

# Rust 入門

# Rust とは

## 速い

- LLVM を利用してネイティブコードを生成する（らしい）

## 簡単に書ける

- 動的型付けのおかげで気軽に書ける。対話環境も充実。

## いろいろな構成に対応

- 多重ディスパッチでOOP（構造体に関数が作用する書き方）や関数型っぽく書ける

## そしてオープンソース

# インストール

- Windows : [Windows](#) で Rust 用の開発環境を設定する
  - Windows では Microsoft C++ Build Tools が必要
  - WSLなら `curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh`
- Mac : [macOSでRustのローカル開発環境を整えるための手順2022](#)
  - WSLと同じコマンドでよさそう
- `cargo --version` でバージョンが表示されればOK
- 適宜 VSCode の拡張機能を入れる ([rust-analyzer](#), [CodeLLDB](#))

# VSCodeでのデバッグ、実行

- `cargo new {project name}` でプロジェクトが作成される
- プロジェクトフォルダをVSCodeで開き、ダイアログで `OK`, `Yes` を押すと `launch.json` が作られる (参考)
- 次から `F5` でデバッグ、`Ctrl+F5` で実行できるようになる
- `rustc {filename}.rs` でビルドされる

# Rust の仕様

# Rustでは所有権を通してオブジェクトに触る

- スコープを抜ける or 所有権を渡すともとの変数が持っていた所有権は消失する。**参照**を渡すとい
- 所有権がすべて失われるとオブジェクトが破棄される

	不変	可変
原本	<code>let a = 10;</code>	<code>let mut b = 20;</code>
参照	<code>let aref = &amp;a;</code>	<code>let bref = &amp;mut b;</code>

- 関数呼び出し時に**借用チェック**がクリアされる必要がある
  - 不変参照と可変参照が同時に存在しない
  - 可変参照は1つしか存在しない

# コピートレイト

- コピートレイトの実装された型のコピーでは、所有権は移動せずオブジェクトがコピーされる
- Rustのプリミティブ型、不変参照はコピートレイトを実装している
- 可変参照はコピーされない
- オブジェクト指向言語でプリミティブ型はdeep copyされるのに似ている
- **トレイト境界**：あるトレイトを実装しているか否かで型を制限できる

```
fn copy_trait_check<T: Copy>(_: T) {}
```

に渡せる型はコピートレイトを実装している

# データ型

- 整数、`f32`, `f64`, `bool`, `char`, タプル、配列など
- 数字リテラルは `_` で区切れる
- `0x`, `0o`, `0b` が16,8,2進数リテラルの接頭辞



# もろもろ

- パターン分解できる： `let (x,y,z) = (1,2,3);`。無視するときは `_`
- 型を明示的に指定できる（普段はrust-analyzerが推定される型を表示している）
- 同じスコープ内で変数を使い回せる
- シャドーイングが有効
- キャストは `as` を使う： `let c = 13u8 as u32 + 7u32;`

# 配列・スライス

- 配列: `let a = [1,2,3,4,5];`
- スライス: `let s = &a[1..3];`
- 範囲の指定 `x..y` :  $x \leq i < y$ , `x..=y` :  $x \leq i \leq y$ , `..y` :  $i < y$  など
- 範囲指定は `for` 文でも使える

```
let mut sum = 0;
for i in 1..10 {
    sum += i;
}
```

- 文字列リテラル、文字列のスライスは `&str` 型
- `String::from("uouo")` は `String` 型

# 関数

```
fn add(a: i32, b:i32) -> i32 {  
    a + b  
}
```

- 引数は（返り値がある場合は返り値も）型を指定する必要がある
- 引数でもパターン分解を使える

```
fn add(&(x, y): &(i32, i32)) -> i32 {  
    x + y  
}  
fn main() {  
    println!("{}", add(&(1, 2)));  
}
```

