# 前回 (2022-07-19) からの差分

特になし

# 今日のゴール

Rust のだいたいの文法がわかる。

# 基礎文法最速マスター

## サンプルコード

```rust
fn main() {
    // variables
    let x = 1;
    let x = x + 2;      // shadowing

    println!("x is {}", x);

    let mut y = 1;      // mut means "mutable"
    y = y + x;
    println!("y is {}", y);

    // function
    sugoi_func(123, "Hello");

    // structure
    let foo = Foo::new("World".to_string());
    println!("{}", foo.message);
}

fn sugoi_func(a: i32, b: &str) -> () {
    println!("sugoi_func: a: {}, b: {}", a, b);
}

#[derive(Debug)]
struct Foo {
```

```
    message: String,
}

impl Foo {
    fn new(m: String) -> Foo {
        Foo { message: m }
    }
}
```

```
x is 3
y is 4
sugoi_func: a: 123, b: Hello
World
```

## 全体

- main() 関数がプログラムのエントリーポイントとなり、プログラム内で 1 つだけ実装する
    - ライブラリを作ってるときは main() 関数を置かない
- main() 関数を置くのは、main.rs というファイル名にすることが多い
    - Cargo で管理しているなら、src/main.rs になる
- インデントは 4 スペース
- 式の末尾にはセミコロンを置く
    - ブロックの最後の行にセミコロンを置かない場合、意図的にそれを最終評価式にすることができる
- コメントは // を使う
    - コメントの書き方を駆使することによって、ドキュメント化できる（cargo doc）
- ファイルごとに世界が隔絶しており、外部に使わせたい何かがあるときは pub キーワードをつける必要あり
- #[XXX] はアトリビュートといって、いろいろな効果を付与できる
    - Windows 用にだけビルドさせたいときは、#[cfg(target_os = "windows")] という感じ

    - #![XXX] （ビックリがついてる）という書き方のときは、適用範囲がクレート全体になる
    - derive を使うパターンは、マクロとなっている
- ! が末尾についてる関数もマクロ（println!() みたいな）。コンパイルのときに多くのコードに展開される
    - 最初から用意されている標準マクロ以外にも、自分で作成することもできる

## 変数

- 初めて出てくる変数には let をつける
    - 定数には const をつける
- let だけの場合、変数は不変となるので、一度束縛すると変更できない
    - （Rust は「代入」と言わずに、「束縛」という単語を使う）
    - これは、配列の中身の変更や、辞書の中身の追加にも適用される
- let mut をつけると、可変変数となる
- 再度 let をつけることで同じ変数名を使用可能（シャドーイング）
    - 別の型でも OK
- 変数束縛時に型を必要とするが、コンパイラが推論できるときは、省略できる
    - 逆に言うと、コンパイラがわからないときは、型は必須となる

## 型

- スカラ型は 4 つ（整数、浮動小数点数、論理値、文字）
    - i8, i16, i32, i64, isize, u8, u16, u32, u64, usize, f32, f64, bool, char
- 複数型
    - タプル型
        - (型名, 型名, 型名...) ← 複数の型をまとめたもの
        - let t: (i32, char) = (123, ‘c’);
        - t.0, t.1 みたいに取り出す
        - 配列ではない
        - 中身が何もないタプルを「ユニット」と呼び、「何もない」的なことを表す

    - 配列型
        - タプルと違い、要素は一種類の型でないといけない

## 関数

- fn で関数定義する
- 引数と返却値には型が必要となる
    - 値を返さないときは、-> () を置くが、置かない方が自然（コンパイラがやってくれる）
- return しなくても、最終評価式が返却値となるが、その際セミコロンを置いてはいけない
- ジェネリクスも使える
    - fn foo<T>(a: T) -> T { … } こんな感じ

## 構造体

- struct で構造体の定義をする。構造体はクラスのようなもの。
- 構造体にフィールド値を渡すことで、インスタンスを作成できる
    - let foo = Foo { x: 123 };
- よくあるのは、new メソッドも用意してあげて、new メソッド経由でインスタンスを取得する方法
    - いわゆるクラスメソッドのことを関連関数と呼ぶ
- インスタンスが実行できるメソッドは、定義の引数の1つ目が self になっている
- ジェネリクスも使える
    - struct Foo<T> { x: T } こんな感じ

## そのほか

- 所有権とライフタイムの概念がある
    - これがマジこれ
- enum がある
    - Option と Result はとにかくよく出てくるので嫌でも覚える
- 継承はない

- トレイトがある
    - Rust のトレイトは、いわゆる「インターフェース」的なやつ
    - トレイトの実装が必要な、トレイト境界 (TraitBound) という仕様がある
- 単体テストが書ける

ジェネリックな型引数とトレイト境界とライムタイムを指定する関数はこんな感じになる。

```
fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str,
    ann: T,
) -> &'a str
where
    T: std::fmt::Display,
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
         x
    } else {
        y
    }
}
```

# 入門用 URL

Rust の最初のステップ - Learn | Microsoft Docs

一般的なプログラミングの概念 - The Rust Programming Language 日本語版