Rustと所有権

～さよならｾｸﾞﾌｫ，はじめましてﾎﾞﾛｰｴﾗｰ!～

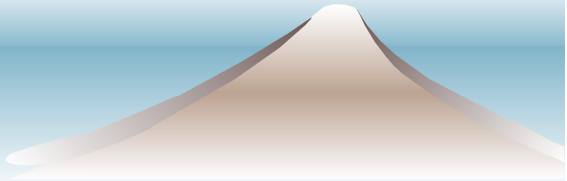
電子情報工学科　近藤 佑亮
吉田 光樹

はじめに

- コメントスクリーンについて
  - 下記リンクからのコメントが
    この共有画面上でリアルタイムで流れます
  - https://commentscreen.com/comments?room=tauIsGod
  - 質問 / 感想などを書き込んでください！
    - Twitterで #tauIsGod をつけても拾ってくれるらしい
- 投げ銭, ｽﾊﾟﾁｬは受け付けておりません
  - どうしてもしたい場合はこちらへ

本日の目標

- 講義をざっくりと復習する
  - メモリ安全性を中心に
- プログラミング言語「Rust」の概要を知る
  - プログラミング言語としてのRustの特徴，他言語との比較
  - メモリ管理手法「所有権」「借用」ついて，長所と短所
- Rustを愛し，Rustに愛される



導入

コーディングで詰まった...

ベースライン研究の論文実装が動かない...
公式ドキュメントを読んでも理由がわからん...

俺の答えはこれや！！！！！

Qiita

俺の答えはこれや！！！！！

Qiita

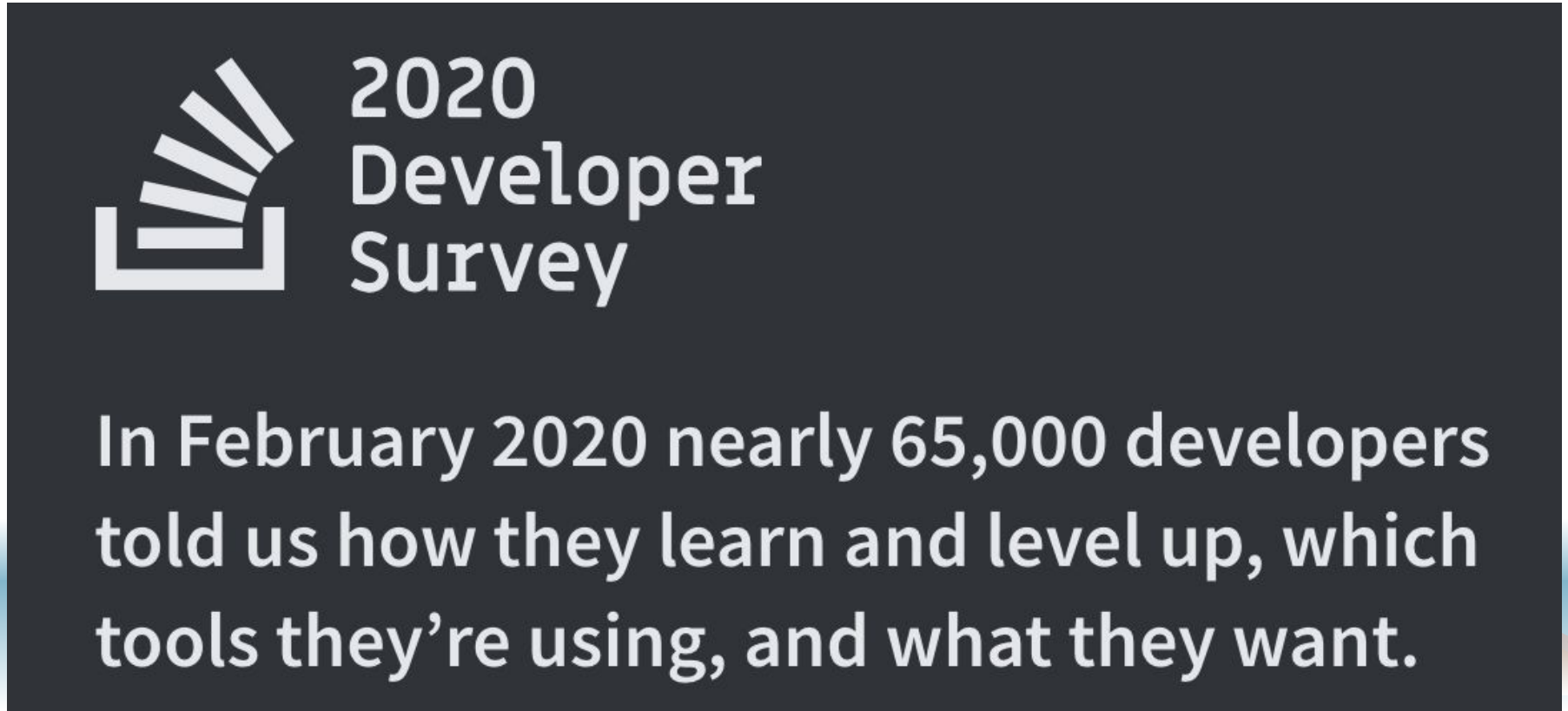
Hatena Blog

俺の答えはこれや！！！！！

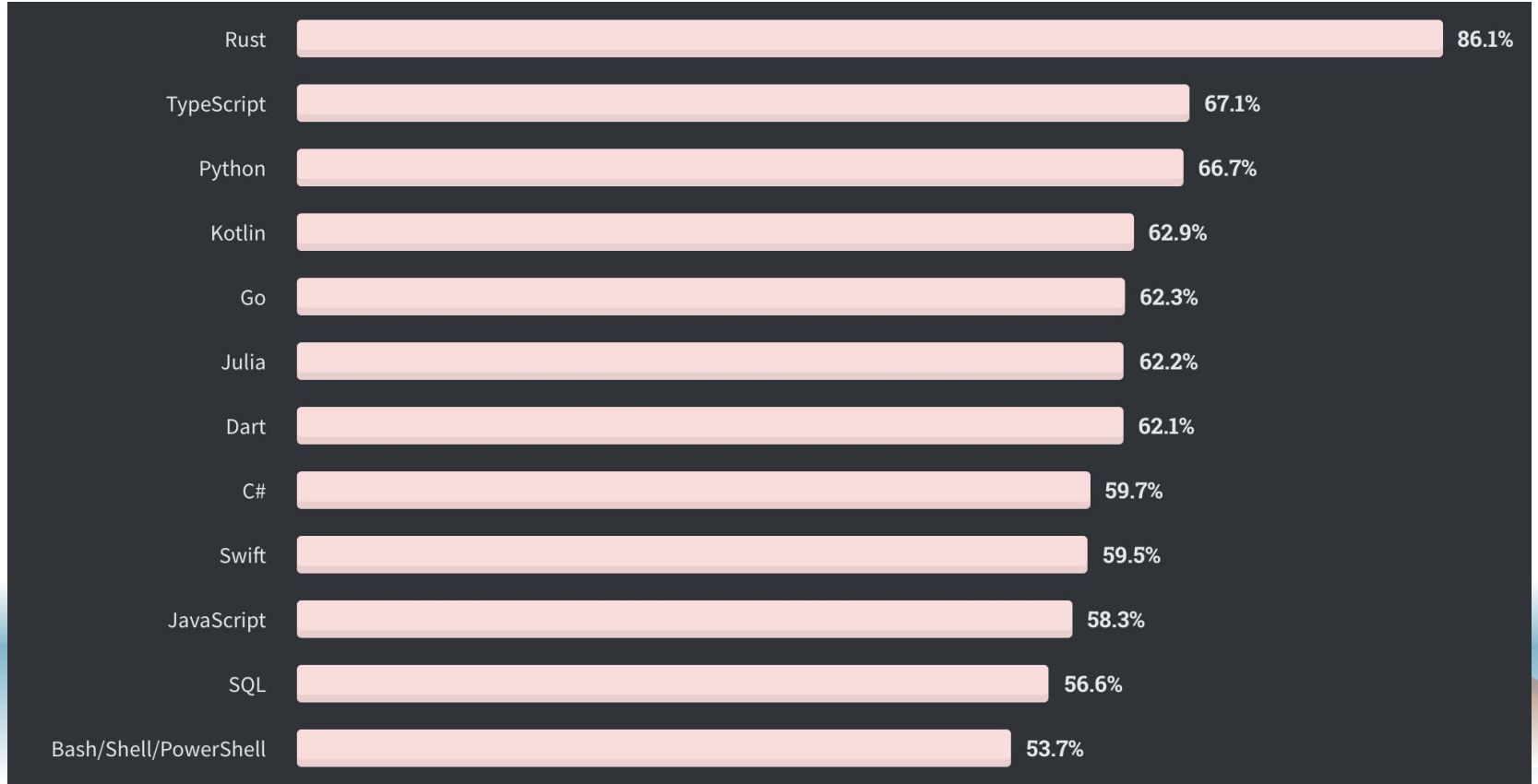
stackoverflow

プログラマの唯一神（諸説あり）

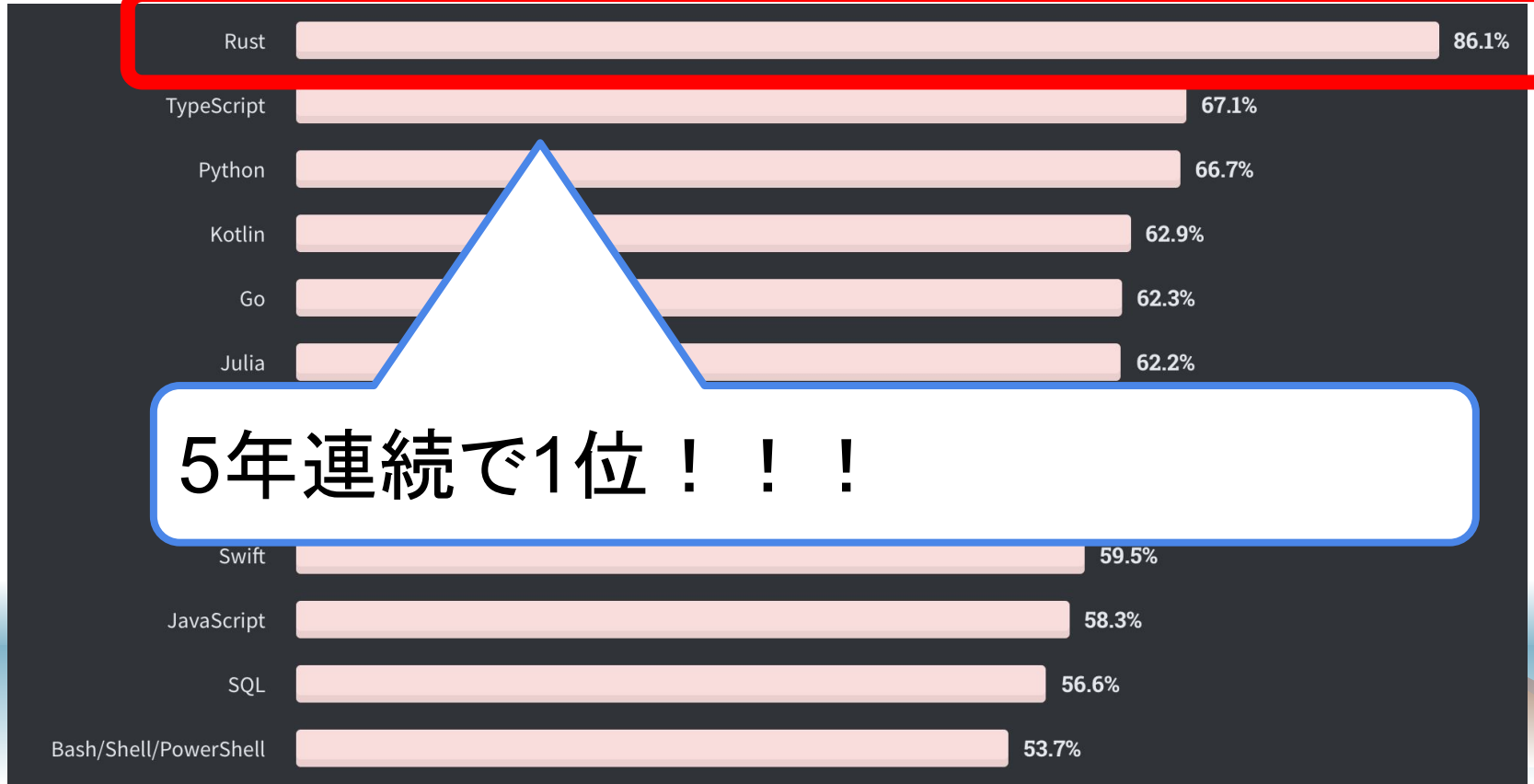
Stack Overflow Developer Survey



Most loved languages

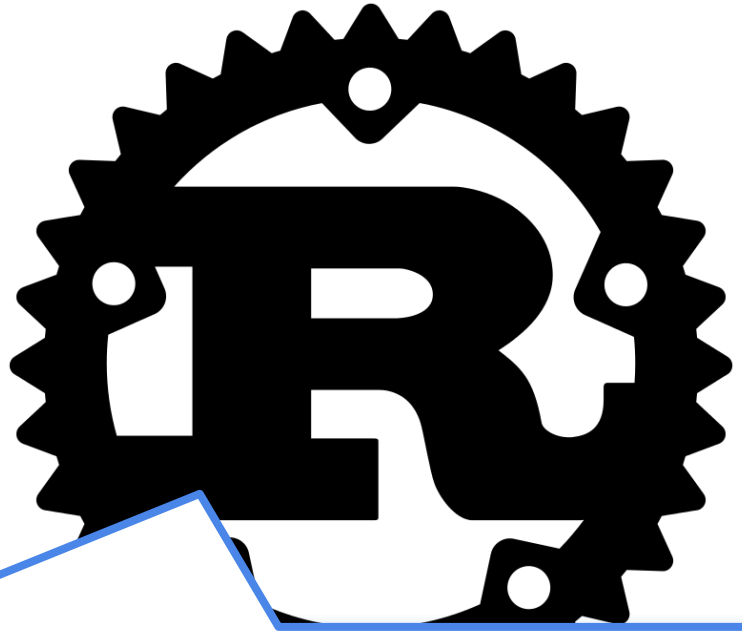


Most loved languages



なぜRustは開発者に好まれるか？

なぜRustは開発者に好まれるか？



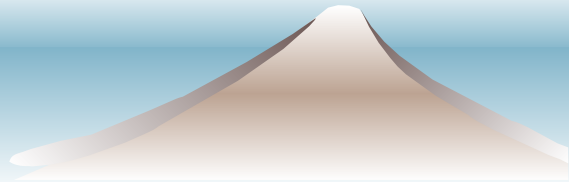
発表を通して
その雰囲気を掴んでいただけたら！

Rustのコンセプト

Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)

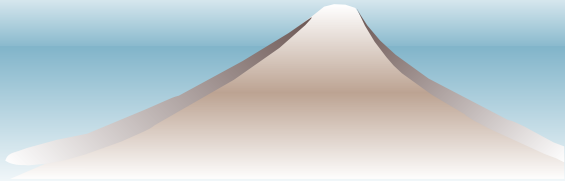


Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)

CやC++と同等（条件によってはそれ以上）に
速い！？



Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)

CやC++と同等（条件によってはそれ以上）に
速い！？

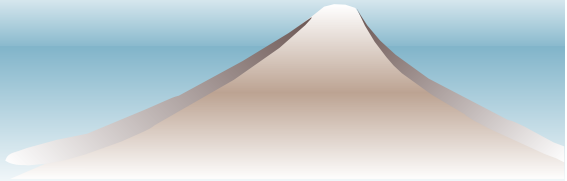
いったんおいておきます

Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)

プログラミング言語が信頼できる
（安全である）ってどういうこと…？



Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)

プログラミング言語が信頼できる
（安全である）ってどういうこと…？

プログラミング言語の安全性…？
ウウッ…頭が…（思い出している）!!

では安全な言語をどう作るか?

講義スライド

「安全性と型 / Safety and Types」より

- その前に「安全な言語」って何?
- 「この言語で書けばセキュリティホールのあるプログラムは決してできませんよ」は素晴らしいが，目標がやや野心的 (無理っぽ) すぎる
- ここでの「安全」の定義はもっと控えめなもの (memory safety)
  - ≈ わけのわからない挙動をしない
    - ★ 例: データはそのデータを変更することでしか書き換わらない
    - ★ C の場合: `a[i] = x` を実行したら変数 `y` が壊され得る
  - ≈ 早い話,「変な場所をアクセスしない」
- 以降「安全」はこの控えめな意味で使う

プログラミング言語が「(メモリ) 安全」であるためには...?

