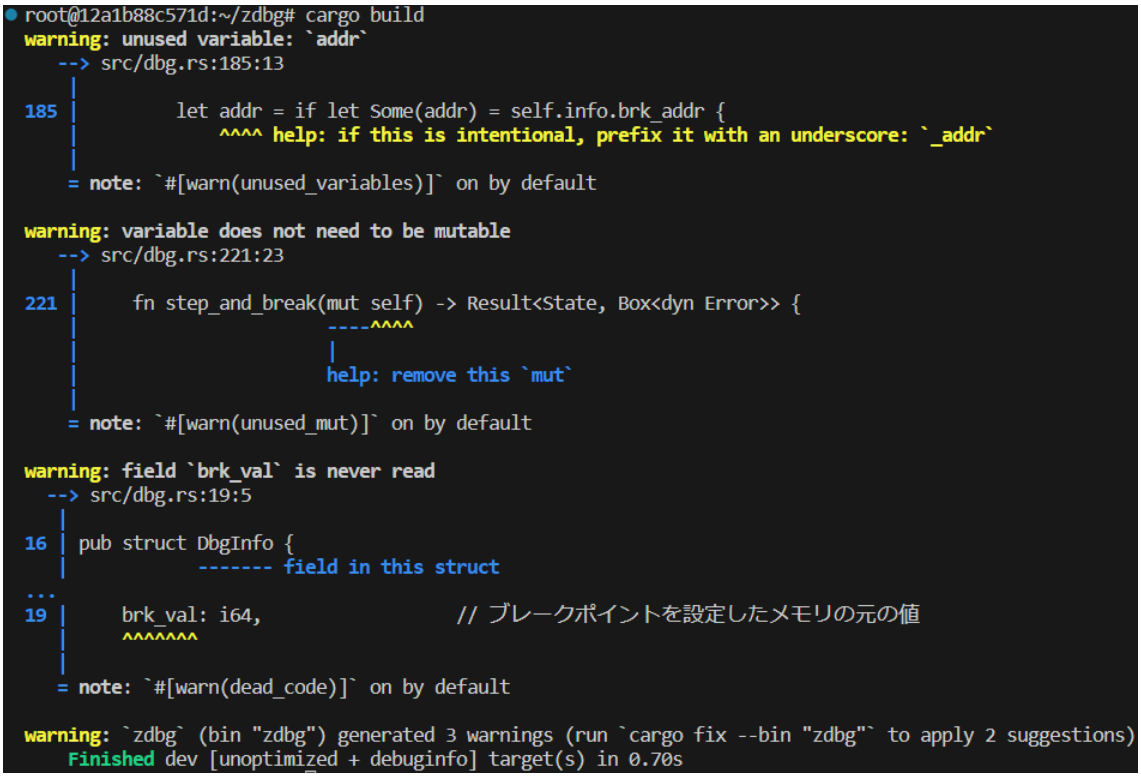
# サイバーセキュリティ PBLII 事前課題

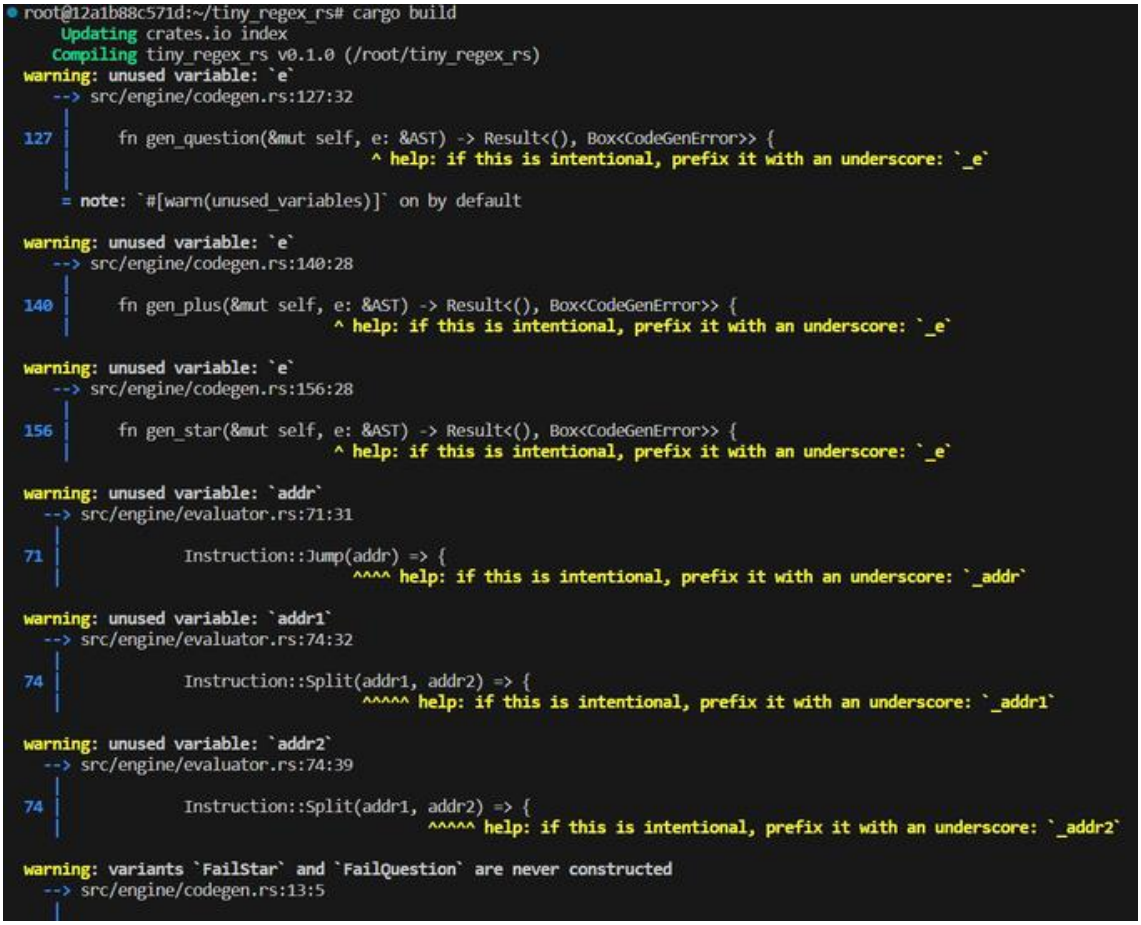
