고급 기능들

2020.4

안전하지 않은 러스트

- 컴파일러가 지나치게 보수적이라 보완책으로 씀.
- 컴퓨터 하드웨어 자체가 안전한 코드로만 사용이 불가능.
- 운영체제와 직접 상호작용하거나 운영체제 자체를 만들 때 안전한 러스트만 가지고 개발이 불가능

Unsafe

- 다음의 4가지 기능을 제공함

- 로우 포인터 (raw pointer) 를 역참조하기
- 안전하지 않은 함수 혹은 메소드 호출하기
- 가변 정적 변수 (mutable static variable) 의 접근 혹은 수정하기
- 안전하지 않은 트레잇 구현하기

- 위 4가지 외에는 unsafe를 사용해도 러스트 컴파일러는 여전히 안전성 체크를 수행함
- unsafe를 안전한 코드로 감싸서 다른 안전한 코드에서 사용하는 방식이 좋음.

로우 포인터 역참조

- *const T, *mut T의 두개 타입을 제공한다.
- 여기서 *는 역참조 연산자가 아니고 타입이름의 일부이다.
- 특성
  - 빌림 규칙 무시 : 불변 가변 여러개를 가질 수 있다.
  - 유효한 포인터임을 보장하지 않음 : 댕글링 포인터 일수 있다.
  - 널 값을 가질 수 있다.
  - 자동으로 메모리 해제가 구현되지 않는다.

로우 포인터

```
let mut num = 5;

let r1 = &num as *const i32;  // unsafe가 사용되지 않았음
let r2 = &mut num as *mut i32; // 안전하게 생성가능


let address = 0x012345usize;  // 이 주소가 유효한지는 보장되지 않음
let r = address as *const i32;
```

```
let mut num = 5;

let r1 = &num as *const i32; // 포인터의 생성 만으로 해를 끼치지는 않음
let r2 = &mut num as *mut i32;  // 역참조 할 때 위험할 수 있음

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

안전하지 않은 함수

```
unsafe fn dangerous() {}

unsafe {
    dangerous();
}
```

```
let mut v = vec![1, 2, 3, 4, 5, 6];

let r = &mut v[..];

let (a, b) = r.split_at_mut(3);

assert_eq!(a, &mut [1, 2, 3]);
assert_eq!(b, &mut [4, 5, 6]);
```

```
fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();

    assert!(mid <= len);

    (&mut slice[..mid],
     &mut slice[mid..])
}
```

```
use std::slice;

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {
    let len = slice.len();
    let ptr = slice.as_mut_ptr();

    assert!(mid <= len);

    unsafe {
        (slice::from_raw_parts_mut(ptr, mid),
         slice::from_raw_parts_mut(ptr.offset(mid as isize), len - mid))
    }
}
```

```
error[E0499]: cannot borrow `*slice` as mutable more than once at a time
 -->
  |
6 |     (&mut slice[..mid],
  |                -----
  |                first mutable borrow occurs here
7 |      &mut slice[mid..])
  |             ^^^^^
  |             second mutable borrow occurs here
8 | }
  | - first borrow ends here
```

Extern

```
extern "C" {
    fn abs(input: i32) -> i32;
}

fn main() {
    unsafe { // C 언어의 안전성을 러스트가 검사할 방법이 없다.
        println!("Absolute value of -3 according to C: {}", abs(-3));
    }
}


#[no_mangle]
pub extern "C" fn call_from_c() {
    println!("Just called a Rust function from C!");
} // unsafe가 필요없다.
```

정적 변수

```
static HELLO_WORLD: &str = "Hello, world!";

fn main() {
    println!("name is: {}", HELLO_WORLD);
} // 읽기만 가능하므로 레이스 상황이 발생 안함.
```

상수와 다른점은 정적 변수는 메모리 내의 고정된 위치에 저장되고
상수는 컴파일러에 의해 대치됨.