安全な言語の要件

「ポインタの dereference」の妥当性を保証する

- 配列の大きさをはみ出した添字でのアクセス (オーバーフロー) をしない
- 型安全 (配列とレコード，異なる種類のレコードを取り違えたアクセスをしない)
  - 動的 (実行時) 型検査; dynamic type check, runtime type check
  - 静的 (実行前・コンパイル時) 型検査; static type check
- 自動メモリ管理
  - すでに解放 (正確には再利用) されている領域へアクセスしない $\Rightarrow$ これからもアクセスされる可能性がある領域を再利用しない

以下は「型安全」の部分に限って議論する

安全な言語の要件

講義スライド
「安全性と型 / Safety and Types」より

「ポインタの dereference」の妥当性を保証する

- 配列の大きさをはみ出した添字でのアクセス (オーバーフロー) をしない
- 型安全 (配列とレコード，異なる種類のレコードを取り違えたアクセスをしない)
  - 動的 (実行時) 型検査; dynamic type check, runtime type check
  - 静的 (実行前・コンパイル時) 型検査; static type check
- 自動メモリ管理
  - すでに解放 (正確には再利用) されている領域へアクセスしない ⇒ これからもアクセスされる可能性がある領域を再利用しない

「型安全」と「メモリ管理」によって
「安全な言語」が形成される！

安全な言語の要件

「ポインタの dereference」の妥当性を保証する

- 配列の大きさをはみ出した添字でのアクセス (オーバーフロー) をしない
- 型安全 (配列とレコード，異なる種類のレコードを取り違えたアクセスをしない)
  - 動的 (実行時) 型検査; dynamic type check, runtime type check
  - 静的 (実[illegible]・コンパイル時) 型検査; static type check
- 自動メモ[illegible]
  - [illegible]は再利用) されている領域へアクセスしな[illegible]

まずは型安全の話！

Rustの「型安全」

「型安全」ってなんだっけ？
そもそも「型」ってなんだったっけ？

プログラミング言語における「型」

- データ型 (data type) または単に型 (type) という
- ≈ データの「種類」
- C の例
  - プリミティブ型: 整数 (char, short, int, ...), 浮動小数点数 (float, double, ...), ...
  - 複合型 (既存の型を組み合わせて作る型): ポインタ，配列，構造体，共用体，...

「型」があると何が嬉しいんだっけ？

型の役割

講義スライド
「安全性と型 / Safety and Types」より

1. 抽象化，隠蔽
2. 性能向上
3. エラーの検出

これらは具体的に何を意味するのだろうか？

型の役割 (1) 抽象化，隠蔽

講義スライド

「安全性と型 / Safety and Types」より

- 複数の値をまとめてひとつの値と見なせる (一つの変数で「持ち歩ける」) ようにする

```
1 typedef struct  {
2   float x; float y;
3 } vec2;
4 vec2 add(vec2 a, vec2 b);
```

- 型の「中身」をわかっていなくても使える

```
1 FILE * fp = fopen("foo.txt", "rb");
2 fread(buf, n, 1, fp);
```

⇒ 理解・再利用しやすいプログラム (部品)，変更しやすいプログラムの構造を作ることができる

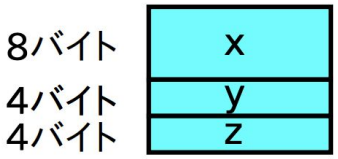
詳細な実装を常に意識しなくても，道具として使える！

型の役割 (2) 効率の向上

- コンパイル時に型がわかれば，ムダのないデータレイアウト，レジスタの利用，命令の生成が可能

```
1  typedef struct {
2    double x; int y; int z;
3  } DI;
4
5  double f(DI * di) {
6    return di->x + di->y + di->z;
7  }
```

⇒




```
1  cvtsi2sd 8(%rdi),%xmm0
2  addsd (%rdi),%xmm0
3  cvtsi2sd 12(%rdi),%xmm1
4  addsd    %xmm1,%xmm0
```

- x は di, y は di+8 [illegible] バイトにある
- y, z は [illegible] 変換する

事前にどのような大きさ・属性のデータが来るか分かれば
最適化することができる！

型の役割 (3) エラーの検出

値の「取り違え」を防ぐ (「型エラー (type error)」)

- 変数代入時の「取り違え」

```
1 char * s;
2 s = 195;  /* char* <- 整数 */
3 putchar(s[0]);    /* 整数の [0]って? */
```

- 関数呼び出し時の「取り違え」

```
1 fprintf("x=%d\n", x); /* FILE*にchar*を渡すことに */
```

きちんと型エラーを検出できるなら,
特定のバグは生じない

安全な言語の要件

「ポインタの dereference」の妥当性を保証する

- 配列の大きさをはみ出した添字でのアクセス (オーバーフロー) をしない
- 型安全 (配列とレコード，異なる種類のレコードを取り違えたアクセスをしない)
  - 動的 (実行時) 型検査; dynamic type check, runtime type check
  - 静的 (実行前・コンパイル時) 型検査; static type check
- 自動メモリ管理
  - すで[illegible]正確には再利用) されている領域へアクセスしな[illegible]アクセスされる可能性がある領域を再利用し

型検査を通過した(型エラーを吐かなかったとき)に
特定の不正動作をしないということか！

Rustの「型安全」

- 静的型付け言語
- 強い型付け
  - 型検査によって型安全性が保証される

Cのような「弱い型付け」の言語では，
（あってないような）型検査を通過しても，
型安全であるとは限らない

偉大なるwikipediaより

「型」に関するRustのトピック

- NULL安全性
- ゼロコスト抽象化

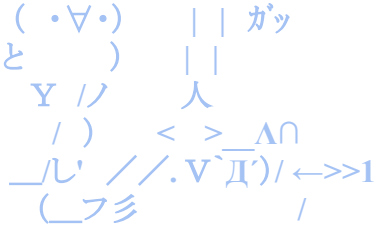
Rustと「NULL安全性」

NULL安全性

^__^
(´∀`)< ぬるぽ

- Rustによるセグフォ回避策の一つ
- C++で**頻発**する、NULL関連の実行時エラーを
  コンパイル時に検出する

NULL安全性

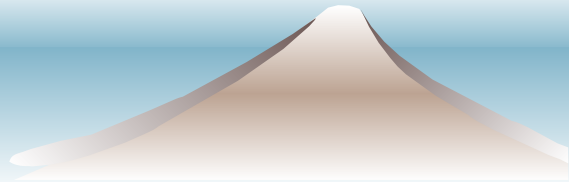


- NULL安全性自体は他言語にも存在する。
  - Kotlin,Python,Typescriptなど

- 型を利用してNULLに対する処理が行われないようにする

NULL安全性の仕組み

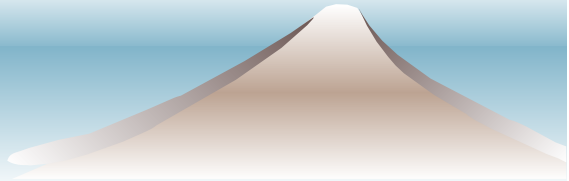
- RustにおけるNULL安全性の仕組みを説明する。



NULL安全性の仕組み

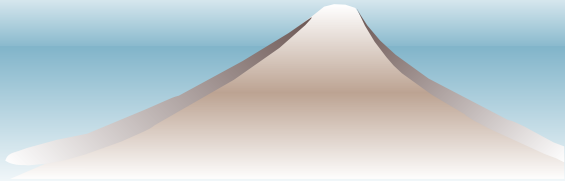
- 「NULLかもしれない変数」
  「確実にNULLではない変数」
  「NULLを表す変数」をそれぞれ別の型に分ける。

- 普段は、変数は「NULLかもしれない変数」の型で保持しておく。



NULL安全性の仕組み

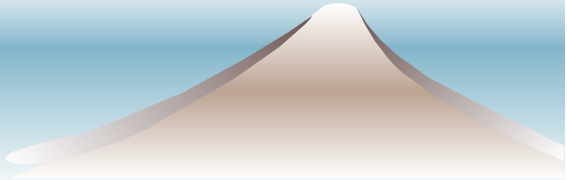
- メソッドやメンバ変数は、
  「確実にNULLではない変数」の型経由でしか
  呼び出せないようにする。
  - 「NULLかもしれない変数」や「NULLを表す変数」から
    メソッドや変数を呼び出すと、未定義要素の呼び出しによって
    型エラーが発生する。



NULL安全性の仕組み

- 「NULLかもしれない変数」に対して操作を行う際
  パターンマッチングで変数がNULLか判定する。

- パターンマッチング後、変数は
  「確実にNULLではない変数」「NULLを表す変数」
  のどちらかに型変換される。



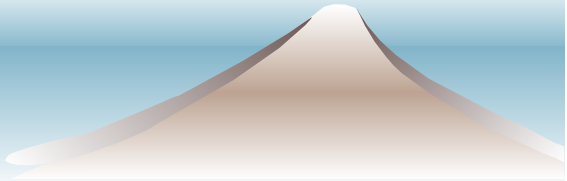
NULL安全性の仕組み

- マッチングの結果が「確実にNULLではない変数」であった場合には通常の処理を、「NULLを表す変数」であった場合はNULL用の処理を行う。

NULL安全性の仕組み

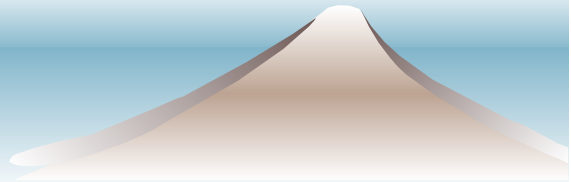
- 間違えて「NULLかもしれない変数」や「NULL」を経由してメソッドにアクセスすると型エラー。
  - C++で言うところのNULLチェック忘れ

- **厳格な（強い）静的型付け言語であるRustでは、型エラーは必ずコンパイル時に発生する。**

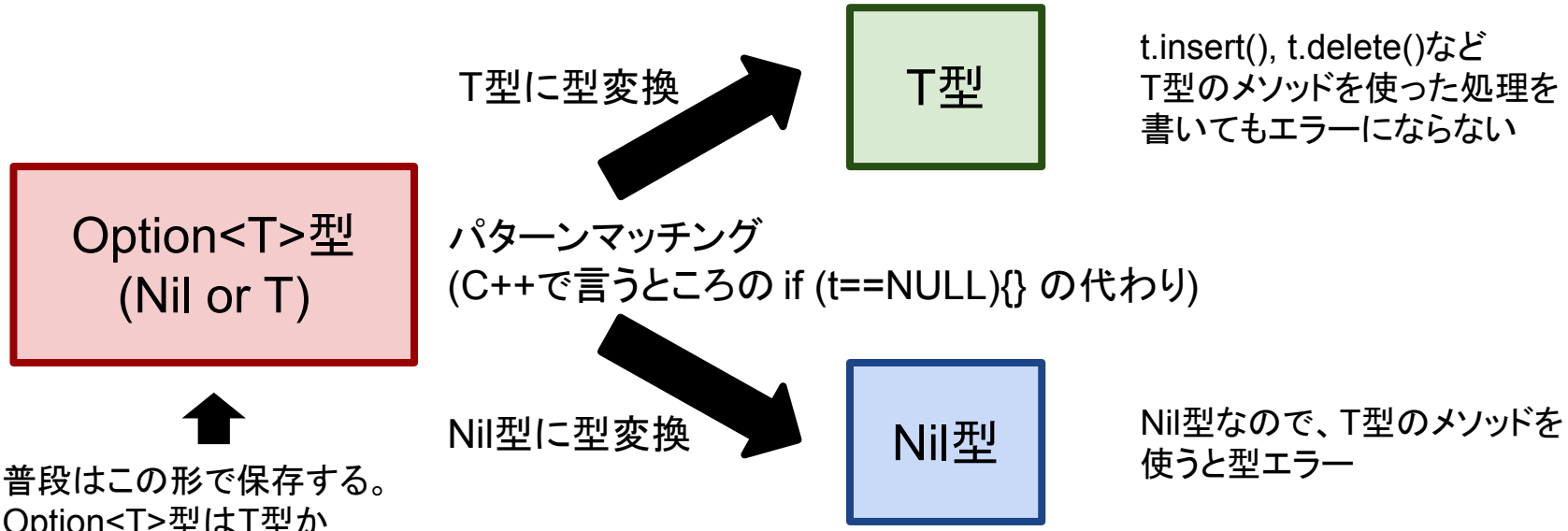


NULL安全性の実現: Rust

- 標準ライブラリのOptionという機能を使う。
- NilがNULLに該当するオブジェクトである。



Rustの場合(図解)



・普段はこの形で保存する。
・Option<T>型はT型か
Nil型のいずれかの値を
内包する型。
・T型のメソッドは使えない。
使うと型エラー

Rustと「ゼロコスト抽象化」

Rustと「ゼロコスト抽象化」

そもそもプログラミング言語における
「抽象化」ってなんだっけ？

「抽象化」とは

- 詳細を捨象して，一度に注目すべき概念を減らすこと

- プログラミング言語が提供する抽象化の例
  - ポリモーフィズム（多相性）
  - 高階関数

偉大なるwikipediaより

「抽象化」とは

- 詳細を捨象して，一度に注目すべき概念を減らすこと

- プログラミング言語が提供する抽象化の例
  - ポリモーフィズム（多相性）
  - 高階関数

なんじゃそりゃ！？

単純な静的型では不十分な場面

- 汎用的なデータ構造 (コンテナ)
  - 配列，リスト，キュー，スタック，グラフ，etc.

```
1 typedef struct { /* 可変長配列 (≈STL vector) */
2   element * a;
3   int n;
4   int sz;
5 } var_array;
```

- 汎用的なアルゴリズム
  - 整列，探索，グラフアルゴリズム，etc.

```
1 void sort(element * a, int n);
```

  - メモリコピー，メモリ管理

どの場合も,「プログラムの文面上ひとつの変数や式が，実行時には複数の型を持ちうる (多相性; polymorphism ポリモルフィズム)」ことが必要

ジェネリクス（generics）

- Rustが提供する多相性（抽象化）の機能の一つ
- 例えば，以下の関数は任意の型を引数に受ける generic function である

```
fn foo<T>(arg: T) { ... }
```

ほとんどのプログラミング言語で同じことができるね！

ジェネリクス（generics）

- Rustが提供する多相性（抽象化）の機能の一つ
- 例えば，以下の関数は任意の型を引数に受ける generic function である

```
fn foo<T>(arg: T) { ... }
```

ほとんどのプログラミング言語で同じことができるね！

「抽象化」とは

- 詳細を捨象して，一度に注目すべき概念を減らすこと

- プログラミング言語が提供する抽象化の例
  - ポリモーフィズム（多相性）
  - 高階関数

関数を受け取り，関数を返す関数！

「抽象化」とは

- 詳細を捨象して，一度に注目すべき概念を減らすこと

- プログラミング言語が提供する抽象化の例
  - ポリモーフィズム（多相性）
  - 高階関数

抽象化ってとっても便利！

「抽象化」とは

- 詳細を捨象して，一度に注目すべき概念
- プログラミング言語が提供する抽象化の
  - ポリモーフィズム（多相性）
  - 高階関数

抽象化ってとっても便利！

抽象化のコスト

```
fn foo<T>(arg: T) { ... }
```

- 例えば，様々な型に対応するために，動的割り当て（ディスパッチ）をする必要が生じる

- よって，抽象化しなかった場合に比べて，追加のコストが発生する

「明日，何かの科目の試験が一つあるから」と言われても...
科目が一つに絞られていたらマシ！

「ゼロコスト抽象化」とは

- 抽象化にあたり，余計な追加コストをゼロにする
  - 理想的な（必要最低限の）コストで抽象化する

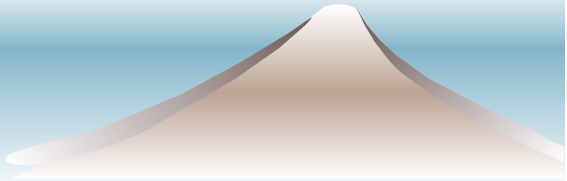
- 抽象化したらゼロコストになるわけでも，
  どんな時でもコストゼロで抽象化できるわけでもない

どうやって抽象化コストを理想化（削減）？？

抽象化コストの値切り方

- 静的ディスパッチ
- 型状態

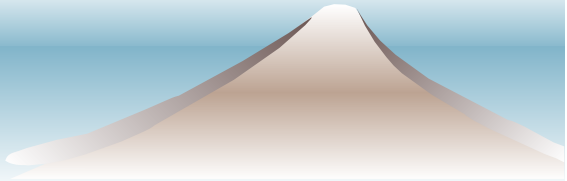
などなど



抽象化コストの値切り方

- 静的ディスパッチ
- 型状態

などなど



ディスパッチの種類

- 動的ディスパッチ
  - 正確な型は実行時に初めて確定する
  - インライン化（コンパイラによる最適化）できない
  - コードは膨張しないが低速な関数を実行する

- 静的ディスパッチ
  - 呼び出される関数（型）はコンパイル時に判明
  - インライン化（コンパイラによる最適化）できる
  - 同じ関数をいくつもコピー（具体化）する

具体的には？

静的ディスパッチ

```
fn foo<T>(arg: T) { ... }
```

↓

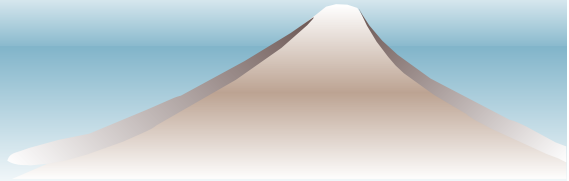
```
fn __foo_bool(arg: &bool) { ... }

fn __foo_i32(arg: &i32) { ... }
```