28G23027
川原尚己

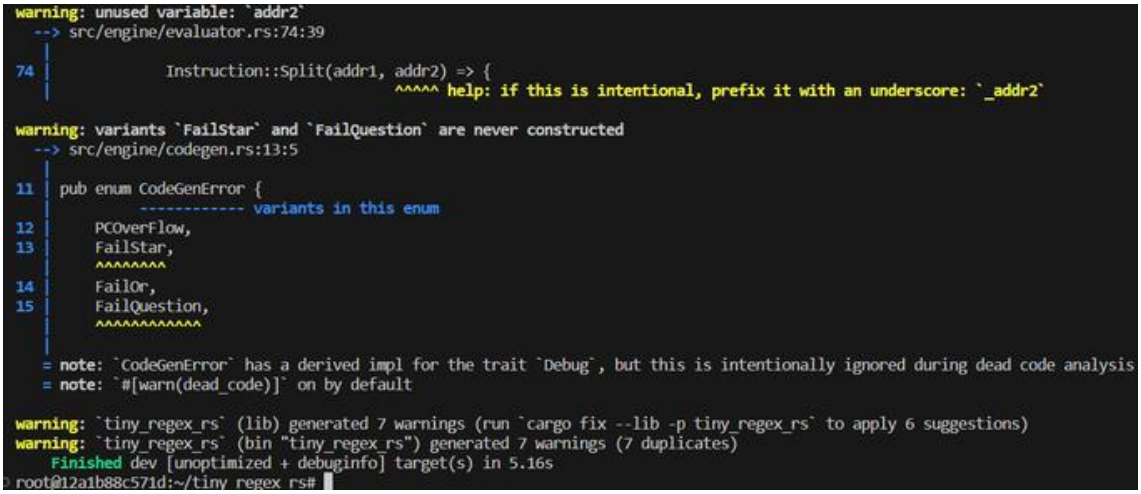
■ Linux 環境を用意し、課題用ソースコードのコンパイルができたことを示す レポートを提出せよ.

