

# 組み込み Rust 講習会

Rust 編

---

JIJINBEI

2024-07-28

HiCoder

# Table of contents

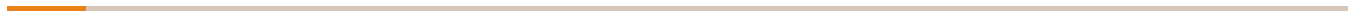
1. はじめに
2. 目標
3. Rust とは
4. 変数
5. データ型
6. 配列
7. 関数、制御フロー
8. 構造体とメソッド
9. ジェネリック
10. トrait

11. 所有権とライフタイム

12.  $+\alpha$

13. 参考文献

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



- 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<sup>1</sup>

**Rust is God**

---

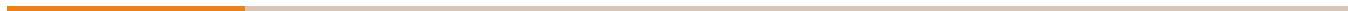
<sup>1</sup>このスライドも Rust 製の Typst で作成

# 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

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

rust

## 変数を可変するには、mut を使う

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

rust

# 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

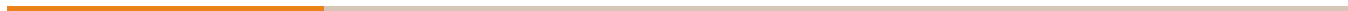
表 1: 整数型

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

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

rust

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

```
1 fn main() {  
2     let world: &str = "world!";  
3     println!("Hello, {}", world);  
4 }
```

rust

```
1 fn main() {  
2     let mut world: String = String::from("world");  
3     world.push_str("!");  
4     println!("Hello, {}", world);  
5 }
```

rust

String 型は配列の長さを変更できる(Vec)

# 関数、制御フロー

---

省略

# 構造体とメソッド

---

# 構造体とメソッド

```
1 #[derive(Debug)]
2 struct Rectangle {
3     width: u32,
4     height: u32,
5 }
6
7 impl Rectangle {
8     fn area(&self) -> u32 {
9         self.width * self.height
10    }
11 }
12
13 fn main() {
14     let rect1 = Rectangle { width: 30, height: 50 };
15
16     println!(
17         "The area of the rectangle is {} square pixels.",
18         rect1.area()
19     );
20 }
```

rust

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

# 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

```
1 struct Point<T> {  
2     x: T,  
3     y: T,  
4 }  
5  
6 fn main() {  
7     let integer = Point { x: 5, y: 10 };  
8     let float = Point { x: 1.0, y: 4.0 };  
9 }
```

rust

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<sup>1</sup>

---

<sup>1</sup>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

```
1 pub trait Summary {  
2     fn summarize(&self) -> String;  
3 }  
4  
5 pub struct NewsArticle {  
6     pub headline: String,  
7     pub location: String,  
8     pub author: String,  
9     pub content: String,  
10 }  
11  
12 impl Summary for NewsArticle {  
13     fn summarize(&self) -> String {
```

rust

# トレイト

```
14         format!("{}", by {} ({}))", self.headline, self.author,  
15 self.location)  
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(
```

```
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

**所有権**とは、メモリの解放を自動的に行うための仕組み

Rust のキモいところ

```
1 fn main() {  
2     let s1 = String::from("hello");  
3     let s2 = s1;  
4  
5     // println!("{}", world!", s1); // ERROR  
6     println!("{}", world!", s2);  
7 }
```

rust