実際に使っている具体的な型を当てはめて，関数を具体化

C++のテンプレートのように，インライン展開してバイナリサイズが大きくなるイメージ！

Rustの「ゼロコスト抽象化」

- 抽象化のために必要なオーバヘッドが
  最低限になるよう，言語レベルでサポートしている！



Rustの「メモリ管理」

安全な言語の要件

講義スライド

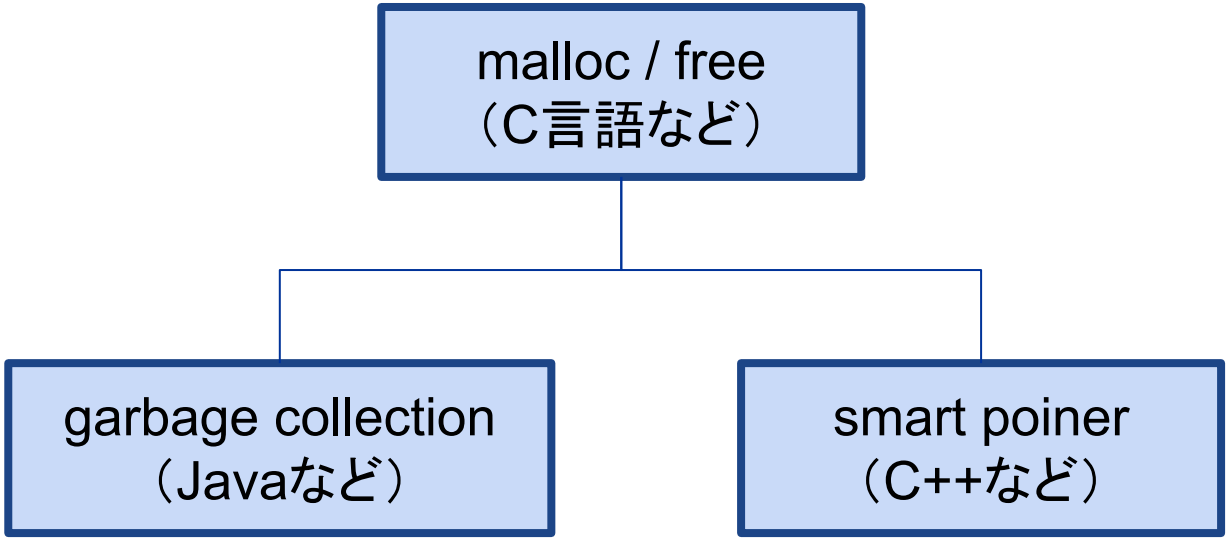
「安全性と型 / Safety and Types」より

「ポインタの dereference」の妥当性を保証する

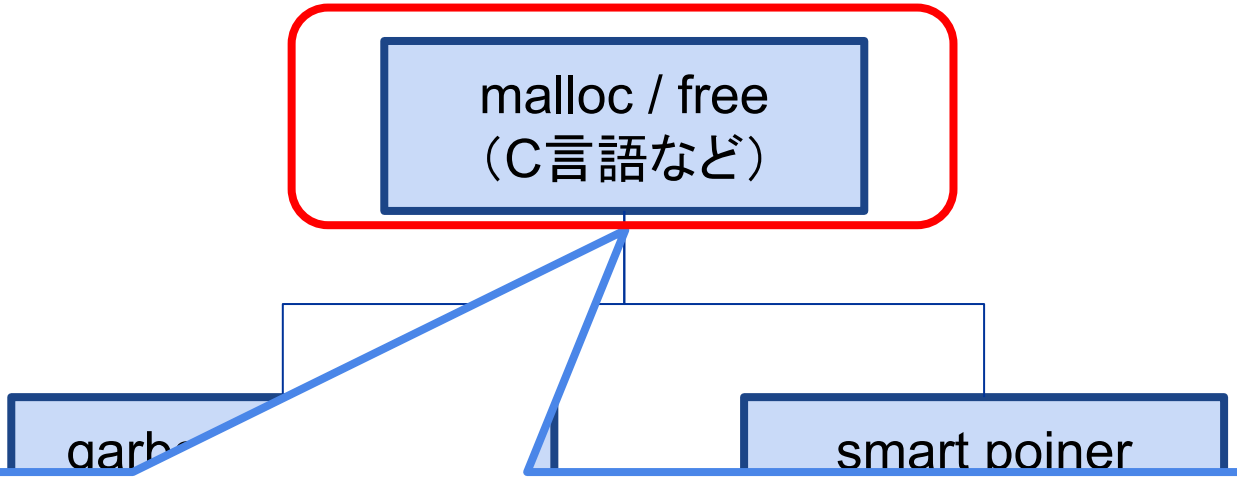
- 配列の大きさをはみ出した添字でのアクセス (オーバーフロー) をしない
- 型安全 (配列とレコード，異なる種類のレコードを取り違えたアクセスをしない)
  - 動的 (実行時) 型検査; dynamic type check, runtime type check
  - 静的 (実行前・コンパイル時) 型検査; static type check
- 自動メモリ管理
  - すでに解放 (正確には再利用) されている領域へアクセスしない $\Rightarrow$ これからもアクセスされる可能性がある領域を再利用しない

メモリ管理は，信頼性の高い(安全な)言語の要件！

メモリ管理のパラダイム



メモリ管理のパラダイム



プログラマの責任でメモリの確保・開放をする.
正しく制御すれば高速度・高効率だが,
現実に人間がミスせずメモリ管理するのは難しい.

起きうる間違い

講義スライド

「ガベージコレクション / Garbage Collection」より

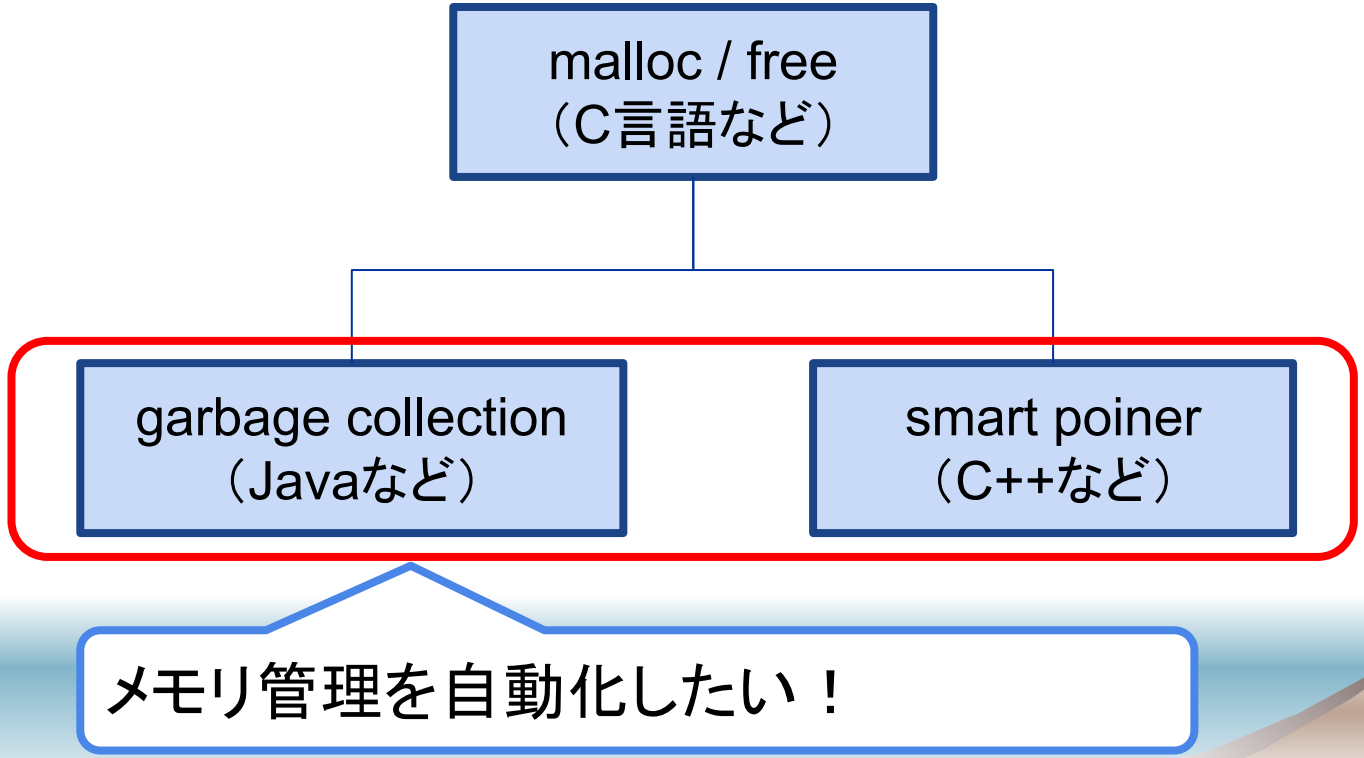
- 寿命を超えて変数をアクセスする
- ヒープから割り当てた変数を開放し忘れる (メモリリーク)

4. 寿命 (lifetime)

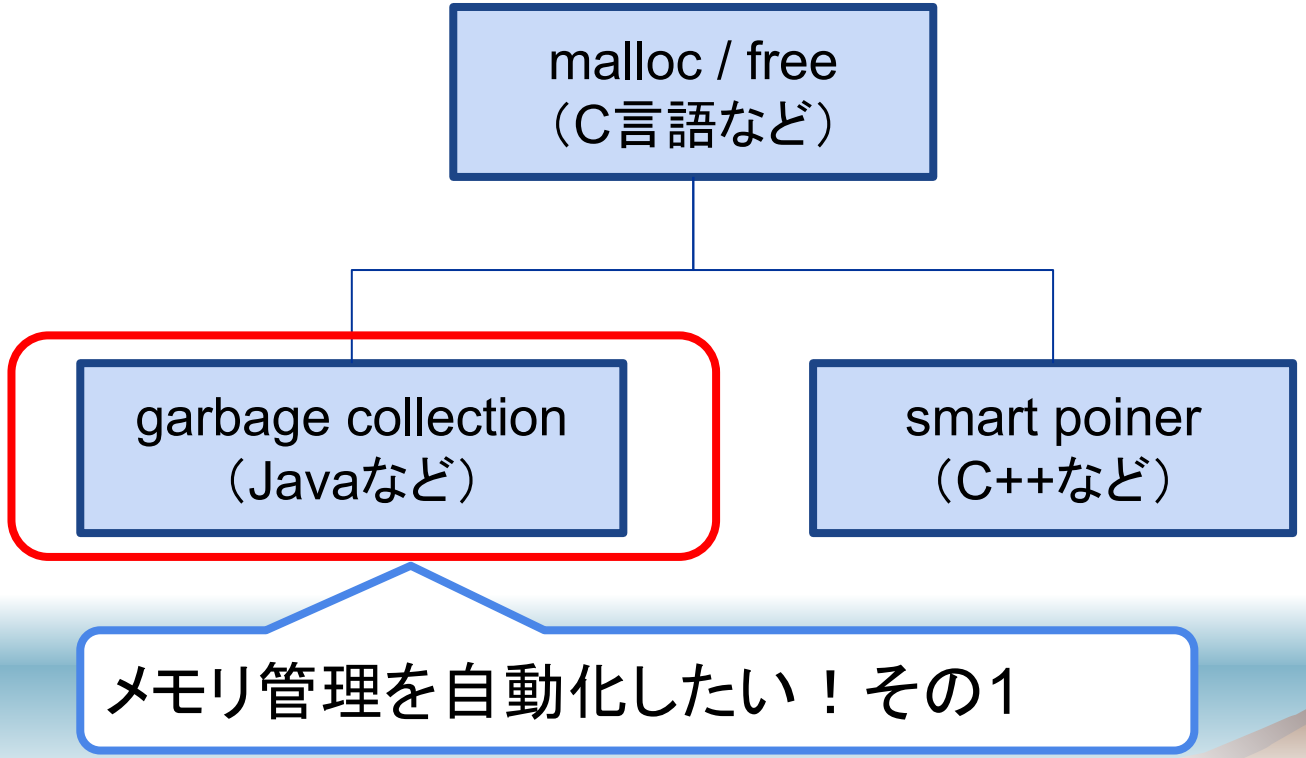
	開始	終了
大域	プログラム開始時	プログラム終了時
局所	ブロック開始時	ブロック終了時
ヒープ	malloc, new	free, delete

プログラマが一切のミスなく管理するのは難しい...

メモリ管理のパラダイム



メモリ管理のパラダイム



ガベージコレクション (Garbage Collection; GC)

講義スライド

「ガベージコレクション / Garbage Collection」より

- つまりは「寿命」と「アクセスする期間」が一致していないのが問題
  - 寿命後でもアクセスできてしまう
  - もうアクセスしないのに開放しない (生きっ放し)
- ⇒ ガベージコレクション (GC)
  - 今後アクセスされ得るものは残し，され得ないものは開放 (再利用) する
  - それを処理系が自動的に行う
  - ⇒ リークや，寿命後のアクセスによるメモリ破壊をなくす
  - C, C++, 古代の言語以外はほぼ搭載している
- 今後アクセスされ{得る・得ない}ものなんてなぜわかるのでしょう?

今後アクセスされない値をどうやって検出する？

2大GC方式

講義スライド

「ガベージコレクション / Garbage Collection」より

- 走査型 GC (traversing GC):
  - 素直にルートからポインタをたどり，ルートから到達可能なオブジェクトを発見
  - 発見されなかったものを回収
  - 2 タイプの走査型 GC
    - ★ マーク&スイープ GC (mark&sweep GC)
    - ★ コピー GC (copying GC)
- 参照カウント GC (reference counting GC):
  - あるオブジェクトを指すポインタの数 (参照数) を数えながら実行
  - 参照数が 0 になったものを回収
  - 注: 参照数が 0 → 到達不能

GCの欠点は何だろうか...？

GCの良し悪しの基準

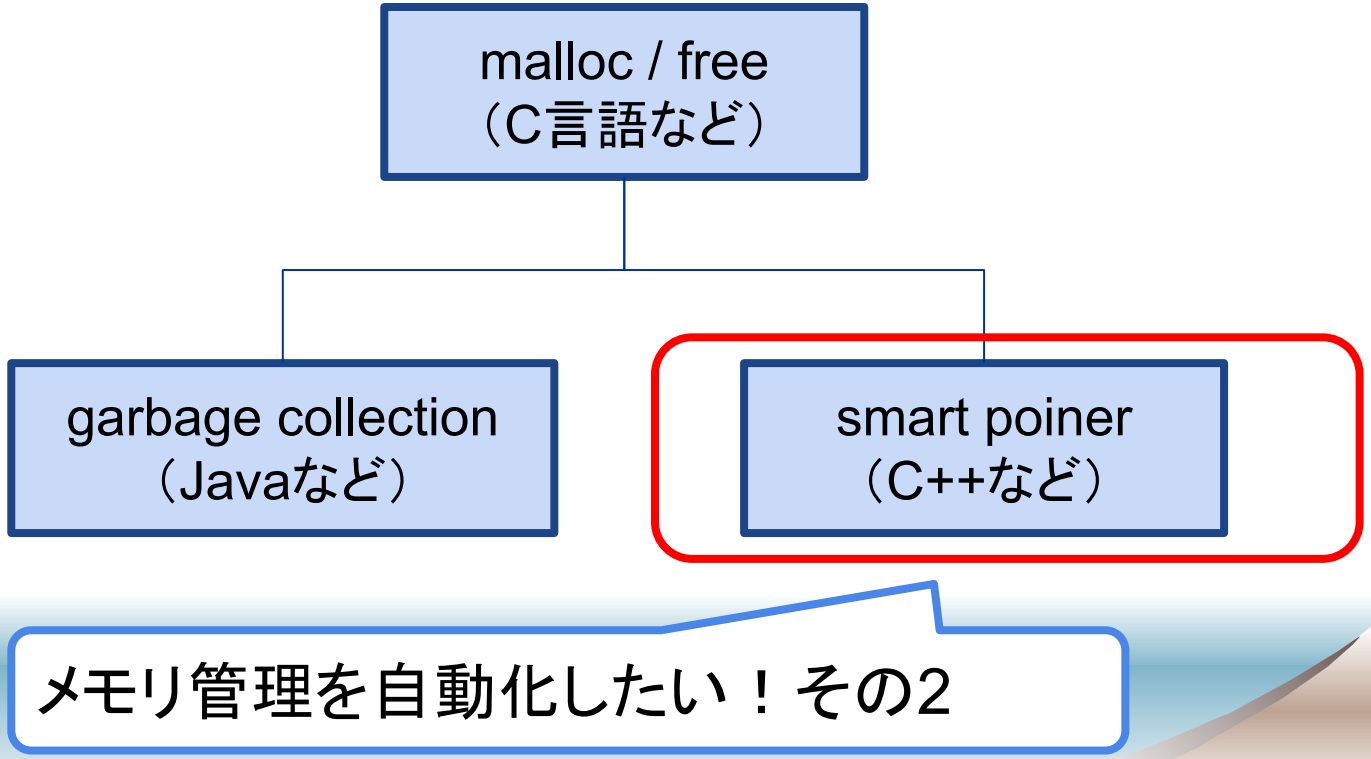
講義スライド

「ガベージコレクション / Garbage Collection」より

1. 正確さ:
   - 回収可能なゴミの範囲が広いか
2. メモリ割り当てコスト:
   - メモリ割当をするのに必要な (GC を含めた) 仕事
3. mutator オーバーヘッド:
   - GC が機能するために mutator に課されるオーバーヘッドが少ないか
4. 停止時間 (pause time):
   - GC が機能するために mutator が (一時的に) 停止しなくてはならない時間が短いか

