# 組み込み Rust 講習会

## Rust 編

JIJINBEI

2024-08-02

HiCoder

# Table of contents

# Table of contents

# はじめに

# はじめに

組み込み Rust 講習会へようこそ

この講習会を開いた理由

- 組み込み Rust が難しくて私の学習が進まないので、一緒に学べる仲間を作りたい
- Rust の普及
- ソフト、競プロだけをやるな！！ハードもやれ！！

# 目標

# 目標

- Rust の基本的な文法を理解する
- 難解な所有権、ライフタイムの概念を理解する
- 「基礎からわかる組み込み Rust」や組み込み Rust の公式ドキュメントを難なく読めるようになる

# Rust とは

# Rust とは

- Mozilla が開発したプログラミング言語
- Rust は最も愛されている言語として 7 年目を迎え、87% の開発者が使い続けたいと答えている。 [link]
- 最近のオープンソースはすべて Rust で書かれていると言っても過言ではない[1]

[1] このスライドも Rust 製の Typst で作成

# Rust が好まれている理由

- 安全性
  - 型安全性
  - メモリ安全性
    - **所有権**
    - **ライフタイム**
- 処理速度の速さ
- 並行処理
  - 所有権によるデータ競合の防止
- バージョンやパッケージ管理
  - cargo
- テストがしやすい

# 変数

# 変数

`let` で変数の定義

```rust
fn main() {
    let x = 5;
    println!("The value of x is: {}", x);
    x = 6; // ERROR
    println!("The value of x is: {}", x);
}
```

変数を可変するには、`mut` を使う

```rust
  fn main() {
      let mut x = 5;
      println!("The value of x is: {}", x);
      x = 6;
      println!("The value of x is: {}", x);
  }
```

# データ型

# データ型(数値)

- Rust は静的型付け言語

| 大きさ | 符号付き | 符号なし |
|---|---|---|
| 8-bit | i8 | u8 |
| 16-bit | i16 | u16 |
| 32-bit | i32 | u32 |
| 64-bit | i64 | u64 |
| arch | isize | usize |

表 1: 整数型

| 大きさ | 浮動小数点 |
|---|---|
| 32-bit | f32 |
| 64-bit | f64 |

表 2: 浮動小数点型

ほかには `char` 型(`a`, `b`, `c` …)や `bool` 型(`true`, `false`)などがある

# 配列

# 配列

- 配列は同じ型の要素を持つ

```rust
fn main() {
    let a: [i32; 5] = [1, 2, 3, 4, 5];
}
```

`i32` が 5 つの要素を持つ配列 `a` を定義で配列の長さは変更できない

# 配列

文字列は文字(char)の配列

文字列は、主に &str 型と String 型がある

```rust
fn main() {
    let world: &str = "world!";
    println!("Hello, {}", world);
}
```

```rust
fn main() {
    let mut world: String = String::from("world");
    world.push_str("!");
    println!("Hello, {}", world);
}
```

String 型は配列の長さを変更できる

String はデータの書き込み、&str はデータの読み込みを行うときに使うとよい

# 関数、制御フロー

# 関数、制御フロー

省略

# 構造体とメソット

# 構造体とメソット

```rust
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle { width: 30, height: 50 };

    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

# 構造体とメソット

関連するデータと機能を 1 つの単位にまとめることで、コードの体系化と再利用性が向上する

new メソットで構造体を生成する書きかたが一般的

```rust
impl Rectangle {
    fn new(width: u32, height: u32) -> Self {
        Rectangle { width, height }
    }
}

fn main() {
    let rect1 = Rectangle::new(30, 50);
}
```

# ジェネリック

# ジェネリック

プログラムを抽象化すると、コードの再利用性が高まり、可読性が向上する