```
unsafe trait Foo {
    // methods go here
}

unsafe impl Foo for i32 {
    // method implementations go here
}  // 안전하지 않은 트레잇
```

Send, Sync가 충족되지 않는 타입을 Send, Sync
로 정의하려면 unsafe를 사용한다.
즉 사람 책임이다.

```
static mut COUNTER: u32 = 0;

fn add_to_count(inc: u32) {
    unsafe {
        COUNTER += inc;
    }
}

fn main() {
    add_to_count(3);

    unsafe {
        println!("COUNTER: {}", COUNTER);
    }
}. // 데이터 레이스 방지는 사람의 책임.
```

Lifetime sub typing

하나의 라이프타임이 다른거 보다 같거나 더 오래 사는 것 보장

```
struct Context(&str);

struct Parser {
    context: &Context,
}

impl Parser {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
}

struct Context<'a>(&'a str);

struct Parser<'a> {
    context: &'a Context<'a>,
}

impl<'a> Parser<'a> {
    fn parse(&self) -> Result<(), &str> {
        Err(&self.context.0[1..])
    }
} // 컴파일은 된다.

fn parse_context(context: Context)
-> Result<(), &str> {
    Parser { context: &context }.parse()
} // 이 함수를 추가하면 옆의 에러가 발생한다.
```

```
error[E0597]: borrowed value does not live long enough
  --> src/lib.rs:14:5
   |
14 |     Parser { context: &context }.parse()
   |     ^^^^^^^^^^^^^^^^^^^^^^^^^^^^ does not live long enough
15 | }
   | - temporary value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the function body at 13:1...
  --> src/lib.rs:13:1
   |
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
   | |_^

error[E0597]: `context` does not live long enough
  --> src/lib.rs:14:24
   |
14 |     Parser { context: &context }.parse()
   |                       ^^^^^^^ does not live long enough
15 | }
   | - borrowed value only lives until here
   |
note: borrowed value must be valid for the anonymous lifetime #1 defined on the function body at 13:1...
  --> src/lib.rs:13:1
   |
13 | / fn parse_context(context: Context) -> Result<(), &str> {
14 | |     Parser { context: &context }.parse()
15 | | }
   | |_^
```

```rust
struct Context<'s>(&'s str);

struct Parser<'c, 's> {
    context: &'c Context<'s>,
}

impl<'c, 's> Parser<'c, 's> {
    fn parse(&self) -> Result<(), &'s str> {
        Err(&self.context.0[1..])
    }
}

fn parse_context(context: Context) -> Result<(), &str> {
    Parser { context: &context }.parse()
}
```

```rust
fn parse<'a>(&'a self) -> Result<(), &'a str> {
```

```rust
struct Parser<'c, 's: 'c> {
    context: &'c Context<'s>,
}
```

```
error[E0491]: in type `&'c Context<'s>`, reference has a longer lifetime than the data it references
 --> src/lib.rs:4:5
  |
4 |     context: &'c Context<'s>,
  |     ^^^^^^^^^^^^^^^^^^^^^^^^
  |
note: the pointer is valid for the lifetime 'c as defined on the struct at 3:1
 --> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_^
note: but the referenced data is only valid for the lifetime 's as defined on the struct at 3:1
 --> src/lib.rs:3:1
  |
3 | / struct Parser<'c, 's> {
4 | |     context: &'c Context<'s>,
5 | | }
  | |_^
```

라이프타임 바운드

```
struct Ref<'a, T>(&'a T);

error[E0309]: the parameter type `T` may not live long enough
 --> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |                   ^^^^^^
  |
  = help: consider adding an explicit lifetime bound `T: 'a`...
note: ...so that the reference type `&'a T` does not outlive the data it points at
 --> src/lib.rs:1:19
  |
1 | struct Ref<'a, T>(&'a T);
  |                   ^^^^^^
```

```
struct Ref<'a, T: 'a>(&'a T);
// T 내의 어떠한 참조자들도 최소한 'a만큼 오래 살 것임을 명시하기
```

트레잇 객체의 경우