GC動作のために，プログラムが一時的に停止したり
プログラムの実行にオーバヘッドが生じたりする

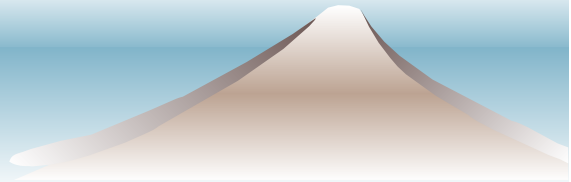
メモリ管理のパラダイム



スマートポインタとは

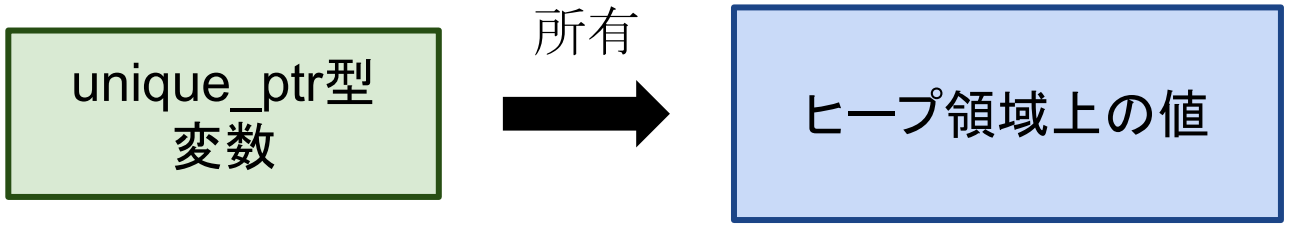
- 動的に確保したメモリを自動的に開放するポインタ

- 例: (C++11~)
  - std::unique_ptr
  - std::shared_ptr
  - std::weak_ptr



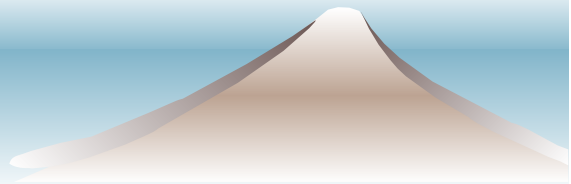
C++11 unique_ptr

- ヒープ領域を「所有」するポインタクラス。



C++11 unique_ptr

- unique_ptr型変数が所有しているヒープ領域は、変数がスコープから抜けた時に解放される。
  - メモリの開放忘れが起きない
- 複数のunique_ptr型変数が同じ領域を所有する事はない。
  - 一度開放したメモリが2回解放されない



C++コード例

```
#include <memory>
#include <iostream>
int main(){
    {
        std::unique_ptr<int> ptr(new int(0));
        while (*ptr<10){
            std::cout << *ptr << std::endl;
            ++(*ptr);
        }
    }
}
```

int型の領域をヒープに確保し、
unique_ptr型の変数ptrと紐づける。
(初期値0)

ptrの指す値(*ptr)を用いて0~9を表示

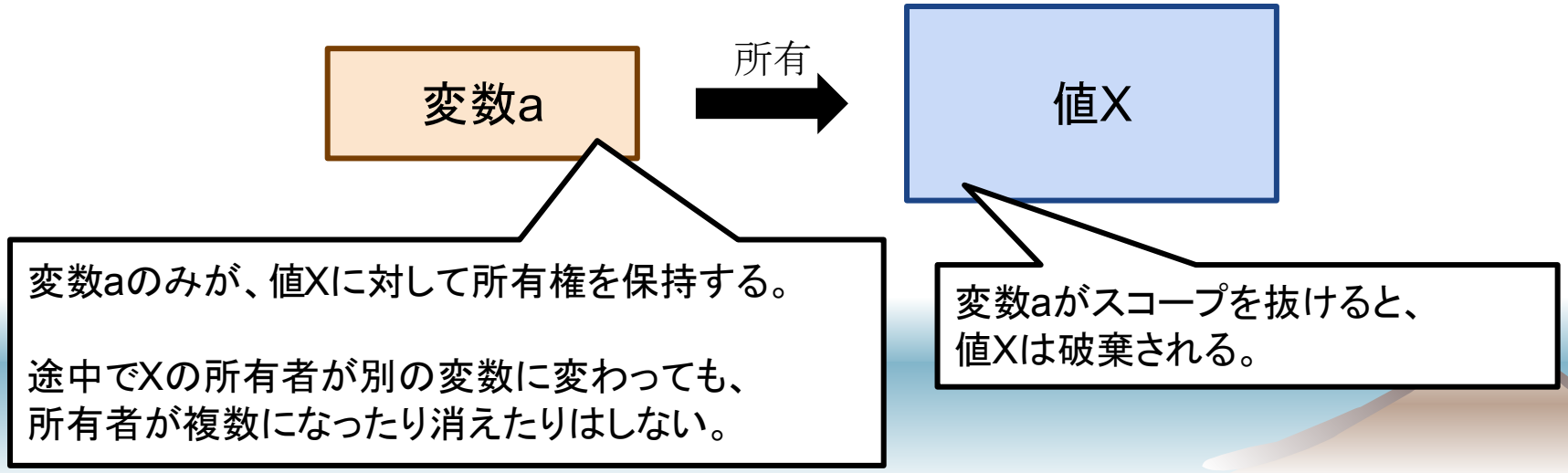
変数ptrのスコープが終了。
紐づけられたヒープ領域は解放される。

:ptrのスコープの範囲

Rustの「所有権」

所有権規則(Ownership Rules)

- Rustの各値は、所有者と呼ばれる変数と対応している。
- いかなる時も所有者は一つである。
- 所有者がスコープから外れたら、値は破棄される。



所有権について: 例

```
fn main(){
    let a = 100;
    {
        let b = a+23;
        println!("{}",b);
    }
}
```

変数aの宣言。初期値は100。
変数aは、メモリ上の値(100)に対し所有権を得る。

変数bの宣言。初期値はa+23=123。
変数bは、メモリ上の値(123)に対し所有権を得る。

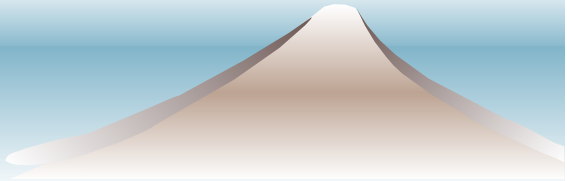
変数bの値を表示する。

変数bのスコープが終了。
bの所有するメモリ上の値(123)が破棄される。

変数aのスコープが終了。
aの所有するメモリ上の値(100)が破棄される。

所有権で一番重要な点

- メモリの所有者は常に単一の変数である
- メモリの所有者がスコープを出て無効になるタイミングと、メモリが開放されて無効になるタイミングが常に等しい。



多重解放

- 一度開放したメモリを再び開放すること

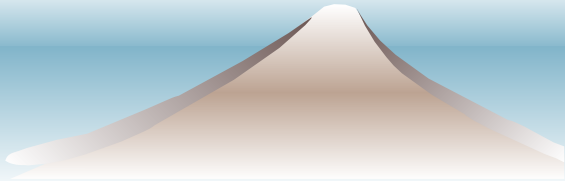
- Rustでは多重解放は起きない
  - 多重解放が起きるためには、メモリの所有者が2回スコープを出なければいけない

メモリの所有者は常に1つなので、
そのような事は起きない

解放済みメモリアクセス

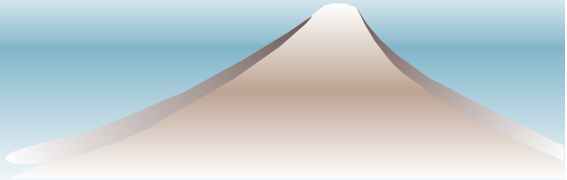
- Rustは解放後のメモリにアクセスしない
  - 変数のスコープと同様に、メモリの生存期間は
    コンパイル時に確定する。

  - 所有者に対する参照がメモリの生存期間を超えている場合、
    コンパイルエラーとして検出できる。

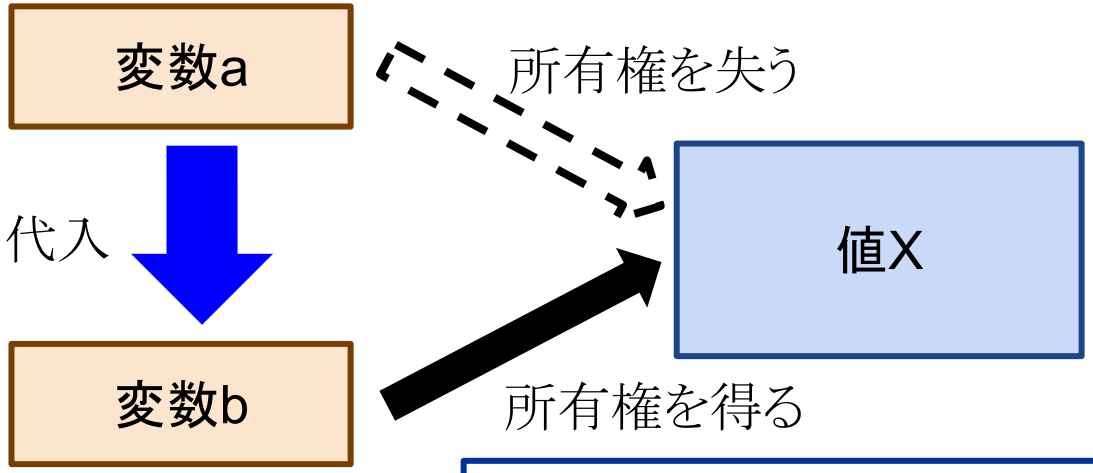


所有権の移動

- 同じメモリを所有する変数は1つなので、
  あるメモリに対して所有権を持つ変数を別の変数に
  代入すると、メモリの所有権ごと移動する。
  - shallow copyに近い
  - 数値型などは代入時にコピーされるため除く
- 元の変数は所有権を失うので、無効になる。



所有権の移動



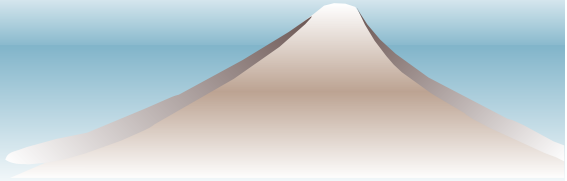
こういった現象は、String型などの代入時に
コピーが発生しない変数型で生じる。
関数の引数に直接変数を渡した場合も、
代入と同じ挙動を示す。

借用規則と並列処理

並列処理における安全性

- 最優先事項: データの読み書きが衝突しない

- 処理の衝突は未定義動作を引き起こす
  - 同時に書き込む、書き込み中に読み出す　などが原因

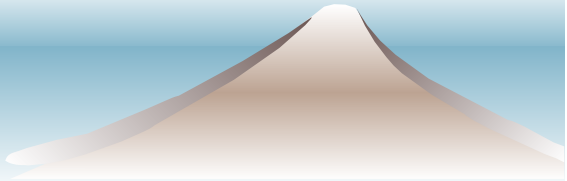


Rustの並列処理

- Rustには変数の読み書きに厳密な規則が存在する
  - 値の読み書きが衝突すること(データ競合)を防ぐ
  - 並列処理におけるメモリの安全性が高まる

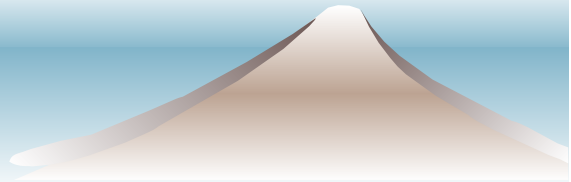
可変性と借用規則

- 可変性
  - ある値を途中で書き換えて良いかどうか
- 借用規則
  - データ競合を起こしうる参照関係を発生させない規則



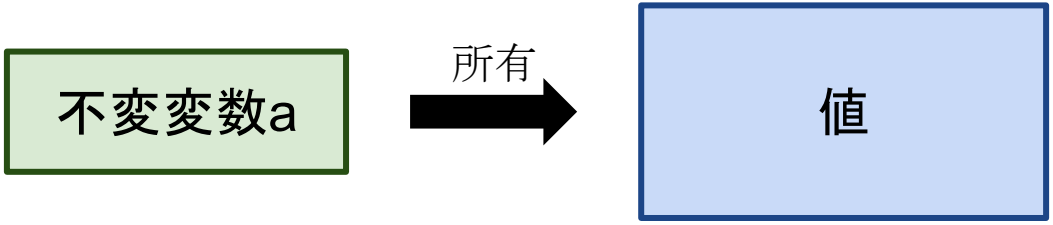
変数の可変性

- Rustの変数には、可変変数と不変変数がある
  - cf.) 可変参照・不変参照



不変変数

不変変数は値の初期化以降、変更が禁止される



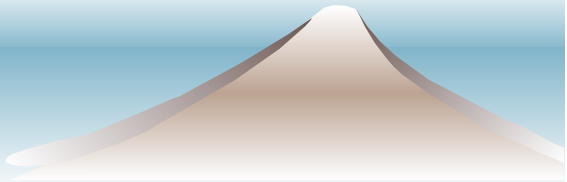
let a=100; → 100で初期化され、以降変更禁止

a=200; ← 初期化後に値を変更したのでコンパイルエラー

(実装)不変変数の宣言

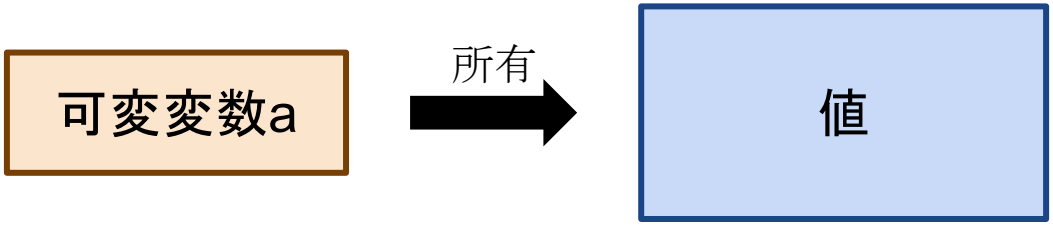
不変変数は以下のいずれかで宣言する
let『変数名』:『型名』=『初期値』;
let『変数名』=『初期値』;

例:
let x: i64 = 100; //i64型の不変変数xを宣言し、100で初期化
let x = 100;　　//i32型の不変変数xを宣言し、100で初期化
　　//xの型は初期値から推論されてi32型になる



可変変数

可変変数は初期化後も値を変更してよい



let mut a=100; → 100で初期化される

a=200; → 200が代入される

a=300; → 300が代入される

(実装)可変変数の宣言

可変変数は以下のいずれかで宣言する

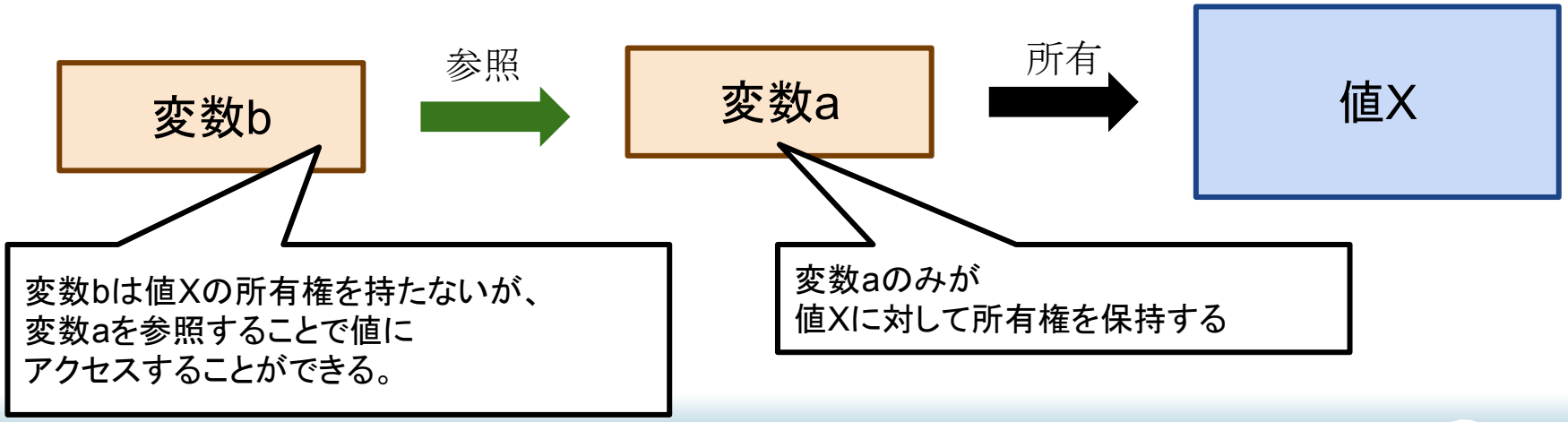
let mut『変数名』:『型名』;
let mut『変数名』:『型名』=『初期値』;
let mut『変数名』=『初期値』;

例:

```
let mut x: f64;          //f64型の可変変数xを宣言
let mut x: i64 = 100; //i64型の可変変数xを宣言し、100で初期化
let mut x = 100;         //i32型の可変変数xを宣言し、100で初期化
```

参照について

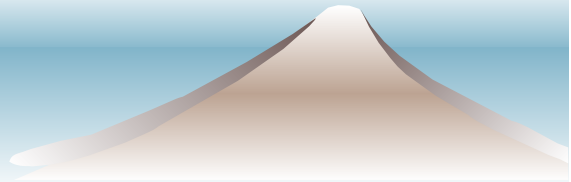
変数の参照をすることで、所有権を持つ変数以外から
値にアクセスできる



参照の可変性

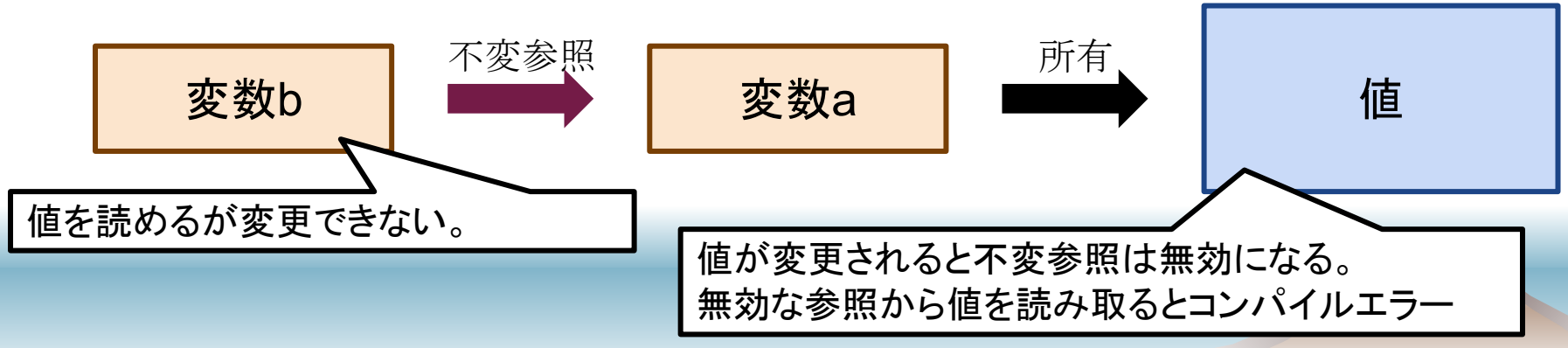
- Rustの参照には、可変参照と不変参照がある
  - cf.) 可変変数・不変変数

