 주인에게서 새 주인에게로 넘어가며, 새 주인이 쥐고 있던 이전 값은 드롭된다.
- Rust는 복합 대입을 지원한다.
`total += item.price`는 `total = total + item.price`와 동일하다.
- Rust는 C언어와 달리 연쇄 대입을 지원하지 않는다.
즉, `a`와 `b`에 3을 대입하기 위해서 `a = b = 3`이라고 쓸 수 없다.
- Rust에는 C언어의 증가 감소 연산자인 `++`과 `--`가 없다.

- Rust에서는 한 타입의 값을 다른 타입으로 변환할 때 보통 명시적인 캐스팅을 요구한다. 캐스팅은 `as` 키워드를 써서 한다.

```
let x = 17; // x is type i32
let index = x as usize; // Convert to usize
```

- 허용되는 캐스팅의 종류는 다음과 같다.
 - 수는 기본 제공 수치 타입의 범주 안에서 자유롭게 캐스팅할 수 있다.
 - `bool`, `char`, C 스타일 `enum` 타입의 값은 어떤 정수 타입으로든 캐스팅할 수 있다.
 - 안전하지 않은 포인터 타입과 관련된 캐스팅도 일부 허용되어 있다. (이 부분은 나중에 살펴 볼 예정이다.)

- Rust의 타입 자동 변환
 - 앞에서 변환이 보통 캐스팅을 요구한다고 말한 바 있다.
그러나 레퍼런스 타입과 관련된 변환 몇 가지는 아주 간단해서 캐스팅하지 않아도 언어가 변환을 수행해 준다.
 - 좀 더 중요한 자동 변환으로는 다음과 같은 것들이 있다.
 - `&String` 타입의 값은 캐스팅 없이 `&str` 타입으로 자동 변환된다.
 - `&Vec<i32>` 타입의 값은 `&[i32]`로 자동 변환된다.
 - `&Box<Chessboard>` 타입의 값은 `&Chessboard`로 자동 변환된다.
 - 이 변환들은 기본 제공 트레이트 `Deref`를 구현하고 있는 타입에 적용된다고 해서 **Deref 강제 변환(Deref Coercion)**이라고 한다. `Deref` 강제 변환의 목적은 `Box` 같은 스마트 포인터 타입을 최대한 실제 값처럼 행동하도록 만드는 것이다.

- Rust는 간단한 함수처럼 생긴 값인 클로저(Closure)를 가지고 있다.

```
let is_even = |x| x % 2 == 0;
```

- Rust는 인수 타입과 리턴 타입을 추론한다. 하지만 필요하다면 함수의 경우처럼 명시적으로 적어줄 수도 있다.
리턴 타입을 지정할 때는 클로저 본문을 블록 안에 두어야 구문 오류를 피할 수 있다.

```
let is_even = |x: u64| -> bool x % 2 == 0; // Error
let is_even = |x: u64| -> bool { x % 2 == 0 }; // OK
```

- 클로저에 대한 이야기는 뒤에서 자세하게 다룰 예정이다.

감사합니다.

utilForever@gmail.com

<https://github.com/utilForever>

Facebook, Twitter: @utilForever