```rust
struct Point<T> {
    x: T,
    y: T,
}

fn main() {
    let integer = Point { x: 5, y: 10 };
    let float = Point { x: 1.0, y: 4.0 };
}
```

ジェネリック<T>を用いることで、i32 型や f64 型などの複数型を受け取ることができる[1]

[1]ジェネリックが用いられている標準ライブラリの例: Option<T>, Result<T, E>

この型はエラーを扱うときに用いられる型

# トレイト

# トレイト

トレイトは、複数の型で共有される振る舞い（メソッド）を定義するインターフェースのような機能

以下の例では、`Summary` トレイトを定義し、`NewsArticle` と `Tweet` 構造体に `Summary` トレイトを実装している

```rust
pub trait Summary {
    fn summarize(&self) -> String;
}

pub struct NewsArticle {
    pub headline: String,
    pub location: String,
    pub author: String,
    pub content: String,
}

impl Summary for NewsArticle {
    fn summarize(&self) -> String {
```

# トレイト

```
14          format!("{}, by {} ({})", self.headline, self.author,
    self.location)
15      }
16  }
17
18  pub struct Tweet {
19      pub username: String,
20      pub content: String,
21      pub reply: bool,
22      pub retweet: bool,
23  }
24
25  impl Summary for Tweet {
26      fn summarize(&self) -> String {
27          format!("{}: {}", self.username, self.content)
28      }
29  }
30
31  fn main() {
32      let tweet = Tweet {
33          username: String::from("horse_ebooks"),
34          content: String::from(
```

# トレイト

```
35              "of course, as you probably already know, people",
36          ),
37          reply: false,
38          retweet: false,
39      };
40
41      println!("1 new tweet: {}", tweet.summarize());
42  }
```

# 所有権とライフタイム

# 所有権の不便な例

Rust のキモいところ

```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;

    // println!("{}, world!", s1); // ERROR
    println!("{}, world!", s2);
}
```

```rust
fn main() {
    let s = String::from("hello");
    takes_ownership(s);
    // println!("{}", s); // ERROR
}

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}
```

# 所有権のルール

公式ドキュメントにかかれている所有権規則

- **各値は、所有者と呼ばれる変数と対応している。**
- **いかなる時も所有者は一つである。**
- **所有者がスコープから外れたら、値は破棄される。**
- **変数をアサインする(`let x = y`)際や、関数に引数を値渡しする(`foo(x)`)際は、所有権が移動（move）**

参照規則

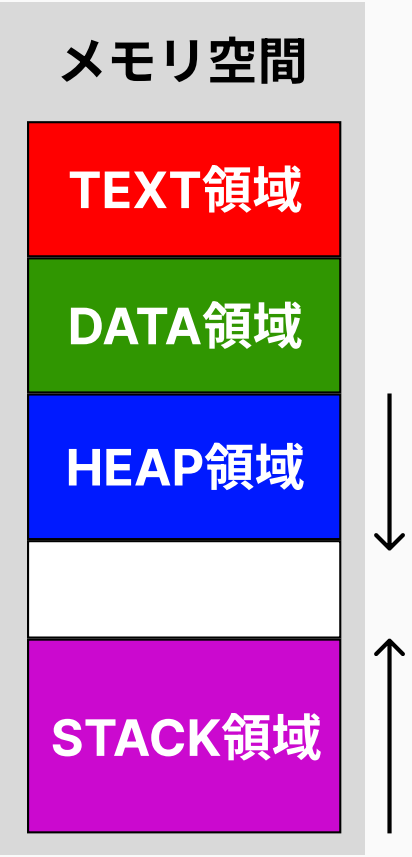
- **任意のタイミングで、一つの可変参照か不変な参照いくつでものどちらかを行える。**
- **参照は常に有効でなければならない。**
  - **同義) 生存期間の長いものが、短いものを参照してはいけない**

？？？？？？？？？

# ヒープとスタック

所有権を理解するためには、ヒープとスタックを理解する必要がある

- TEXT 領域
  - 機械語に翻訳されたプログラム
- DATA 領域
  - 初期値
    - global 変数, 静的変数
- **HEAP 領域**
  - 寿命がある値（可変長）
    - `malloc()`でヒープ領域を確保し、`free()`でメモリの開放
- **STACK 領域**
  - stack(下から積み上げ)をしていく寿命がある値（固定長）
    - 関数の引数、ローカル変数
    - 配列の場合、ヒープ領域を参照するポインタ



# バイナリを見る

Rust のバイナリを見てよう