# 制御構文

- `if 条件 { ... } else if 条件 { ... } else { ... }`
- `if` 文も式を返す: `let mod2 = if a % 2 == 1 {"odd"} else {"even"};`
- `loop` 文は `break` で値を返せる (条件式はない)

```
fn main() {  
    let mut a = 7; let mut ret = 1;  
    let a0 = a;  
    let result = loop {  
        ret *= a; a -= 1;  
        if a == 0 { break ret; }  
    };  
    println!("{}", a0, result);  
}
```

- `for`, `while` は値を返さない (`()` を返す)

# 構造体

```
struct Cmplx { real: f32, imag: f32 }
fn main() {
    let mut z = Cmplx { real: 0.5, imag: 0.9 };
    z.imag = -3.;
    fn build(real: f32, imag: f32) -> Cmplx { Cmplx {imag, real} }
    let w = Cmplx {imag: 4., ..build(1., 2.)};
    println!("w = {} + {} i", w.real, w.imag); // w = 1 + 4 i
    println!("z = {} + {} i", z.real, z.imag); // z = 0.5 + -3 i
}
```

- `let mut` するとフィールド全てが可変になる
- 変数名と一致しているフィールド名は省略できる
- 指定しなかったフィールドは `..` 後のオブジェクトのフィールドを束縛する

# タプル構造体

```
struct Cmplx(f32, f32);  
fn main() {  
    let z = Cmplx(1., 2.);  
    let Cmplx(x, y) = z;  
    println!("{}", z.0, z.1);  
    println!("{}", x, y);  
}
```

- `.0` などでアクセスする
- 要素の分解もできる

# メソッド、関連関数

```
struct Cmplx { real: f32, imag: f32 }
impl Cmplx {
    fn abs(&self) -> f32 { self.real * self.real + self.imag * self.imag }
    fn new(real: f32, imag: f32) -> Self { Self { real, imag } }
}
fn main() {
    let z = Cmplx::new(3., 5.);
    println!("{}", z.abs());
}
```

- 書き換えるときは `&self` の代わりに `&mut self`
- `self` だと呼び出し元のオブジェクトの所有権がメソッドに渡り、元の場所でそのオブジェクトを使えなくなる
- `Self` は型エイリアス

# 枚举型

```
enum OpticalDisc { CD(u32), BD(String, u32) }  
fn main() {  
    let bd = OpticalDisc::BD(String::from("ROM"), 120);  
}
```

- 枚举子に値を持たせられる。持たせる値の型は枚举子ごとに違っていい
- メソッドも定義できるシトレイトも実装できる
- 枚举子に持たせたオブジェクトは `match` 文で取り出せる



# ジェネリクス

```
use std::f32::consts::PI;
enum OpticalDisc<T> { CD(T), BD(String, T) }
impl<T> OpticalDisc<T> {
    fn newBD(r:T) -> OpticalDisc<T> {
        OpticalDisc::BD(String::from(""), r)
    }
}
impl OpticalDisc<f32> {
    fn area(&self) -> f32 {
        match *self {
            OpticalDisc::CD(r) => r.powi(2) * PI / 4.,
            OpticalDisc::BD(_, r) => r.powi(2) * PI / 4.
        }
    }
}
fn main() {
    let bd = OpticalDisc::<f32>::newBD(120.);
    println!("area of bd is {}", bd.area());
}
```

- ジェネリック型：任意の型を受け取れるようにしてコードの重複を回避する仕組み
- ジェネリック型のメソッドを定義できる。特定の型のときのメソッドを定義することもできる
- コンパイル時は型が分かるので型に特化したコードが生成される（単相化）
- **turbofish演算子** `::<...>` で型を明示的に指定できる

# パターンマッチ

```
fn main() {  
    let a: Option<String> = Some(String::from("hello"));  
    match a {  
        Some(x) => println!("{}", x), // move ownership  
        None => ()  
    }  
    println!("{:?}", a);  
}
```

- パターンマッチはマッチした腕だけが評価される（短絡評価）
- 残りのパターンは `_` で受け取れる
- パターンマッチで所有権が移る

# 参考文献

- [Rust入門](#)