- 参照の可変性は変数の型で管理される。
  - つまり可変参照型と不変参照型がある



不変参照

- 不変参照は、参照先の値を変更できない参照である。
- プログラマは参照値の不変性を保証しなければならない。
  - 保証されないコードはコンパイルエラーとなる。



(実装)不変参照の宣言

```
let y =&x;        //変数xに対する不変参照を宣言する
                  //yはxの型への不変参照型をもつ不変変数となる
let mut y =&x; //変数xに対する不変参照を宣言する
                  //yはxの型への不変参照型をもつ可変変数となる
```

形を明示したい場合は以下のようにする。

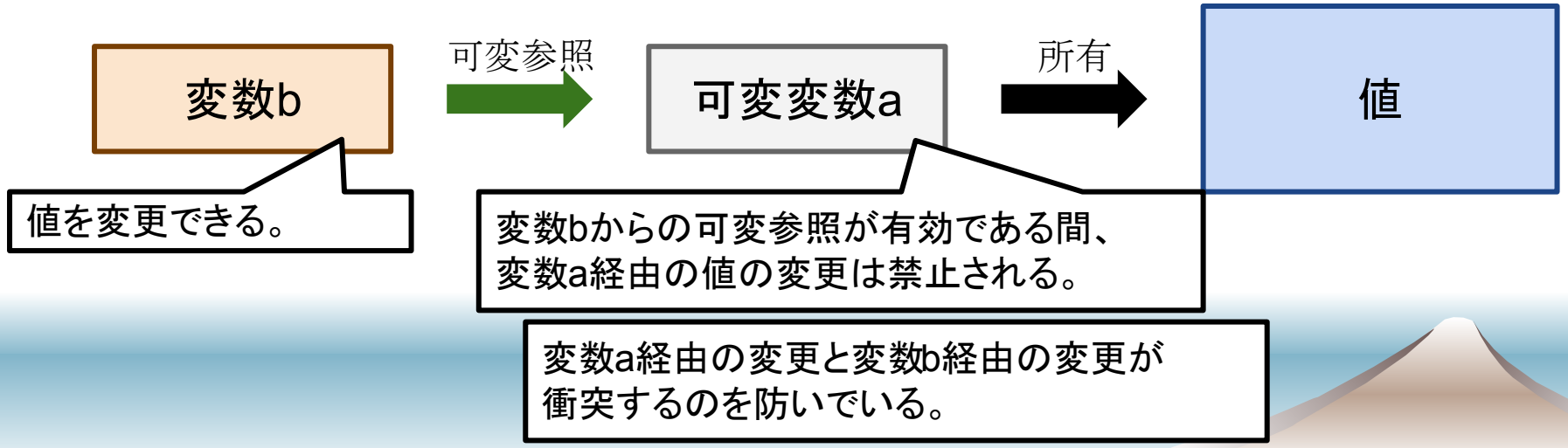
```
let y : &『xの形』=&x;
```

例:

```
let y : &i64 = &x; //yはi64型への不変参照型をもつ不変変数
```

可変参照

- 可変参照は、参照先の値を変更可能な参照である。
- 参照先の変数は可変変数でなければならない。
- 参照先の変数は、一時的に値を変更する権利を失う。



(実装)可変参照の宣言

```
let y =&mut x;        //変数xに対する不変参照を宣言する
                      //yはxの型への可変参照型をもつ不変変数となる
let mut y =&mut x; //変数xに対する不変参照を宣言する
                      //yはxの型への可変参照型をもつ可変変数となる
```

形を明示したい場合は以下のようにする。

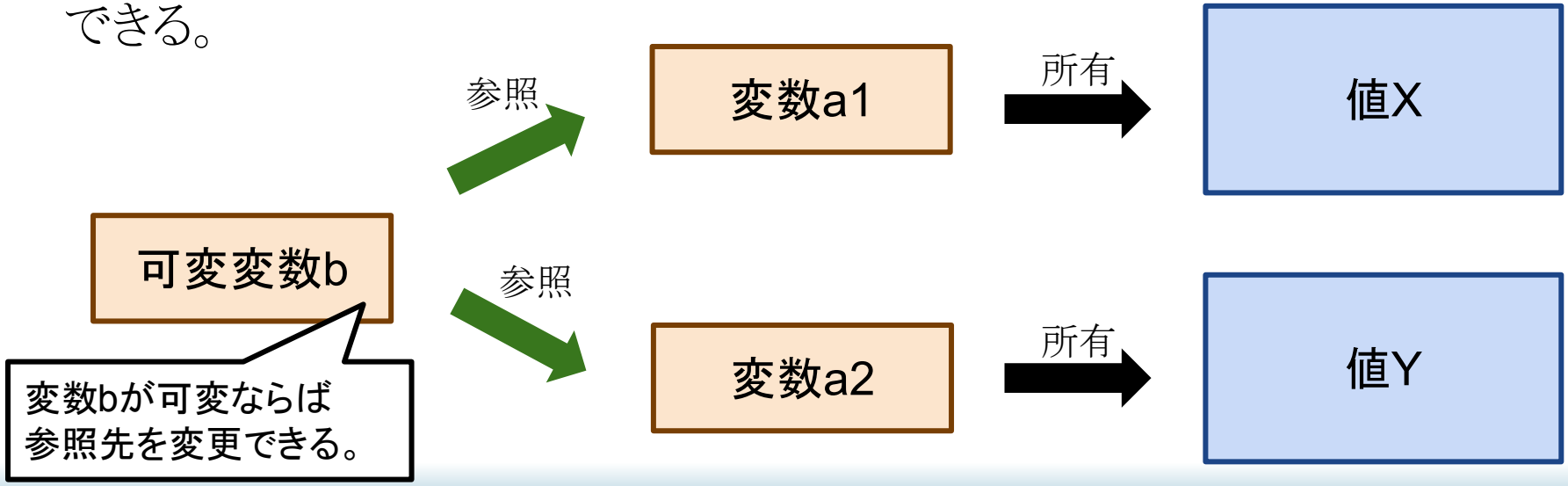
```
let y : &mut『xの形』=&mut x;
```

例:

```
let y : &mut i64 = &mut x;
                      //yはi64型への不変参照型をもつ不変変数
```

参照変数の可変性

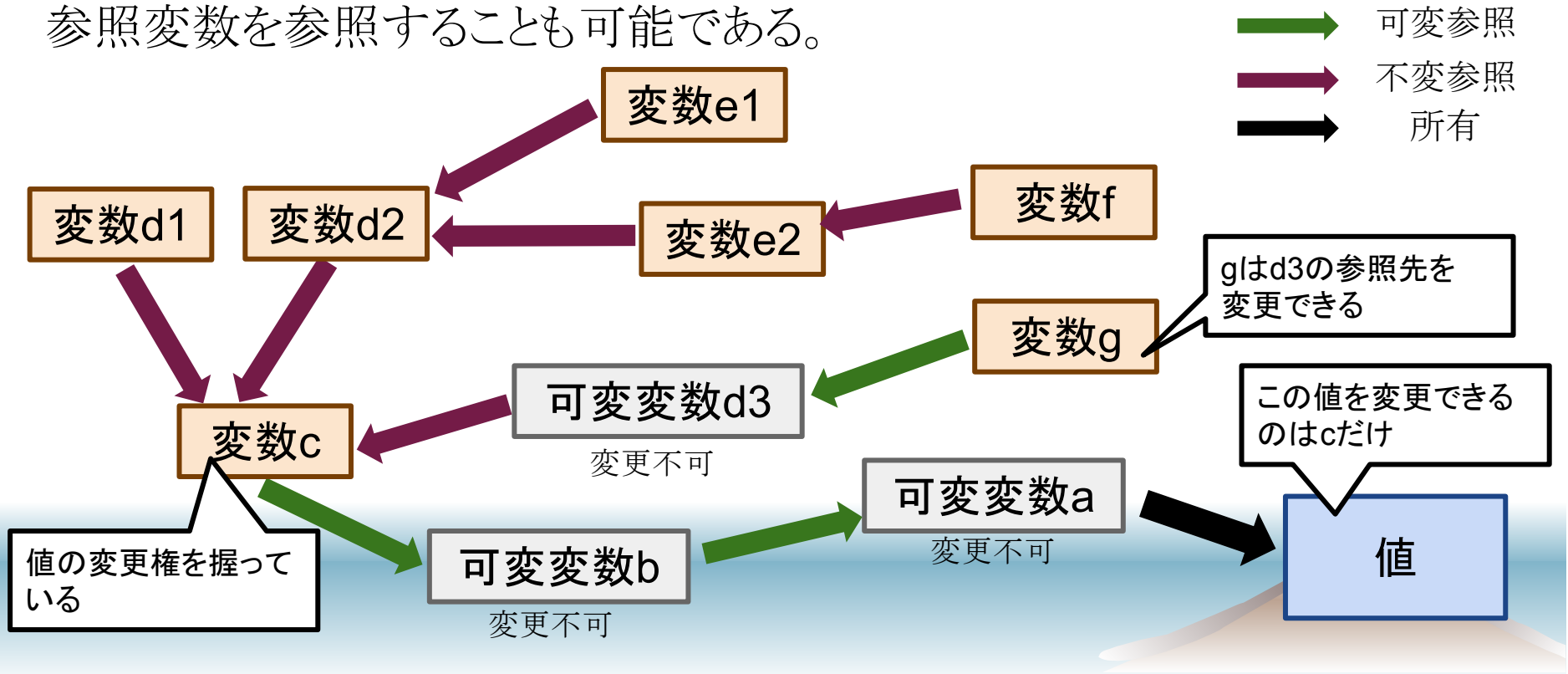
参照に使う変数が可変ならば、「参照先を」変更することができる。



注: 変数の可変性と参照の可変性は別

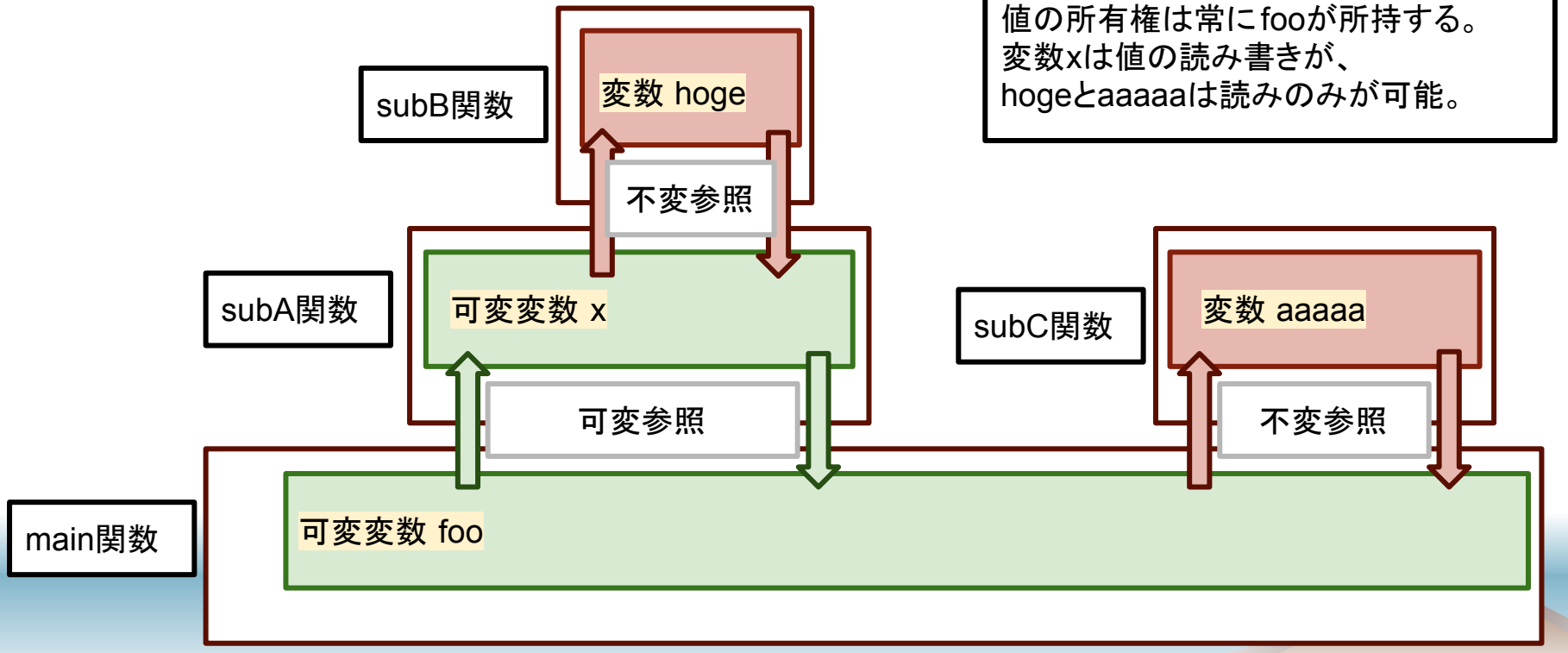
参照の参照

参照変数を参照することも可能である。



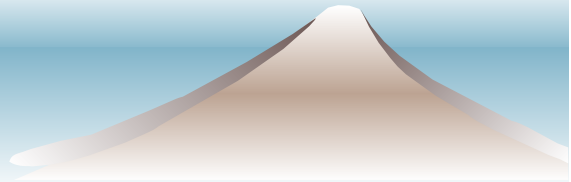
例: 関数の引数を参照で渡す

値の所有権は常にfooが所持する。
変数xは値の読み書きが、
hogeとaaaaaは読みのみが可能。



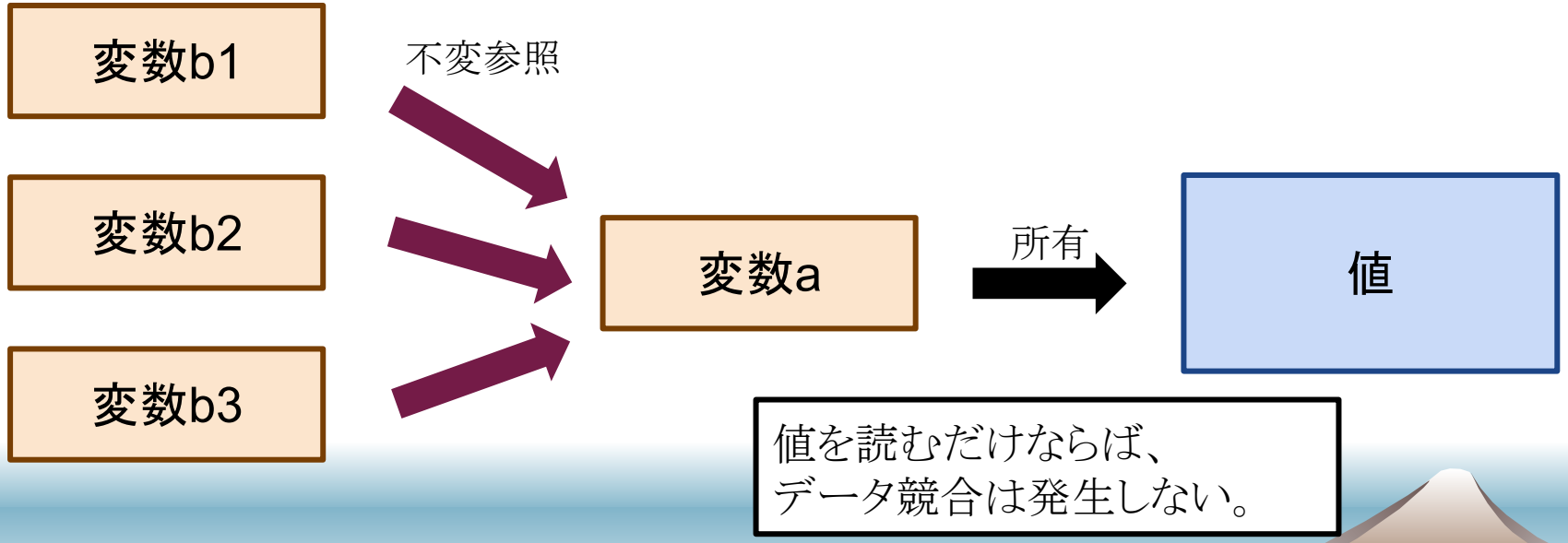
借用規則(Borrowing Rules)

- データ競合を防ぐために、
  全ての参照は借用規則に従う。
  - データ競合とは、同一の値に対する読み書きが衝突すること
  - データ競合は未定義動作の原因となる。