```terminal
$ objdump -S binary_file
$ objdump -h binary_file
```

`.text` や `.rodata` や `.bss`(heap に対応？)がある

# 所有権のイメージ

所有権は、**変数にメモリの解放をする責任を持たせる**

知っておくこと

- **値と変数の意味は異なる**
  - 値は具体的なデータで、変数は値の名前
- スコープの領域が展開されると、スタックは下から積み上げられ、ヒープで値を確保される
- 所有権を持っている変数たちは、スコープの外で、ヒープのメモリは解放され、スタックはポップされる
- & で値を参照したら返さないといけない(**借用**)

イラストで理解しよう！！

# 文字列の実態

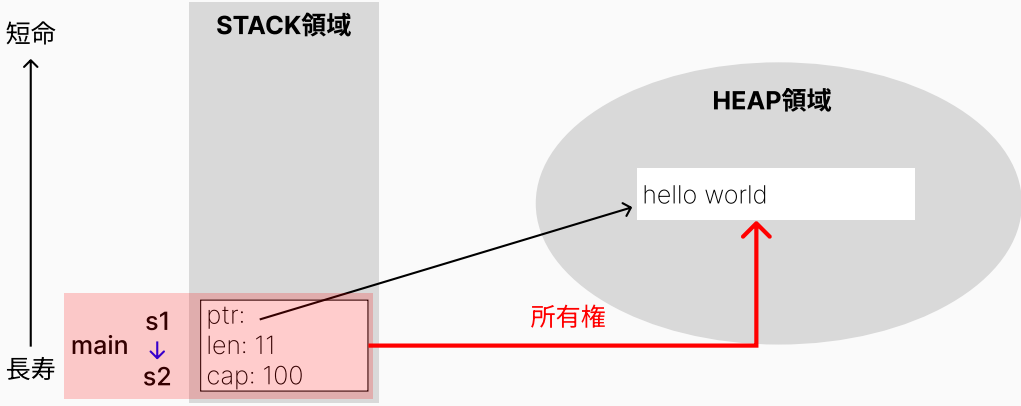
`String` の実態はポインタ、長さ、容量の 3 つの要素とヒープ上の文字列（値）

`&str` の実態はポインタと長さの 2 つの要素とヒープ上の文字列

⇒ スタック領域のデータは固定長に落ちている

# 所有権のイメージ(値と変数)

```rust
fn main() {
    let s1 = String::from("hello world");
    let s2 = s1;
}
```



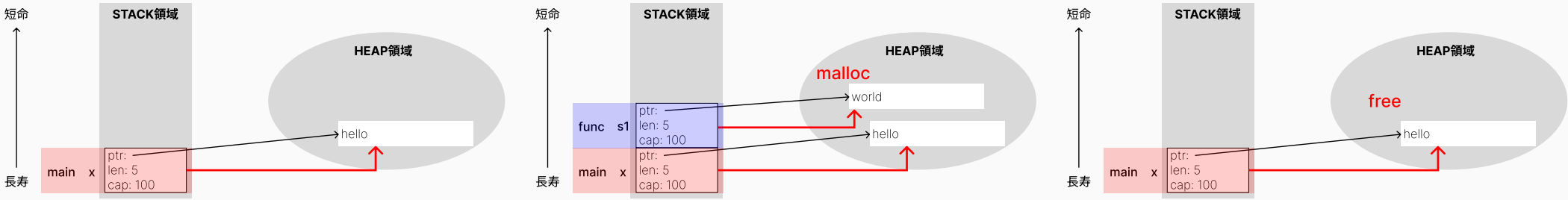

s1 と s2 は、同じ値を指している

s2 が s1 の後に使えないのは、値を持っている所有権が s2 に移動しているから

# 所有権のイメージ(スコープ 1)

