第2章 メモリ安全，所有権，借用

安全な変数アクセスの考え方と実現方法

池田 有

リスト1　Rustの所有権を使ったプログラム

```
 1:let numgpio_array: Vec<usize> = vec![42, 5, 6];
 2:green_led::init(numgpio_array);
 3:
 4:copy
 5:mod green_led {
 6:        :
 7:        :
 8:    pub fn init(num: Vec<usize>) {
 9:        // Configure the GPIO pin for output
10:        gpio_regs().gpfsel[num[0] / gpio::
               GPFSEL::PINS_PER_REGISTER].modify(
11:            gpio::GPFSEL::pin(num[0] % gpio::
               GPFSEL::PINS_PER_REGISTER).val(
               gpio::GPFSEL::OUTPUT),
12:        );
13:        :
14:        :
15:}
```

```
16:let numgpio_array: Vec<usize> = vec![42, 5, 6];
17:green_led::init(numgpio_array);
(省略)
23:
24:mod green_led {
25:        :
26:        :
27:    pub fn init(num: Vec<usize>) {
28:        // Configure the GPIO pin for output
29:        gpio_regs().gpfsel[num[0] / gpio::
             GPFSEL::PINS_PER_REGISTER].modify(
30:          gpio::GPFSEL::pin(num[0] %
             gpio::GPFSEL::PINS_PER_REGISTER).val(
             gpio::GPFSEL::OUTPUT),
31:        );
32:        :
33:        :
34:}
```

Rustでは，扱うデータに対し所有権という考え方があります．C言語ではなじみのない概念ですので，理解するのが難しいです．

ここでは，第2部第1章で作成したLチカ・プログラムを元に，少しコードを加えながら解説します．

C/C++とRustの所有権の違い

● C/C++の所有権

C言語の場合，動的メモリ領域はmallocで確保しfreeで解放しています．プログラマがfreeを書き忘れた場合，確保したメモリが解放されずにそのまま残るためメモリ・リークになります．するとやがてメモリを食いつぶし，システム・クラッシュに至ります．既にfree済みの領域に対してアクセスしてしまったり，freeしているのにまたfreeしてしまったりすることもあるかもしれません．

C++の場合は，newで確保しdeleteで開放します．deleteを忘れるとメモリ・リークになります．

これに対処するため，C++にはスマート・ポインタというものが存在します．スマート・ポインタは次のようにメモリを自動で解放してくれるものです．

- 変数がスコープから外れたときにメモリが開放するstd::unique_ptr
- 参照カウントを持ち，参照カウントがゼロになったときにメモリを解放するstd::shared_ptr

● Rustの所有権

Rustのメモリ管理は次のような概念で設計されています．

- メモリ（場所）には所有権がある
- 所有権を持っている人が使い終わったら（スコープ外になれば）勝手に開放する

つまり，全データ型[注1]に所有権が存在する一方，プログラマが解放に責任を持つ必要がなくなりました．

プログラムを例に…Rustの所有権を見てみる

● 所有権の移動

▶所有権を持つ変数は常に1つ

対象の変数が有効である範囲のことを，スコープと言います．Rustでは，変数に対し所有権を持つのは，必ず1つのスコープ内からのみです．スコープ外のところにデータが移るのであれば，移った先のスコープ

注1：usize型など，Copyトレイトといって所有権を自動で複製するような型も一部に存在します．この場合，プログラマは所有権を意識する必要はありません．