```
trait Red { }

struct Ball<'a> {
    diameter: &'a i32,
}

impl<'a> Red for Ball<'a> { }

fn main() {
    let num = 5;

    let obj = Box::new(Ball { diameter: &num }) as Box<Red>;
} // 아래의 규칙 때문에 잘 컴파일 된다.
```

- 트레잇 객체의 기본 라이프타임은 `'static` 입니다.
- `&'a Trait` 혹은 `&'a mut Trait`을 쓴 경우, 트레잇 객체의 기본 라이프타임은 `'a` 입니다.
- 단일 `T: 'a` 구절을 쓴 경우, 트레잇 객체의 기본 라이프타임은 `'a` 입니다.
- 여러 개의 `T: 'a` 같은 구절들을 쓴 경우, 기본 라이프타임는 없습니다; 우리가 명시적으로 써야합니다.

연관 타입, 기본 타입

```
pub trait Iterator {
    type Item;
    fn next(&mut self) -> Option<Self::Item>;
}

impl Iterator for Counter {
    type Item = u32;

    fn next(&mut self) -> Option<Self::Item> {
        // —snip--
```

```
pub trait Iterator<T> {
    fn next(&mut self) -> Option<T>;
}

Impl Iterator<String> for Counter
Impl Iterator<i32> for Counter
Impl Iterator<f64> for Counter
```

```
use std::ops::Add;

#[derive(Debug, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}

impl Add for Point { // 기본 타입인 경우
    type Output = Point;

    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}

fn main() {
    assert_eq!(Point { x: 1, y: 0 } + Point { x: 2, y: 3 },
               Point { x: 3, y: 3 });
}

trait Add<RHS=Self> {. // 기본 타입이 Self
    type Output;

    fn add(self, rhs: RHS) -> Self::Output;
}
```

```
use std::ops::Add;

struct Millimeters(u32);
struct Meters(u32);

impl Add<Meters> for Millimeters {
    type Output = Millimeters;

    fn add(self, other: Meters) -> Millimeters {
        Millimeters(self.0 + (other.0 * 1000))
    }
} // 기본 타입이 아닌 경우
```

모호성 방지

```
trait Pilot {
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking.");
    }
}

impl Wizard for Human {
    fn fly(&self) {
        println!("Up!");
    }
}

impl Human {
    fn fly(&self) {
        println!("*waving arms furiously*");
    }
}
```

```
fn main() {
    let person = Human;
    person.fly();
}
```

```
fn main() {
    let person = Human;
    Pilot::fly(&person);
    Wizard::fly(&person);
    person.fly();
}
```

```
trait Animal {
    fn baby_name() -> String;
}

struct Dog;

impl Dog {
    fn baby_name() -> String {
        String::from("Spot")
    }
}

impl Animal for Dog {
    fn baby_name() -> String {
        String::from("puppy")
    }
}

fn main() {
    println!("A baby dog is called a {}", Dog::baby_name());
} // self가 없는 연관 함수인 경우.

fn main() {
    println!("A baby dog is called a {}", Animal::baby_name());
} // error


fn main() {
    println!("A baby dog is called a {}", <Dog as Animal>::baby_name());
}
```

Super trait

```
**********
*        *
*  (1, 3) *
*        *
**********
```

```
use std::fmt;

trait OutlinePrint: fmt::Display {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len + 4));
        println!("*{}*", " ".repeat(len + 2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len + 2));
        println!("{}", "*".repeat(len + 4));
    }
}

struct Point {
    x: i32,
    y: i32,
}

impl OutlinePrint for Point {}
```

```
error[E0277]: the trait bound `Point: std::fmt::Display` is not satisfied
  --> src/main.rs:20:6
   |
20 | impl OutlinePrint for Point {}
   |      ^^^^^^^^^^^^ `Point` cannot be formatted with the default formatter;
try using `:?` instead if you are using a format string
   |
   = help: the trait `std::fmt::Display` is not implemented for `Point`
```