```rust
fn main() {
    let x = String::from("hello");
    func();
}

fn func() {
    let s1 = String::from("world");
}
```

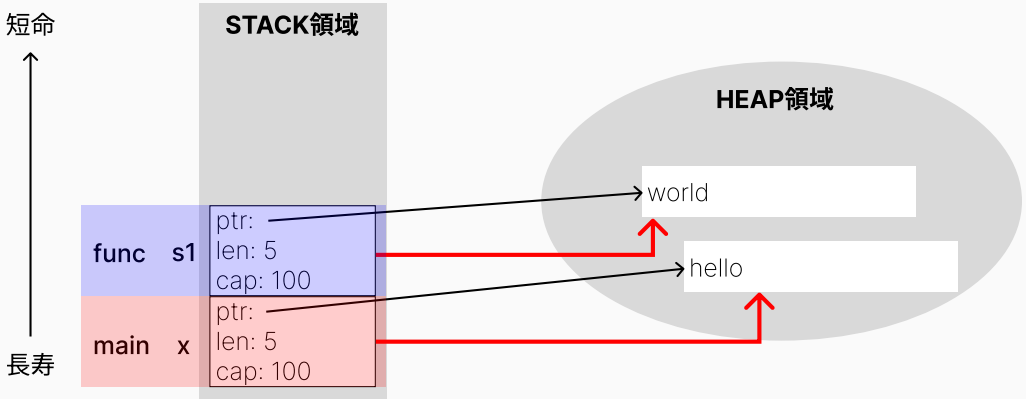


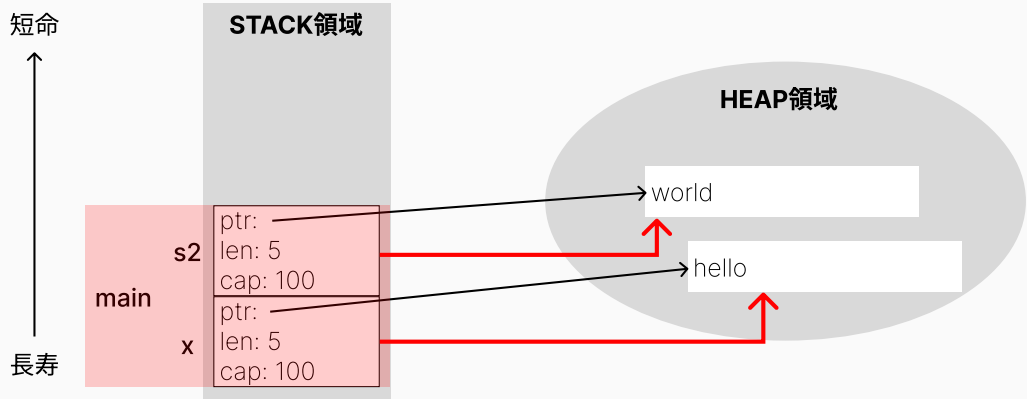

func のスコープの外で所有権を持っている s1 が破棄される

# 所有権のイメージ(スコープ 2)