以下に zbdg と tiny_regex_rs のコンパイル結果を示す.




zdbg のコンパイル結果




Tiny_regex_rs のコンパイル結果

■ The Rust Programming Language (https://doc.rust-jp.rs/book-ja/)の 5 章までを読んで、Rust の基本を解説した簡単なドキュメントを作成せよ。

1. プロジェクト作成：”cargo new <フォルダ名>”で行うことができる．このコマンドを用いてプロジェクトを作成すると，自動的に Rust ファイルと共に cargo の設定ファイル

である cargo.toml 及び git リポジトリも生成される．

2. 変数と可変性

変数宣言は“let <変数名>”によって宣言できる．このように宣言した時には，変数は不変となり，一度宣言した後の書き換え（図１）が不可能である．

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

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
error[E0384]: cannot assign twice to immutable variable `x`
              (不変変数`x`に2回代入できません)
 --> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         (`x`への最初の代入)
  |         help: consider making this binding mutable: `mut x`
3 |     println!("The value of x is: {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable

For more information about this error, try `rustc --explain E0384`.
error: could not compile `variables` due to previous error
```

図１　不変な変数宣言と実行結果

しかし，変数宣言を”let mut <変数名>”とすると，可変な変数となり，書き換えが可能となる（図２）

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

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.30s
     Running `target/debug/variables`
The value of x is: 5    (xの値は5です)
The value of x is: 6
```

- 定数

  定数は”const <変数名>: <型名> = <値>”によって宣言できる．定数は変数とは異なり，常に不変となる．定数の宣言の際には，必ず型を宣言する必要がある．命名規則は，すべて大文字で，単語間はアンダースコアで区切ることである．

- シャドーイング

  Rust では，前に定義した変数と全く同じ変数を新しく宣言でき，このとき，新しい変数は前の変数を「覆い隠す」．（図３）

```
fn main() {
    let x = 5;

    let x = x + 1;

    {
        let x = x * 2;
        println!("The value of x in the inner scope is: {}", x);
    }

    println!("The value of x is: {}", x);
}
```

```
$ cargo run
   Compiling variables v0.1.0 (file:///projects/variables)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31s
     Running `target/debug/variables`
The value of x in the inner scope is: 12
The value of x is: 6
```

図３　シャドーイングの例と実行結果

図３のソースコードでは，３回 let 文によって変数 x が宣言されている．まず，一つ目の let 文によって，x に 5 を格納する．二つ目の let 文によって一つ目の宣言での x が覆い隠され，x に 6 が格納される．同様に三つ目の let も二つ目の x を覆い隠し，x に 12 が格納される．こうして一つ目の println!で 12 が出力され，波括弧を抜けるとシャドーイングは終了し，x の値は 6 に戻る．

シャドーイングと mut を用いることとの違いは二点存在する．一つ目は，シャドーイングの場合は変数の普遍性が保たれることである．二つ目は，シャドーイングでは実質的に新しい変数を宣言していることと等しいため，値の型を変えつつ，同じ変数名を使いまわせることである．

3. データ型

Rust は静的型付き言語であるため，コンパイル時にすべての変数の型が判明してい

る必要がある．

- スカラー型

　スカラー型は単独の値を表す．Rust には主に整数，浮動小数点数，理論値，文字の四つのスカラー型が存在する．

- 整数型

　整数型には 8,16,32,64-bit 及びコンピュータのアーキテクチャ依存のビット数と符号付きか符号なしかの計 10 種類が存在する．符号付きなら”i{ビット数}”，符号なしなら”u{ビット数}”という型名になる．また，アーキテクチャ依存の場合には”{u または i}size”という型になる．例えば，符号なしの 64-bit 整数型なら”u64”となる．整数型の基準は”i32”型であり，64 ビットシステム上でもこの型が通常，最速となる．

　また，整数リテラルは 10 進数，16 進数，8 進数，2 進数，バイト表現（”u8”のみ）があり，10 進数を除き，それぞれ”0x”，”0o”，”0b”，”b”をつけることによって表現できる．

- 浮動小数点型

　浮動小数点型には 32 ビットと 64 ビットサイズがそんざいしており，それぞれ”f32”，”f64”で得られる．基準型は”f64”であり，型を明示せずに浮動小数点数を宣言した際には”f64”型が選ばれる．