```
use std::fmt;

impl fmt::Display for Point {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.x, self.y)
    }
}
```

New type pattern

```
use std::fmt;

struct Wrapper(Vec<String>);

impl fmt::Display for Wrapper {
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "[{}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(vec![String::from("hello"), String::from("world")]);
    println!("w = {}", w);
}
```

트레잇 구현 외에도,
타입의 제약 강화,
내부 구현사항 숨기기 등의 역할을 한다.

타입 별칭

```
type Kilometers = i32;

let x: i32 = 5;
let y: Kilometers = 5;

println!("x + y = {}", x + y); // 가능하다.
```

뉴타입 패턴처럼 타입 제약 및 검사의 이점이 없다.

```
use std::io::Error;
use std::fmt;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize, Error>;
    fn flush(&mut self) -> Result<(), Error>;

    fn write_all(&mut self, buf: &[u8]) -> Result<(), Error>;
    fn write_fmt(&mut self, fmt: fmt::Arguments) -> Result<(), Error>;
}

type Result<T> = Result<T, std::io::Error>;

pub trait Write {
    fn write(&mut self, buf: &[u8]) -> Result<usize>;
    fn flush(&mut self) -> Result<()>;

    fn write_all(&mut self, buf: &[u8]) -> Result<()>;
    fn write_fmt(&mut self, fmt: Arguments) -> Result<()>;
}
```

! 타입

```
let guess: u32 = match guess.trim().parse() {
    Ok(num) => num,
    Err(_) => continue,
};

let guess = match guess.trim().parse() {
    Ok(_) => 5,
    Err(_) => “hello",  // &str ? , i32 ?
}

impl<T> Option<T> {
    pub fn unwrap(self) -> T {
        match self {
            Some(val) => val,
            None => panic!("called `Option::unwrap()` on a `None` value"),
        }
    }
}

loop {
    print!("and ever ");
}
```

DST(DYNAMICALLY SIZED TYPE)

```
let s1: str = "Hello there!";
let s2: str = "How's it going?”;
```

// str type의 크기는 얼마인가?
그때 그때 다르므로 이런 타입의
변수 선언은 불가능하다.
그래서 &str로 사용한다.

```
fn generic<T>(t: T) {
    // --snip--
}
```

는 자동으로

```
fn generic<T: Sized>(t: T) {
    // --snip--
}
```

로 인식된다.

함수 포인터

```
fn add_one(x: i32) -> i32 {
    x + 1
}

fn do_twice(f: fn(i32) -> i32, arg: i32) -> i32 {
    f(arg) + f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);

    println!("The answer is: {}", answer);
}
```

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(|i| i.to_string())
    .collect();
```

```
let list_of_numbers = vec![1, 2, 3];
let list_of_strings: Vec<String> = list_of_numbers
    .iter()
    .map(ToString::to_string)
    .collect();
```

함수 포인터는 클로저 트레잇 세 종류 (`Fn`, `FnMut`, 그리고 `FnOnce`) 모두를 구현하므로,
우리는 언제나 클로저를 인자로서 기대하는 함수에게 함수 포인터를 넘길 수 있습니다.

클로저 반환하기

```
fn returns_closure() -> Fn(i32) -> i32 {
    |x| x + 1
}
```

```
error[E0277]: the trait bound `std::ops::Fn(i32) -> i32 + 'static:
std::marker::Sized` is not satisfied
 -->
  |
1 | fn returns_closure() -> Fn(i32) -> i32 {
  |                         ^^^^^^^^^^^^^^ `std::ops::Fn(i32) -> i32 + 'static`
 does not have a constant size known at compile-time
  |
  = help: the trait `std::marker::Sized` is not implemented for
 `std::ops::Fn(i32) -> i32 + 'static`
  = note: the return type of a function must have a statically known size
```

```
fn returns_closure() -> Box<Fn(i32) -> i32> {
    Box::new(|x| x + 1)
}
```

Q & A