```
1 fn main() {  
2     let s = String::from("hello");  
3     takes_ownership(s);  
4     // println!("{}", s); // ERROR  
5 }  
6  
7 fn takes_ownership(some_string: String) {  
8     println!("{}", some_string);  
9 }
```

rust

# 所有権のルール

- 値(データ)は所有者と呼ばれる変数と対応 (1 対 1 対応)
- 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)
- 所有者がスコープから外れたら、値は破棄
- 参照渡し(&)を使うことで、所有権を移さずに値を借用
  - 生存期間の長いものが、短いものを参照してはいけない

# 所有権のルール

- 値(データ)は所有者と呼ばれる変数と対応 (1 対 1 対応)
- 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)
- 所有者がスコープから外れたら、値は破棄
- 参照渡し(&)を使うことで、所有権を移さずに値を借用
  - 生存期間の長いものが、短いものを参照してはいけない

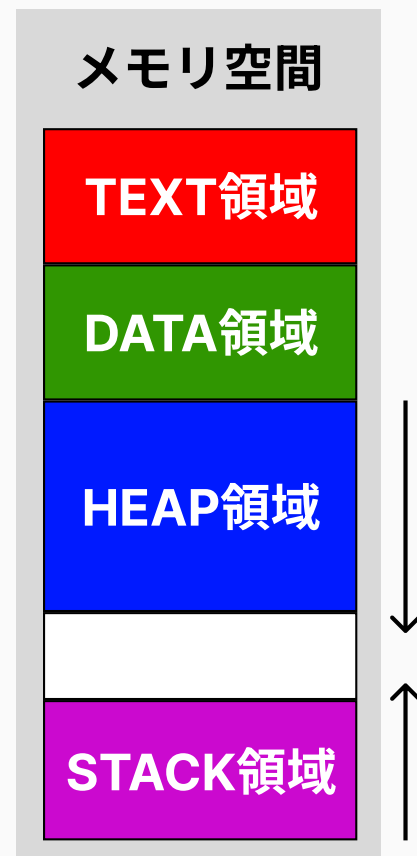
? ? ? ? ? ? ? ?



# ヒープとスタック

所有権を理解するためには、ヒープとスタックの違い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
  - 初期値のデータ
    - global 変数, 静的変数
- **HEAP 領域**
  - 寿命があるデータ
    - 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データ（固定長）
    - ヒープ領域を参照するポインタを持つ



# バイナリを見る

Rust のバイナリを見てよう

```
1 $ objdump -S binary_file  
2 $ objdump -h binary_file
```

terminal

.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

- スタックにはデータのポインタが、ヒープにはデータがある
- 変数の定義で、スタックは下から積み上げられ、ヒープでデータを確保される
- 所有権を持っている変数は、スコープ<sup>1</sup>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

---

<sup>1</sup>{ } の領域内

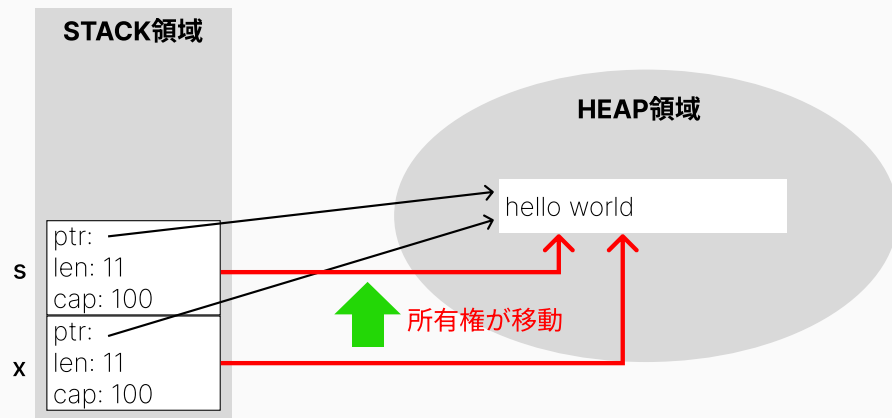
# 所有権のイメージ

```
1 fn main() {  
2     let x = String::from("hello  
3     world");  
4     func(x);  
5 }  
6 fn func(s: String) {  
7     println!("{}", s);  
8 }
```

rust

短命

↑  
長寿



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

## データの流れ

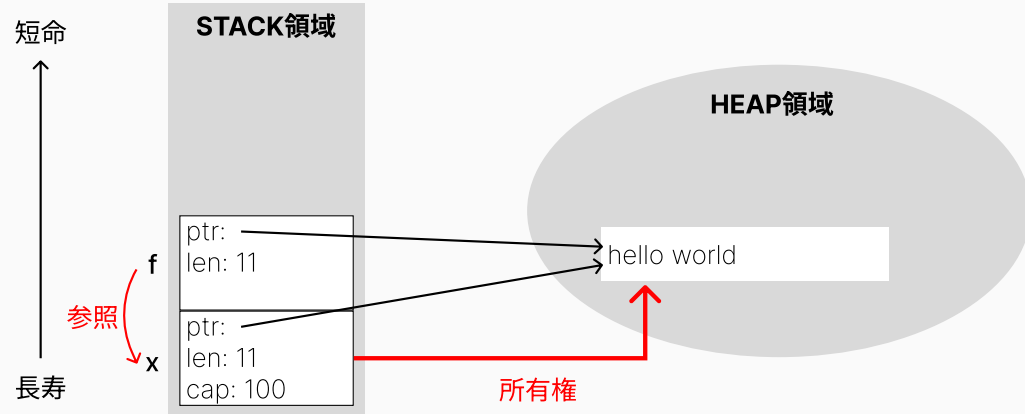
1. **x** が stack + ヒープにメモリ確保(malloc) + **x** がデータの所有権を持つ
2. **s** が stack + 所有権の移動(move)
3. **func** のスコープの外で **s** が破棄(push)、ヒープのメモリを解放(free)

- データのコピーはコストがかかるが、今回の例ではコストが小さい
- `func(x)`のあとで `x` は、メモリを解放されているので使うことはできない

# 所有権のイメージ