```rust
fn main() {
    let x = String::from("hello");
    let s2 = func();
}

fn func() -> String {
    let s1 = String::from("world");
    s1
}
```

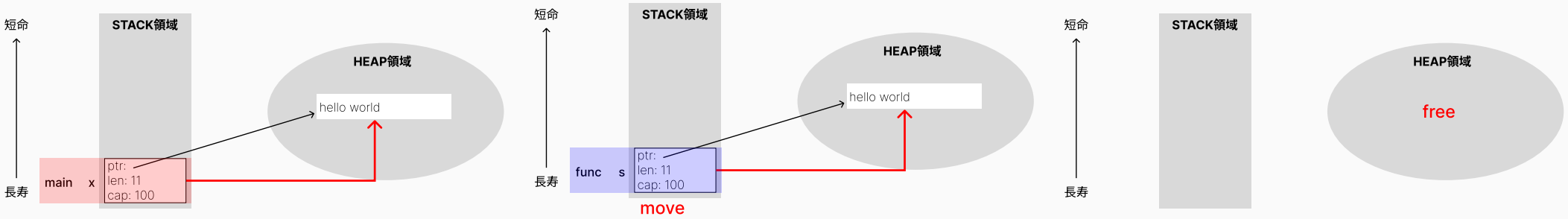


# 所有権のイメージ(所有権の移動)

```rust
fn main() {
    let x = String::from("hello world");
    func(x);
}

fn func(s: String) {
    println!("{}", s);
}
```



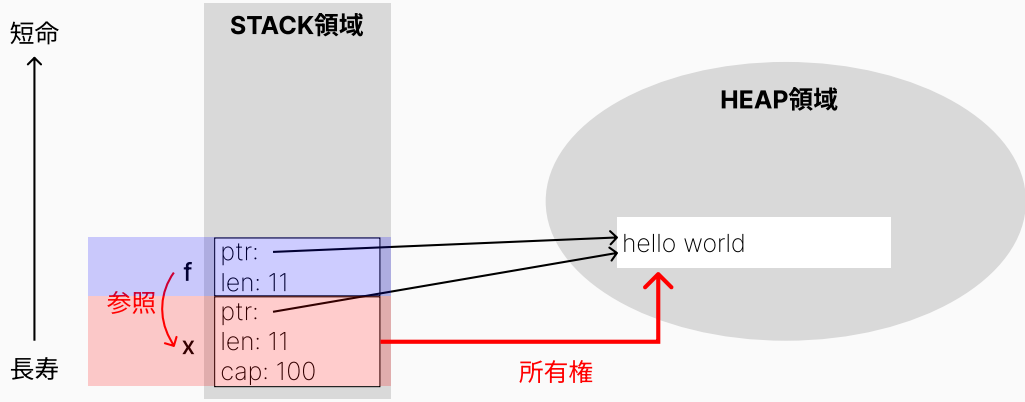

引数に所有権を渡すと、所有権が移動する

このあと、x を使うことができない

# 所有権のイメージ(借用)

```rust
fn main() {
    let x = String::from("hello world");
    func(&x);
}

fn func(f: &str) {
    println!("{}", f);
}
```




- f は x の値を借りている
- f のほうが寿命が短いので、x を参照できる
- func(&x)のあとで x は、メモリを解放されていないので使うことができる
- 参照している間に、x を変更することはできない

# では、こちらはどうなる？

```rust
fn main() {
    let x = f();
}

fn f() -> Vec<String> {
    let mut v = vec![String::from("hello")];
    let a: String = String::from("!!");
    v.push(a);
    v
}
```

```rust
fn main() {
    let x = f();
}

fn f() -> Vec<&str> {
    let v = vec!["hello", "world"];
    let a: &str = "!!";
    v.push(a);
    v
}
```

# では、こちらはどうなる？

それではこちらはどうなる？

```rust
fn main() {
    let s = "hello";
    let x = f(s);
}

fn f(s: &str) -> Vec<&str> {
    let mut v = vec!["hello"];
    v.push(s);
    v
}
```

# 参照

# 参照

- まず参照は危険という認識が必要
  - 参照先を書き換えると、そのデータを利用している他の参照にも影響がある

可変な参照を行うには、

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

    change(&mut s);
}

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

制約として、可変な参照は１つのスコープ内で１つしか持てない

# ライフタイム

- x のほうが f よりも長く生存している
- func あとに x を使えるか使えないかは
  - heap のメモリが解放されているかどうか
  - 所有権を持っているかどうか

**ライフタイム**(その参照が有効になるスコープ)が自然にわかる