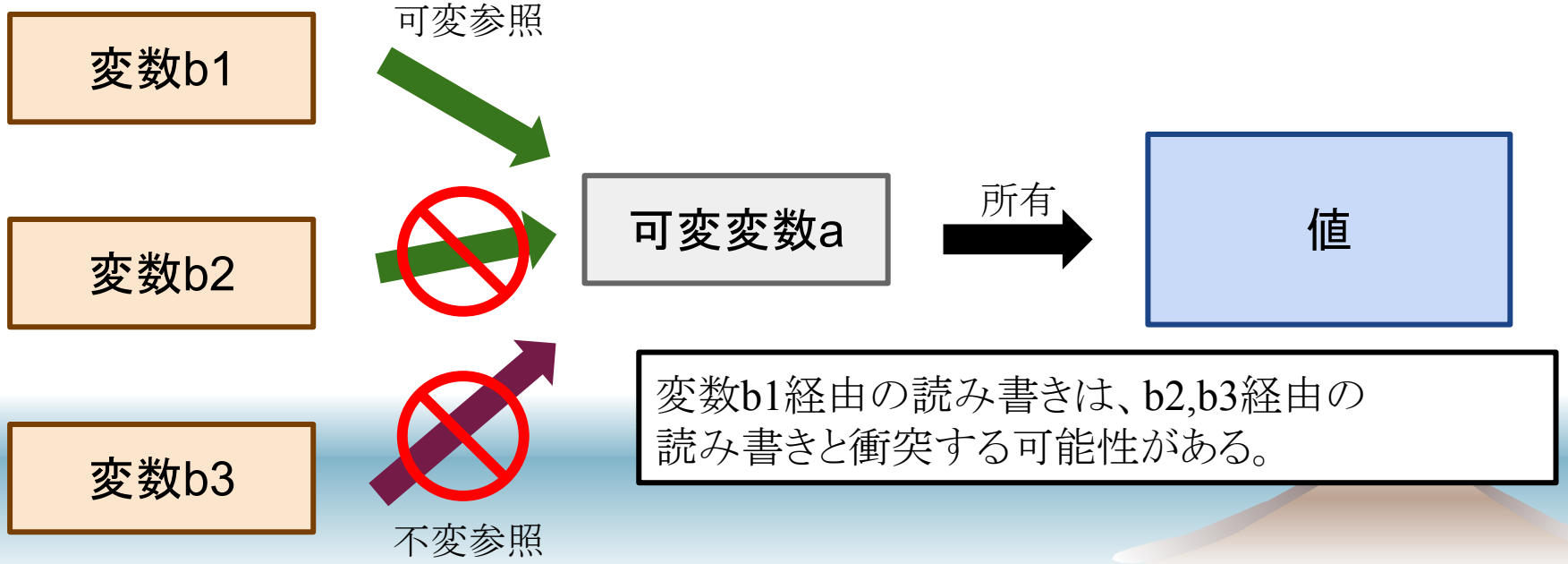
借用規則

不変参照は、同一の変数に対して複数定義してよい。



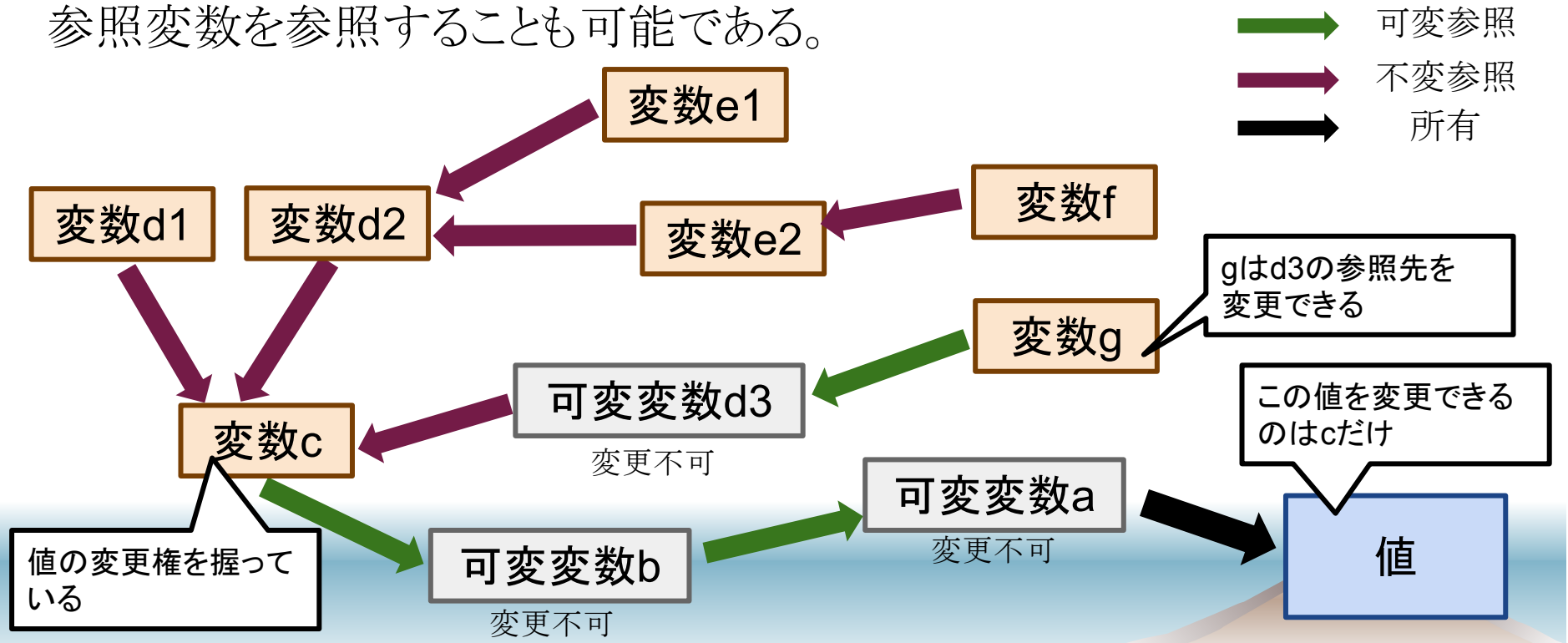
借用規則

可変参照は、同一の変数に対して同時に複数定義できない。
不変参照と同時に定義することも禁止である。



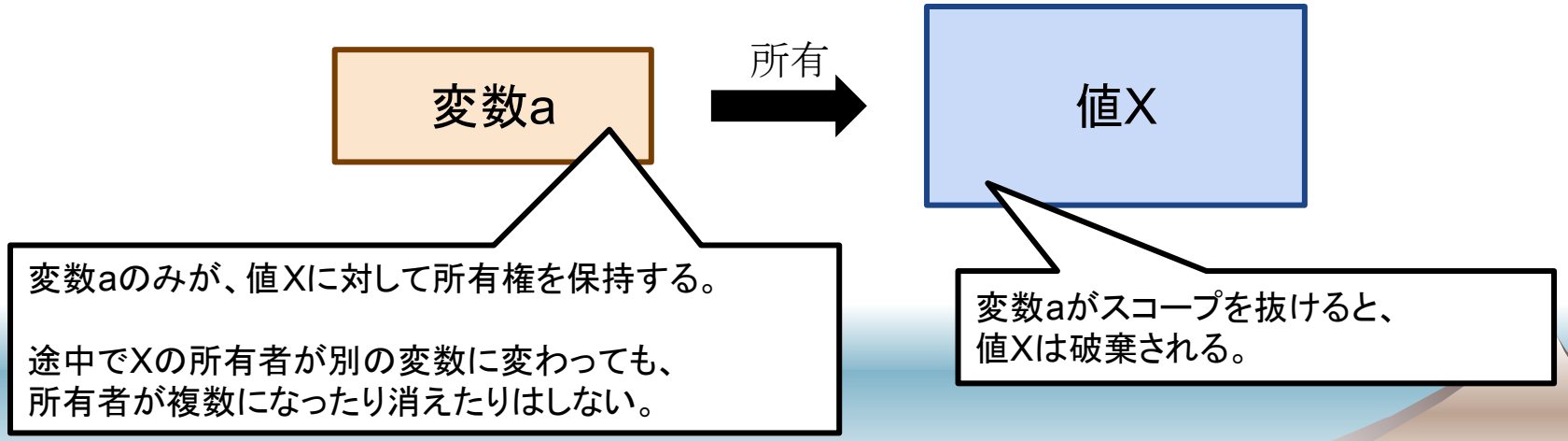
参照の参照　再

参照変数を参照することも可能である。



所有権規則(Ownership Rules) 再

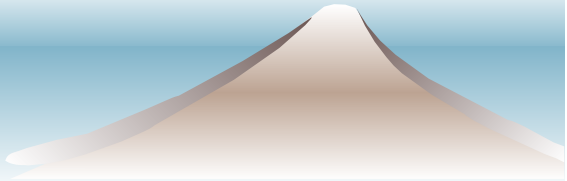
- Rustの各値は、所有者と呼ばれる変数と対応している。
- いかなる時も所有者は一つである。
- 所有者がスコープから外れたら、値は破棄される。



安全性の保証

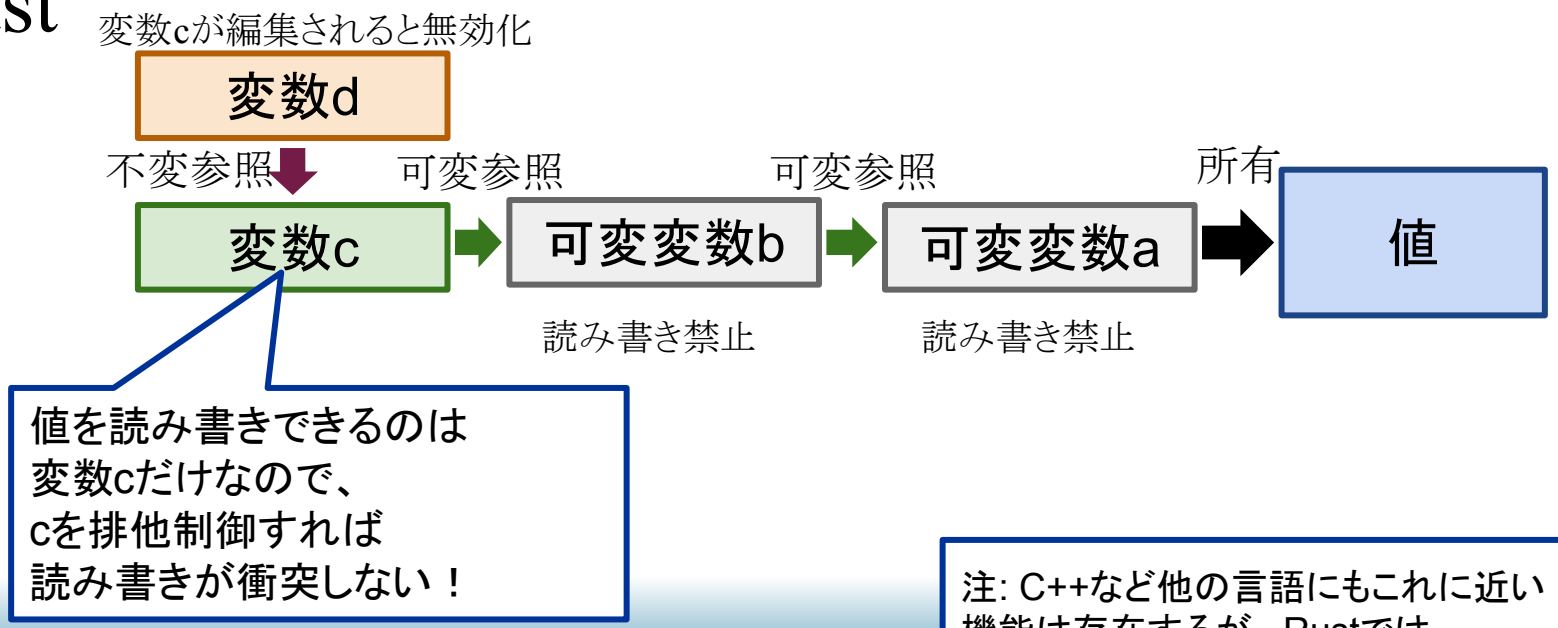
- 所有権規則と借用規則によって、以下が保証される。
  - プログラム上の全ての値が、任意のタイミングで
    ちょうど1つの変数を介してのみ変更される。