- 理論値

論理値型は”bool”によって型を宣言することができ，その値は”true”または”false”である．

- 文字

　Rust の”char”型は最も基本的なアルファベット型である．”char”型はダブルクォーテーションマークを使用する文字列に対して、シングルクォートで指定される．

- 複合型

　複合型により，複数の値を一つの型にまとめることができる．Rust には，タプルと配列という二種類の基本的な複合型がある．

- タプル

　タプルは，丸括弧の中にカンマ区切りの値リストを書くことで生成できる．タプルの位置ごとに型があり，タプル内の値はそれぞれが同じ型である必要はない．（図４）

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
}
```

図４　タプルの宣言例

　タプル内の各値へのアクセスしたい値の番号をピリオドに続けて書くこともできる．（図５）

```rust
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;
}
```

図 5　タプルの各値へのアクセス

- 配列型

　配列は，各括弧内にカンマ区切りリストとして書くことで生成できる．タプルと異なり，配列の全要素は同じかでなければならない．また，Rust の配列は固定長であるため一度宣言されたらそのサイズを伸ばすことも縮めることもできない．

4. 関数

　関数は”fn”キーワードによって定義できる．関数宣言において，引数をもつように定義する場合には，仮引数の型を必ず宣言しなければならない．また，関数に戻り値を設定する場合には”->”の後に型を書いて宣言することができる．Rust では，関数の戻り値は，関数本体ブロックの最後に式の値と同義である．”return”キーワードによって関数から早期にリターンし，値を返すこともできるが，多くの関数は最後の式を暗黙的に返す．（図 6 ）もし図 6 の five 関数内の”5”の後ろにセミコロンをつけ”5;”としてしまうと，コンパイルエラーが発生する．これは five 関数の定義では”i32”型を返すと宣言しているのに”5”という式ではなく，”5;”という文になってしまい，何も返されないことになってしまうからである．

```rust
fn five() -> i32 {
    5
}

fn main() {
    let x = five();

    println!("The value of x is: {}", x);
}
```

図 6　関数宣言

5. 所有権

　次の図 7 のような処理を考える．