```rust
{
    let r;

    {
        let x = 5;
        r = &x;
    }

    println!("r: {}", r); // ERROR
}
```

# メモリ安全性

ヒープ領域が `free` されると、スタック領域にあるポインタが無効になる

所有権では絶対に以下の問題が起こらない

- ダンダリングポインタ：メモリ領域の無効な場所を指し示しているポインタ
  - 2 重 free が起こると、メモリの破壊が起こる

# 問題 1

通る？

```rust
fn main() {
  let mut v = vec![1, 2, 3, 4, 5];
  let first = &v[0];
  v.clear();
  println!("The first element is: {}", first);
}
```

# 問題 1

通る？

```rust
fn main() {
  let mut v = vec![1, 2, 3, 4, 5];
  let first = &v[0];
  v.clear();
  println!("The first element is: {}", first);
}
```

- v の要素への参照を保持したまま、v の内容をクリアしている
- 無効な参照を引き起こす可能性があるため、コンパイラによってエラーとして検出される

# 問題 2

```rust
fn add_to_slice(slice: &[str], new_item: &str) {
    slice.push(new_item);
}

fn main() {
    let mut words = vec!["hello", "world"];
    let slice = &mut words[..];
    add_to_slice(slice, "!");
}
```

# 問題 2

```rust
fn add_to_slice(slice: &[str], new_item: &str) {
    slice.push(new_item);
}

fn main() {
    let mut words = vec!["hello", "world"];
    let slice = &mut words[..];
    add_to_slice(slice, "!");
}
```

`&[str]`は `push` というメソッドを持っていない

もっていたとしても、所有権を借りている身で `words` の配列の長さを変更することはできない

# 問題 3

```rust
fn get_slice_length(slice: &[str]) -> usize {
    slice.len()
}

fn main() {
    let words = vec!["hello", "world"];
    let slice = &words[..];
    println!("Slice length: {}", get_slice_length(slice));
}
```

# 問題 3

```rust
fn get_slice_length(slice: &[str]) -> usize {
    slice.len()
}

fn main() {
    let words = vec!["hello", "world"];
    let slice = &words[..];
    println!("Slice length: {}", get_slice_length(slice));
}
```

`&[str]`で参照渡しをしており、データの所有権を移動していない

データの長さを取得している(非加工)ので、問題ない

# 問題 4

```rust
struct Person {
    name: String,
    age: u32,
}

fn main() {
    let p = Person {
        name: String::from("Alice"),
        age: 30,
    };
    let name = p.name;

    println!("The person's age is: {}", p.age);
    println!("The person's name is: {}", p.name);
}
```

# 問題 4

```rust
struct Person {
    name: String,
    age: u32,
}

fn main() {
    let p = Person {
        name: String::from("Alice"),
        age: 30,
    };
    let name = p.name;

    println!("The person's age is: {}", p.age);
    println!("The person's name is: {}", p.name);
}
```

構造体の一部のフィールド（name）を移動させると、元の構造体全体が部分的に移動した状態になり、移動されたフィールドにアクセスできなくなる

$+\alpha$

追加でしゃべりたいこと

- `Result` と `Option` の使い方
- `ptr` と `len` の万能さ
  - 配列のスライス
  - 文字列の抽出
  - `Copy` しないことのメモリ効率
- `'static`, `const` のデータは `.rodata` にある
- `let x = 5; let y = x;` は `Copy` なので注意
- Rust はバッファオーバーフロー攻撃がないメモリ安全
- 型安全性

# 参考文献

# 参考文献


- The Rust Programming Language 日本語版 [link]
  - コードをたくさん引用しました。ありがとうございます。
- 【Rust 入門】宮乃やみさんに Rust の所有権とライフタイムを絶対理解させる [link]
  - 所有権の世界一わかりやすい説明