- 各変数に対する書き込みを個別に制御するだけで
  データ競合が起きなくなる。



安全性の保証

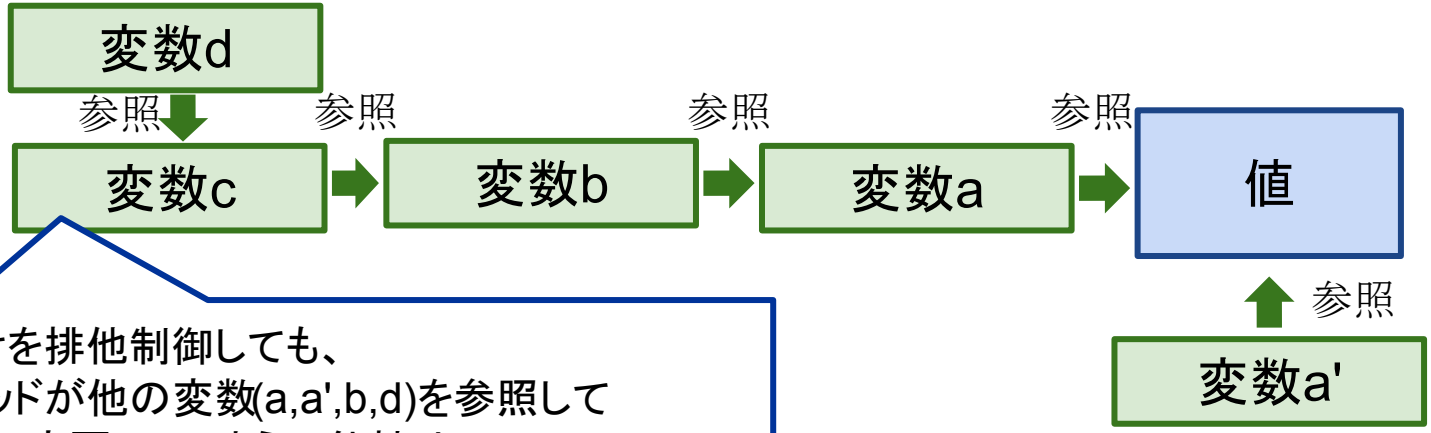
Rust



注: C++など他の言語にもこれに近い機能は存在するが、Rustでは言語レベルで実装されている。

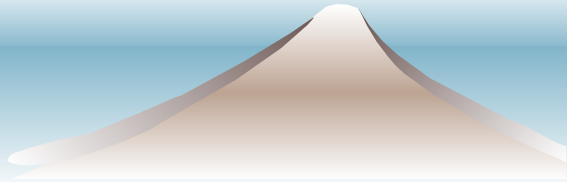
安全性の保証

一般の言語

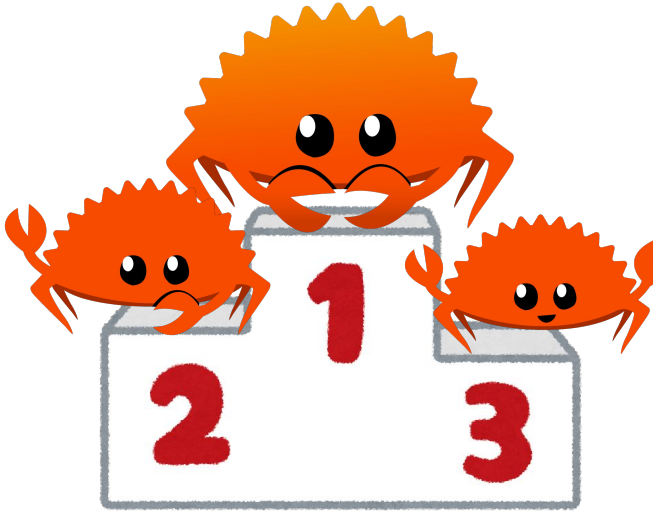


変数cだけを排他制御しても、
他のスレッドが他の変数(a,a',b,d)を参照して
値を同時に変更してしまう可能性や
値の更新中に読んでしまう可能性がある。

Rustの弱点



Rust最強論



- Rust vs C++ vs その他
  - Rust: 実行時に不正なアクセスが起きにくい。

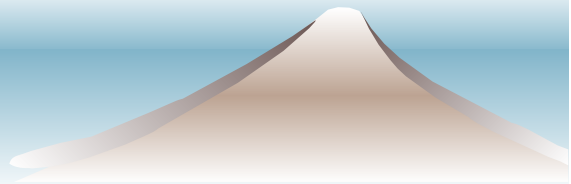
    並列処理にも強い。

    C++と同じぐらい速い。
  - C++: `Segmentation fault: 11`
  - その他の言語: C系の言語よりだいたい遅い だP遅

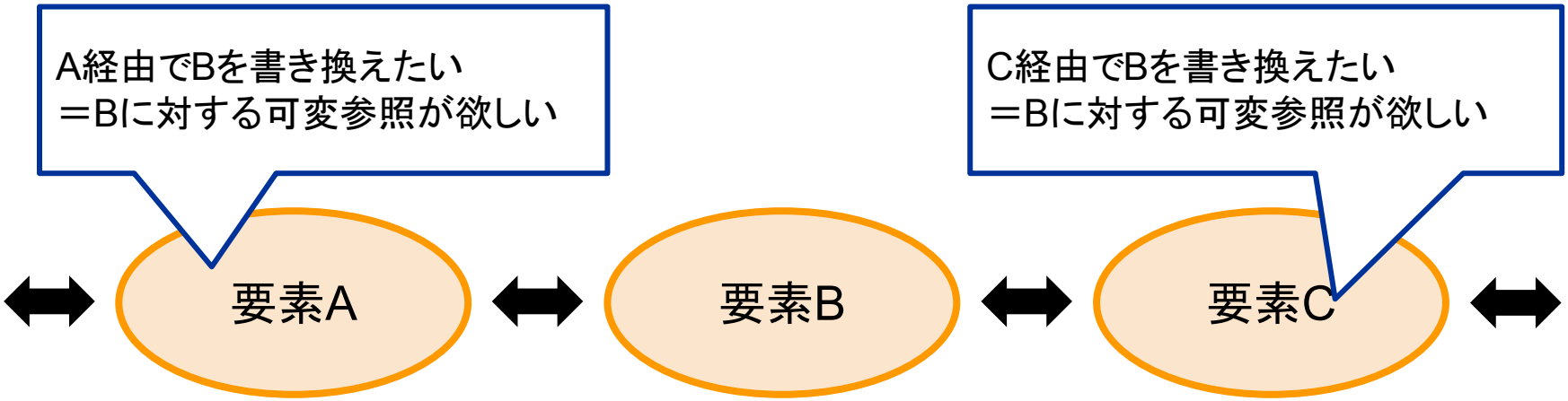
借用規則の欠点

- 同一のオブジェクトに対して、
  複数の変数が同時に可変参照を持てない
  - 可変参照は対象を書き換えるのに必須

- グラフや双方向連結リストが苦手！
  - C++では構造体+ポインタで実装可能
  - Rustではどうする？



例：双方向連結リスト

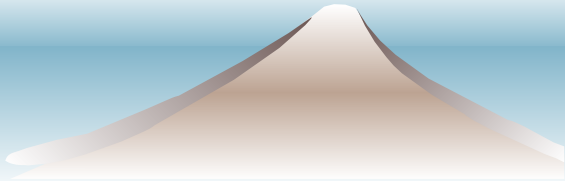


Bに対する可変参照を、
AとCのどちらが持つかが対立する！

参照のおさらい

- 借用規則を満たすならば、
  全ての値は所有権を有する変数と、
  その可変参照からしか変更できない。

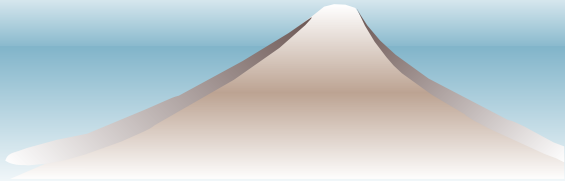
- 同じ値に対する複数の可変参照を同時に作れない
  - 詰んだ



そもそも

- 借用規則が厳しく複雑なのは、データ競合を
  コンパイル時に検出するため

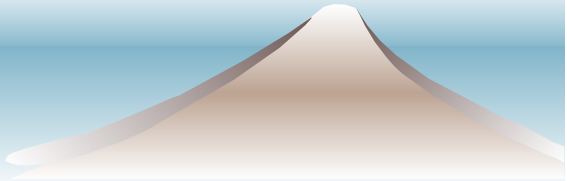
- 妥協して、コンパイル時ではなく実行時に
  データ競合を監視すればよい



荒技

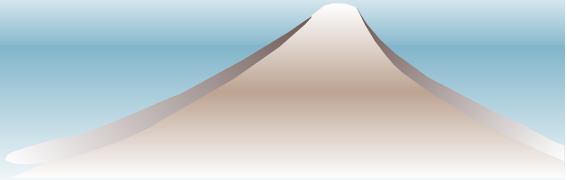
- Rustの標準ライブラリに存在する、Rc,Refcellという2種類のスマートポインタ+αを用いる

- これらは所有権規則と借用規則を部分的に無視する



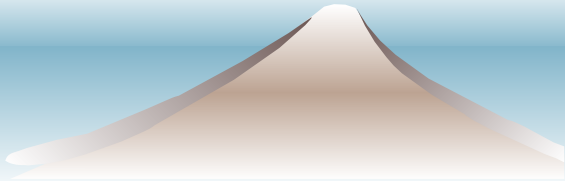
スマートポインタ

- Rustには標準ライブラリにスマートポインタが用意されている
- スマートポインタとは、不適切な処理が行われないように工夫されたポインタのこと



スマートポインタ(1):Box

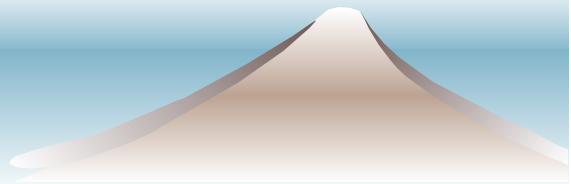
- Boxはヒープ領域に対して単一の所有権を持つ。
  - 同時に複数のBoxが同一のヒープ領域を所有しない。
- 特徴の無い、普通のスマートポインタ
  - c++の std::unique_ptr に近い



スマートポインタ(2):Rc

- Rcは、同一のヒープ領域を
  複数のスマートポインタが所有することを認める。
  - BoxとRefcellはできない
  - C++のstd::shared_ptrにやや近い

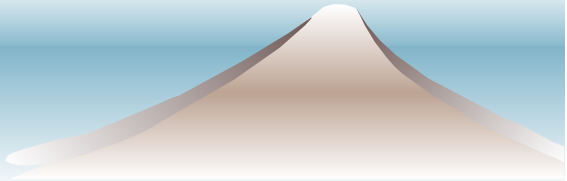
- ただし、所有している領域の値を変更できない。
  - BoxとRefcellはできる



スマートポインタ(2):Rc

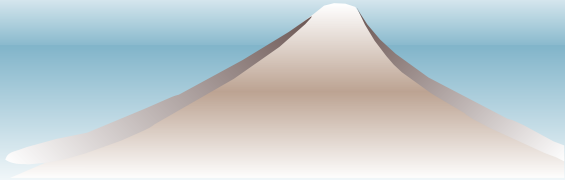
- 参照カウント方式のため、参照ゴミに弱い。

- が、Rustでは参照ゴミによるメモリリークは
  メモリ安全ということになっているので問題ない
  - メモリリークは開放したメモリの再利用よりははるかに安全



スマートポインタ(3):Refcell

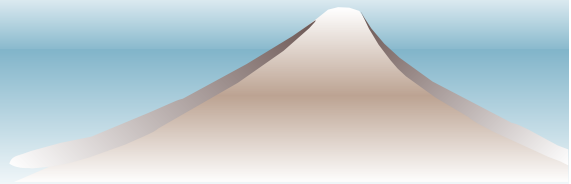
- Refcellはヒープ領域に対して単一の所有権を持つ。
  - この点ではBoxと同じ
- Refcell自体が可変か不変かによらず、半強制的に中の値を書き換えられる。(内部可変性)
  - 代わりに、借用規則のチェックを実行時に行う。
    そのためコンパイル時に規則違反を検出できない。



Rc<Refcell>

- Rcは同一のヒープ領域を
  複数のスマートポインタが所有することを認める。
  - 所有権規則を無視

- Refcellは可変性を無視して、半強制的に中の値を
  書き換えられる。
  - 借用規則を無視

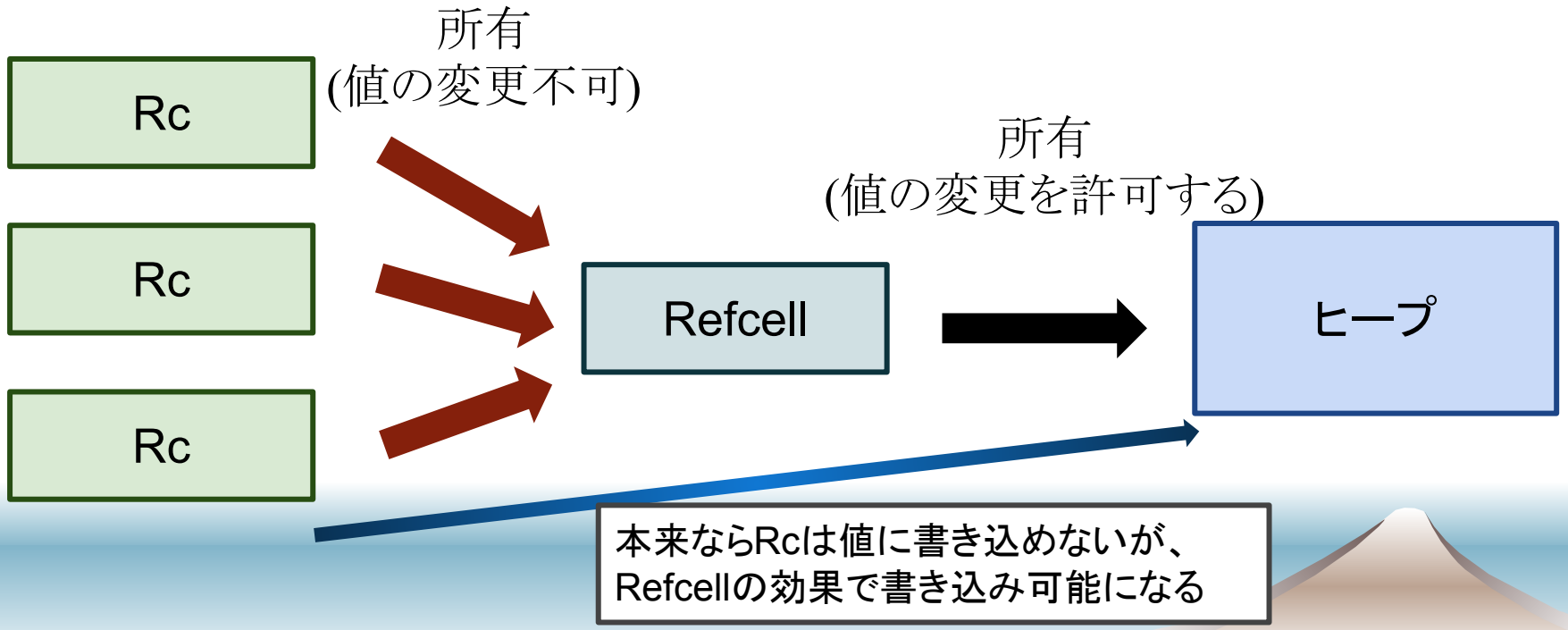


Rc<Refcell>

- RcとRefcellを合わせることで、
  同一の領域に対する変更権を、複数の変数が
  同時に持てるようになる！
  - 代償として、コンパイル時に借用規則違反を検出できない
  - が、一部の実装では定性的に止むを得ない選択

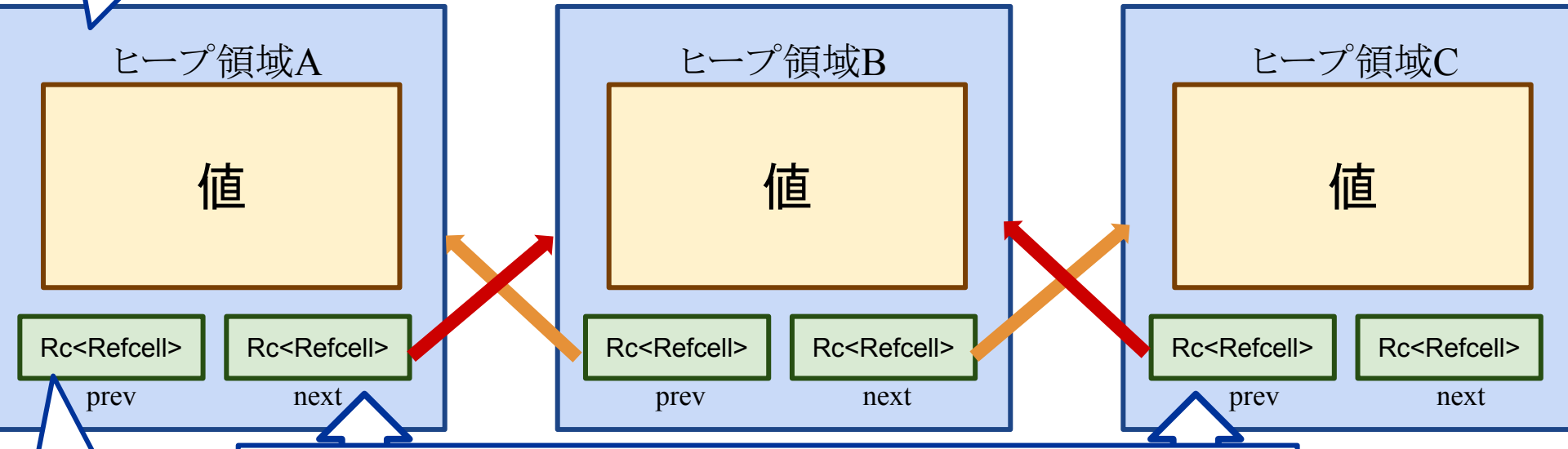
厳密な人向けの注記：正確には、参照先はNilの可能性があるのでOption<Rc<Refcell>>となる。

Rc<Refcell>


所有
(値の変更不可)
所有
(値の変更を許可する)
Rc
Rc
Rc
Refcell
ヒープ
本来ならRcは値に書き込めないが、
Refcellの効果で書き込み可能になる

双方向連結リスト

各ヒープ領域は
Refcellが所有する

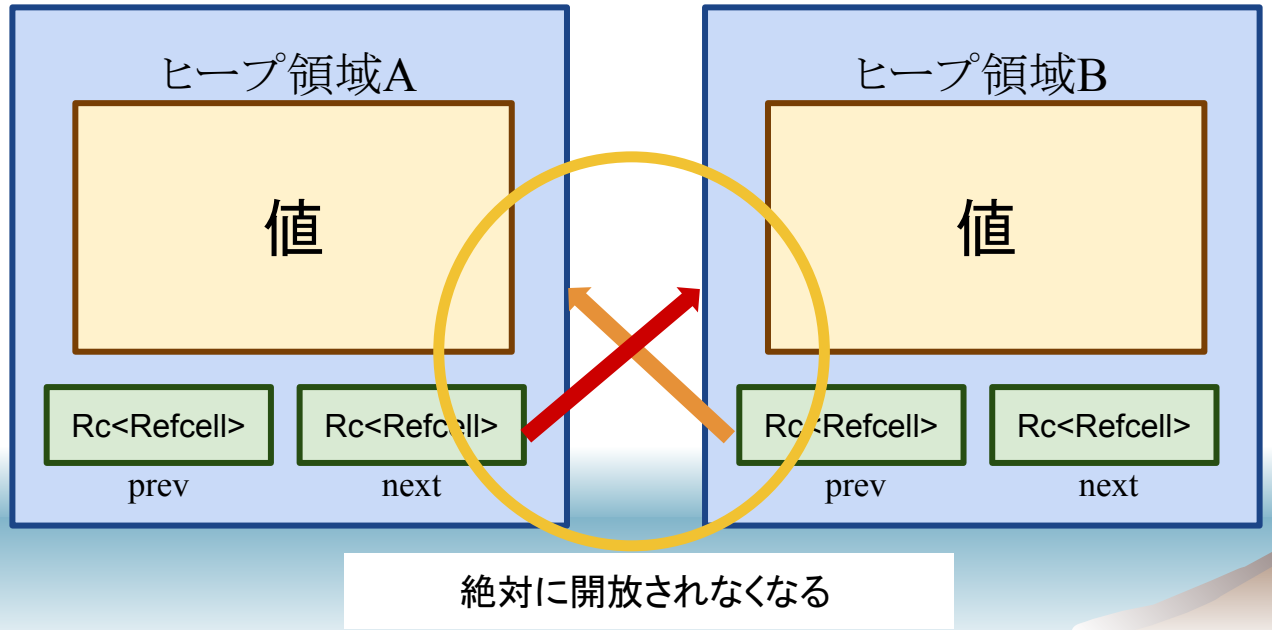


Refcellに
対するRcで
隣接要素に
アクセス。

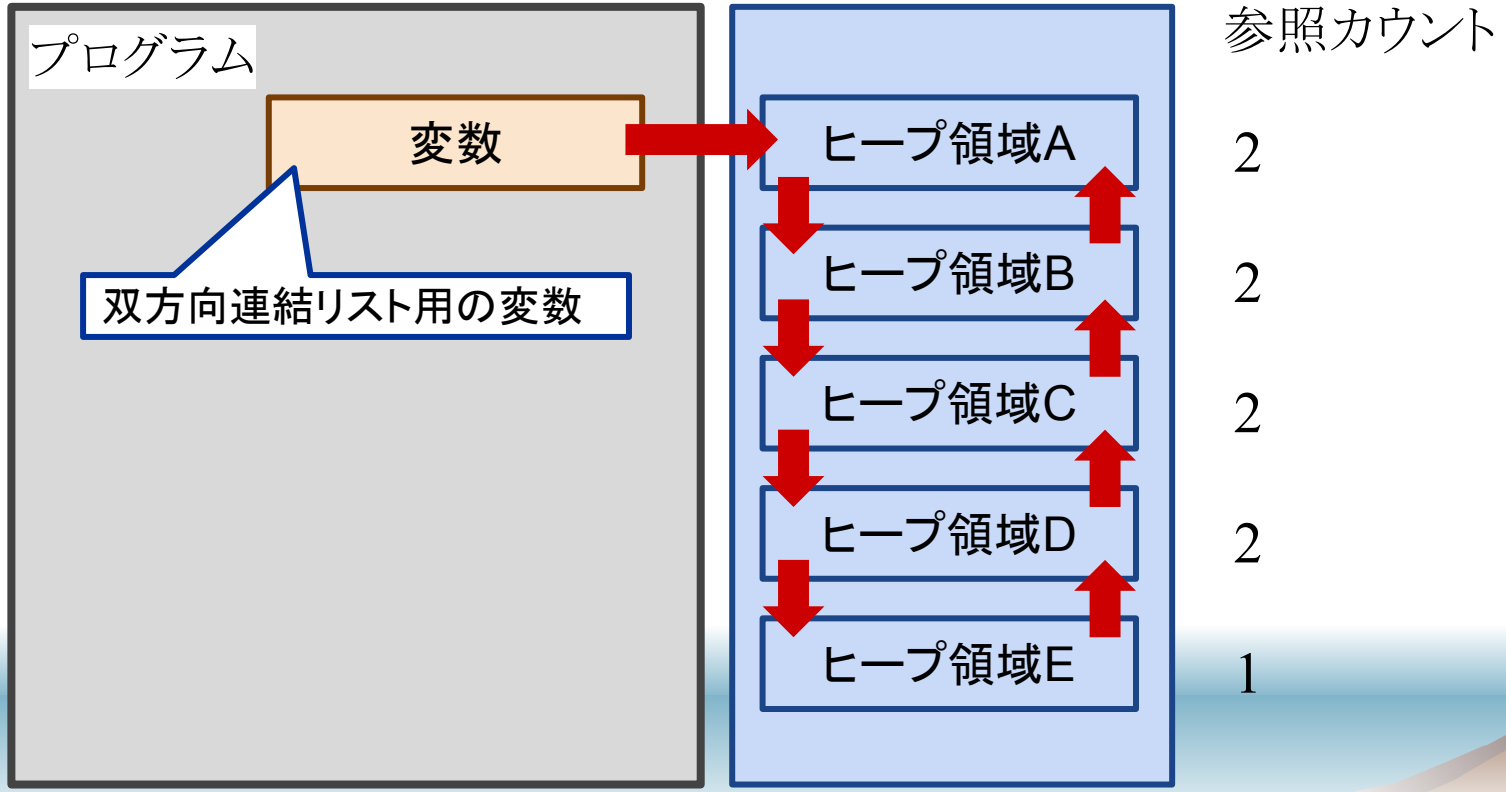
複数のスマートポインタが同じ対象を所有しているが、
Rcの特性により問題なく動作する。
また、Refcellの特性により、隣接ノードの操作が可能となる。