```
1 fn main() {  
2     let x = String::from("hello  
3     world");  
4     func(&x);  
5 }  
6 fn func(f: &str) {  
7     println!("{}", f);  
8 }
```

rust



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

## データの流れ

1. x が stack + ヒープにメモリ確保(malloc) + x がデータの所有権を持つ
2. f が stack + x を参照 + 所有権の移動が起こらない
3. func のスコープの外で f が破棄(pop)、所有権を持っていないのでヒープのメモリは解放されない

- 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
- `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

# 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

```
1 {  
2     let r;  
3  
4     {  
5         let x = 5;  
6         r = &x;  
7     }  
8  
9     println!("r: {}", r); // ERROR  
10 }
```

rust



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

- ダングリングポインタ：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？

```
1 fn main() {  
2     let mut v = vec![1, 2, 3, 4, 5];  
3     let first = &v[0];  
4     v.clear();  
5     println!("The first element is: {}", first);  
6 }
```

rust

# 問題 1

通る？

```
1 fn main() {  
2     let mut v = vec![1, 2, 3, 4, 5];  
3     let first = &v[0];  
4     v.clear();  
5     println!("The first element is: {}", first);  
6 }
```

rust

vector の要素への参照を保持したまま、vector の内容をクリアしている。

これは、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ます。

## 問題 2

```
1 fn add_to_slice(slice: &[str], new_item: &str) {  
2     slice.push(new_item);  
3 }  
4  
5 fn main() {  
6     let mut words = vec!["hello", "world"];  
7     let slice = &mut words[..];  
8     add_to_slice(slice, "!");  
9 }
```

rust

## 問題 2

```
1 fn add_to_slice(slice: &[str], new_item: &str) {  
2     slice.push(new_item);  
3 }  
4  
5 fn main() {  
6     let mut words = vec!["hello", "world"];  
7     let slice = &mut words[..];  
8     add_to_slice(slice, "!");  
9 }
```

rust

&[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

```
1 fn get_slice_length(slice: &[str]) -> usize {  
2     slice.len()  
3 }  
4  
5 fn main() {  
6     let words = vec!["hello", "world"];  
7     let slice = &words[..];  
8     println!("Slice length: {}", get_slice_length(slice));  
9 }
```

rust

## 問題 3

```
1 fn get_slice_length(slice: &[str]) -> usize {  
2     slice.len()  
3 }  
4  
5 fn main() {  
6     let words = vec!["hello", "world"];  
7     let slice = &words[..];  
8     println!("Slice length: {}", get_slice_length(slice));  
9 }
```

rust

&[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

```
1 struct Person {
2     name: String,
3     age: u32,
4 }
5
6 fn main() {
7     let p = Person {
8         name: String::from("Alice"),
9         age: 30,
10    };
11    let name = p.name;
12
13    println!("The person's age is: {}", p.age);
14    println!("The person's name is: {}", p.name);
15 }
```

rust



## 問題 4

```
1 struct Person {  
2     name: String,  
3     age: u32,  
4 }  
5  
6 fn main() {  
7     let p = Person {  
8         name: String::from("Alice"),  
9         age: 30,  
10    };  
11    let name = p.name;  
12  
13    println!("The person's age is: {}", p.age);  
14    println!("The person's name is: {}", p.name);  
15 }
```

rust

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

- 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

## 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