```
let s1 = String::from("hello");
let s2 = s1;
```

図 7　所有権

このコードではまず一行目で String 型の”hello”という値を s1 に格納し，二行目で s2 に s1 の値をコピーしているように見える．しかし，実際には異なった処理がなされている．まず一行目において a1 には次の図 8 のような値がスタックされる．

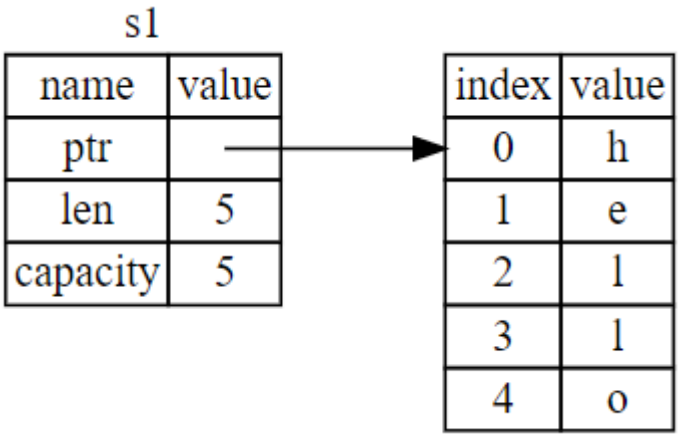



図 8　s1 への値の格納

もし，s2 への値の格納が先ほど説明したような方法で行われているのであれば，いかのように表される（図 9）．しかし，実際の Rust ではこのような処理は行っていない．このようにしてしまうと，ヒープ上のデータが大きいときに”s2=s1”という処理の実行時性能がとても悪くなってしまう可能性があるためである．実際の Rust では，図 10 のような処理を行っている．すなわち，スタックにあるポインタ，長さ，許容量はコピーするが，ポインタがさすヒープ上のデータはコピーしない．

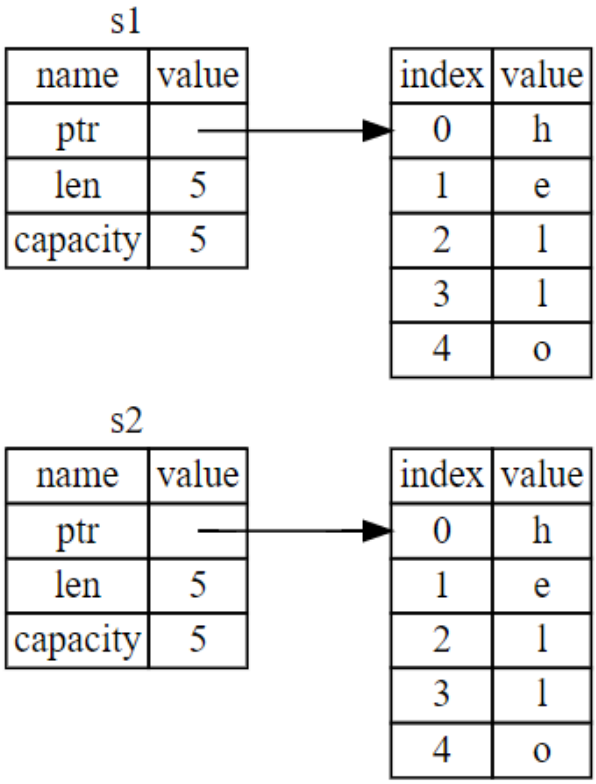



図 9　誤った”s2=s1”の解釈

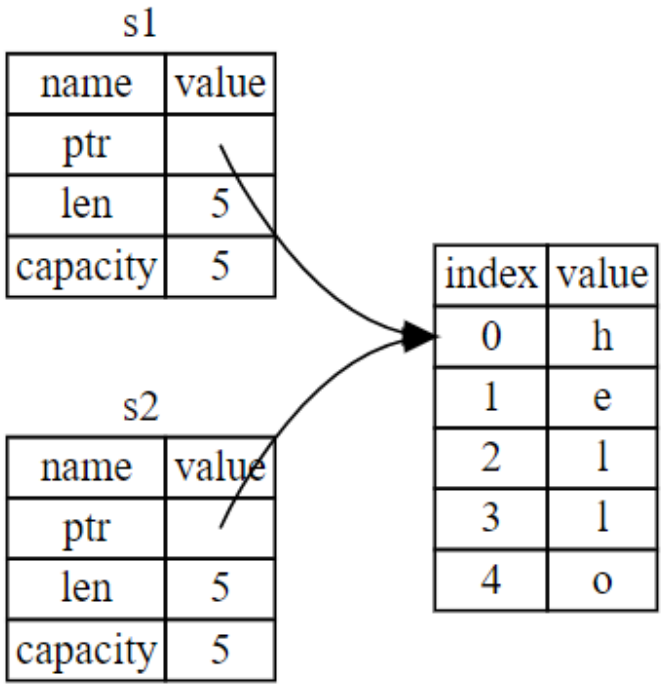



図 10　正しい”s2=s1”の解釈

しかし，このように複数の変数が一つのヒープを参照する場合には，「二重開放エラー」

が起きてしまう可能性がある．s1 と s2 がスコープを抜けたら両方とも同じ目盛りを解放しようとするからである．二重開放エラーを防ぐために行われている操作として「ムーブ」がある．s2 を生成した後に s1 がもはや有効ではないと考え，s1 がスコープを抜けた後に何も解放しなくてよいようにするのである．関数に値を渡したり，値を変数に代入することによって所有権が移る．

6. 構造体

構造体は”struct <構造体名>{<変数名>: <型名>}”という形で表される．（図 11）

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

図 11　構造体定義の例

また，構造体のインスタンスは図 12 のように宣言できる．

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
```

図 12　インスタンス宣言の例

インスタンスから特定の値を取得したいときには，ドット記法が利用できる．例えば，図 12 のインスタンスから email の値を取り出したいときは”user1.email”と書けばよい．また，インスタンスを可変として定義している（図 12 の例ならば”let mut user1 = …”とすればよい）ならば，”user1.email=hogehoge”とすることで書き換えが可能である．また，Rust はインスタンス内の一部のフィールドのみを可変にするということはできず，すべて不変にするか，すべて可変にするかのどちらかしかできない．