循環参照の発生

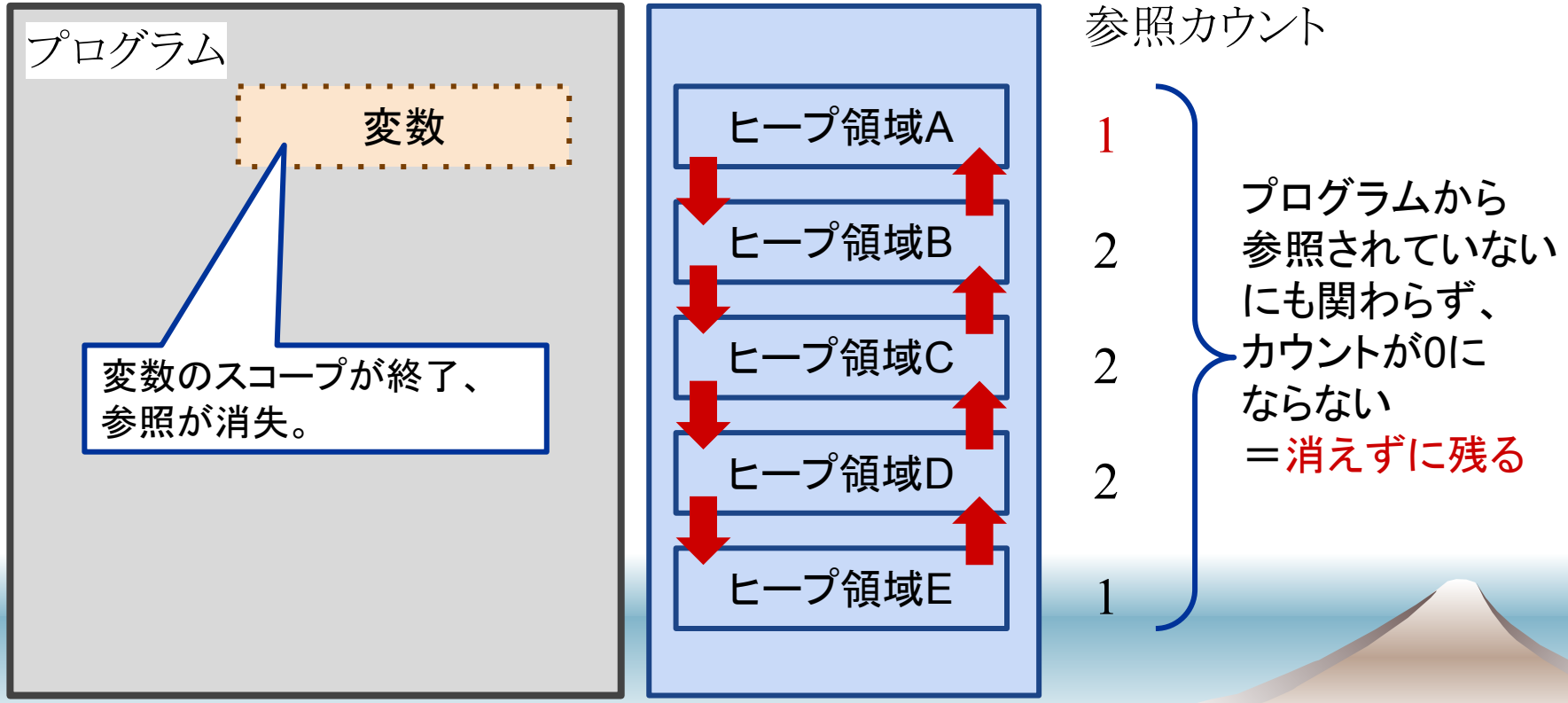
- 実は、ここで循環参照が発生している。



問題となる挙動


プログラム
変数
双方向連結リスト用の変数
ヒープ領域A
ヒープ領域B
ヒープ領域C
ヒープ領域D
ヒープ領域E
参照カウント
2
2
2
2
1

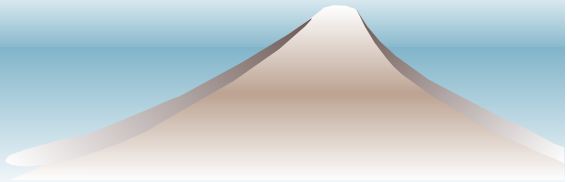
メモリリークが起きました



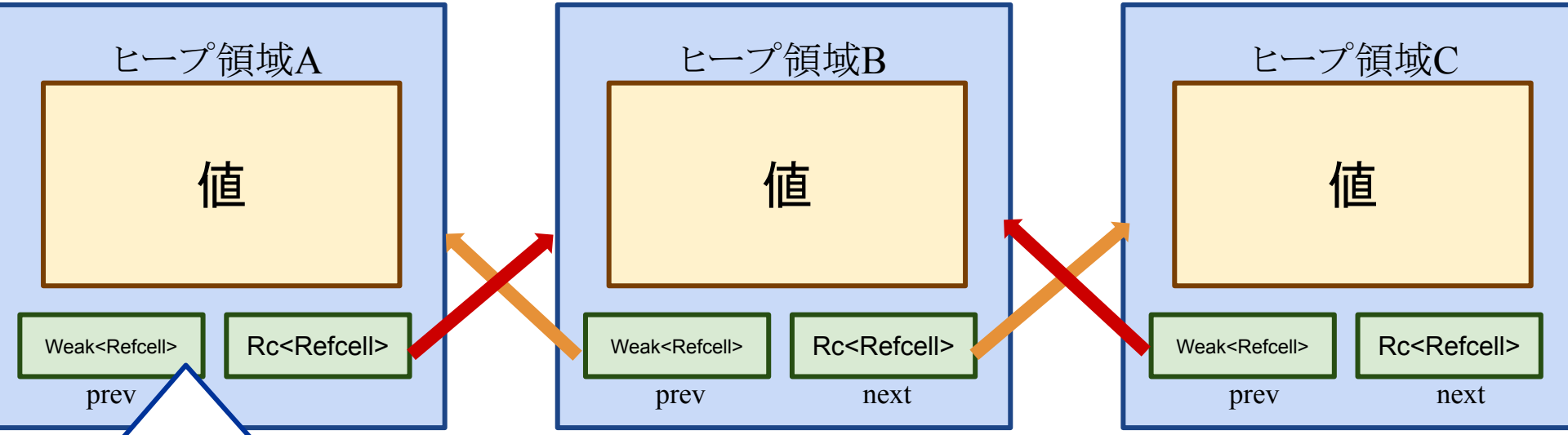
スマートポインタ(4):Weak

- Rcの弱参照版
  - Rcと同様に、値が複数の所有者を持つことを認める
  - C++のstd::weak_ptrに近い

- 参照カウントにおいてWeakの参照数は
  Rcの参照数と区別され、領域の開放に関わらない。

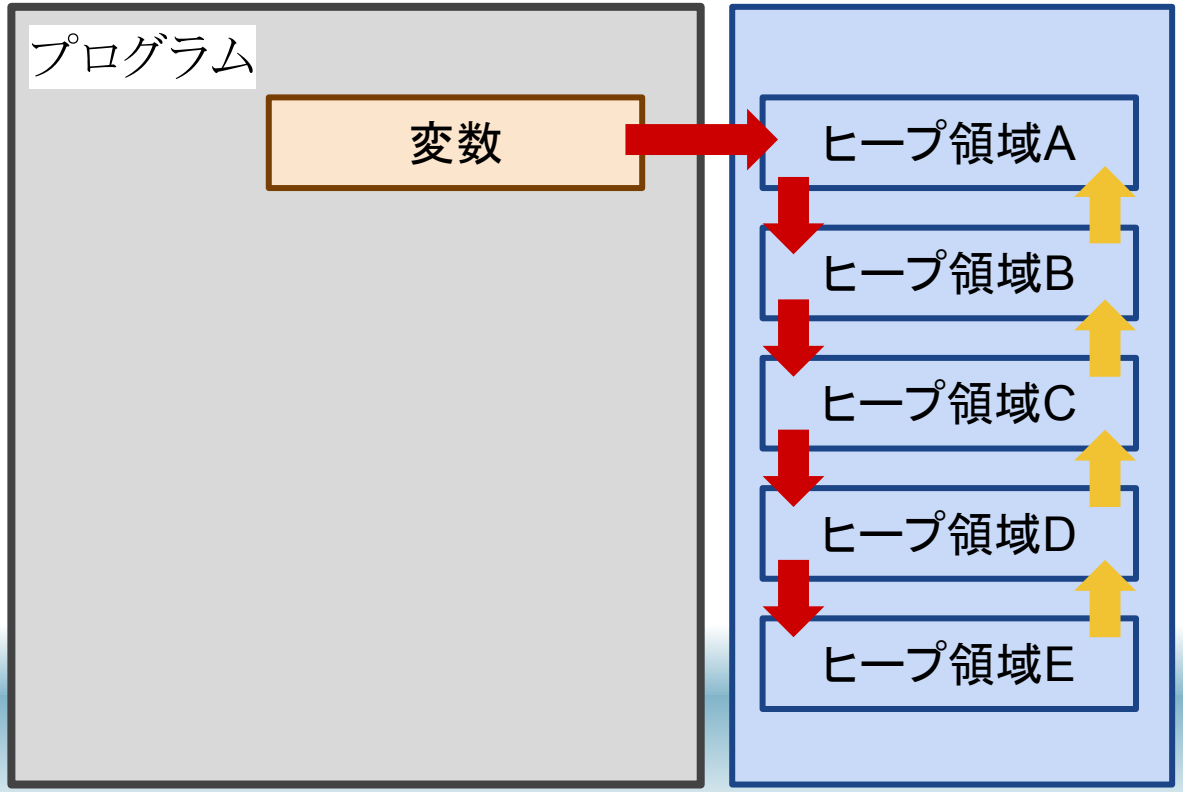


双方向連結リスト



片方の参照をWeakにすることで
循環参照によるメモリリークを回避

修正版の挙動

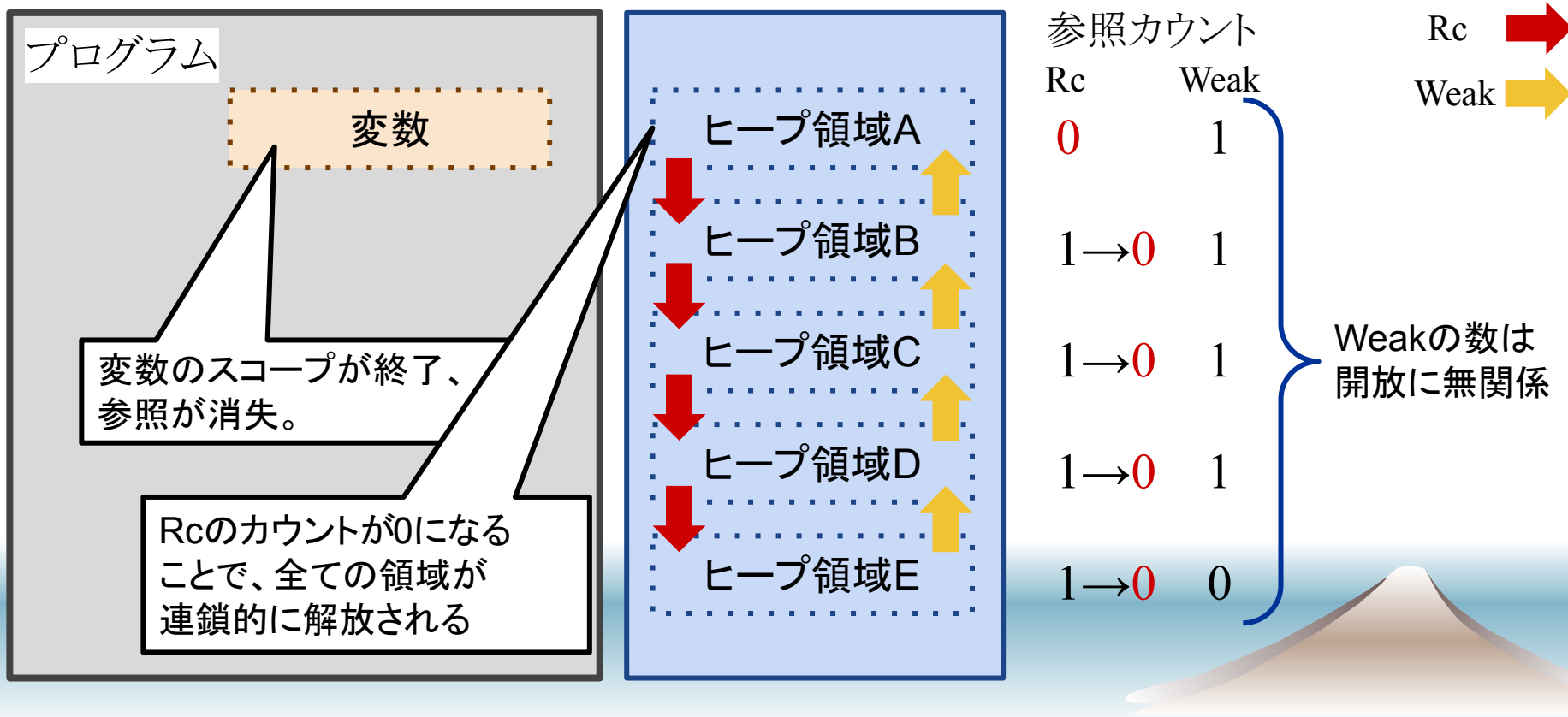


参照カウント	
Rc	Weak
1	1
1	1
1	1
1	1
1	0

Rc

Weak

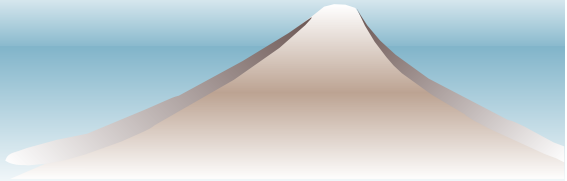
修正版の挙動


プログラム
変数
変数のスコープが終了、
参照が消失。
Rcのカウントが0になる
ことで、全ての領域が
連鎖的に解放される
ヒープ領域A
ヒープ領域B
ヒープ領域C
ヒープ領域D
ヒープ領域E
参照カウント
Rc
Weak
0 1
1→0 1
1→0 1
1→0 1
1→0 0
Rc
Weak
Weakの数は
開放に無関係

スマートポインタ: まとめ

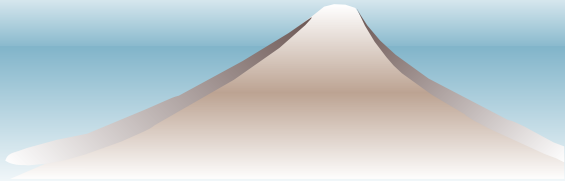
- Rustではスマートポインタを使うことで、
  所有権規則・借用規則を無視した実装が可能となる

- ただし、Rustのスマートポインタは
  メモリの削除に関してC++のような柔軟性がない
  - その分不正は起きにくい



Unsafe Rust

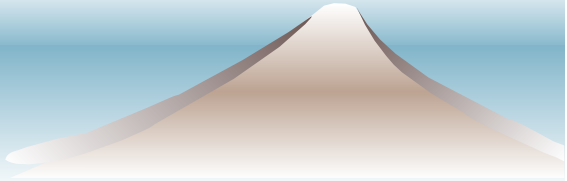
- 最終手段として、メモリの確保・開放を直接行えるUnsafe Rustという手段が存在する。
  - 実質C++



スマートポインタ余談

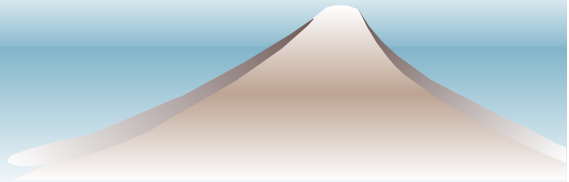
- 実はスマートポインタはC++の標準ライブラリにも存在する(std::unique_ptr など)

- C++では通常のポインタで代用が効くのに対し、Rustの場合一部の実装でほぼ必須



スマートポインタ余談

- C++のスマートポインタは、new等を用いて
  ヒープ領域の確保を直接行う必要がある。
  - Rustは不要

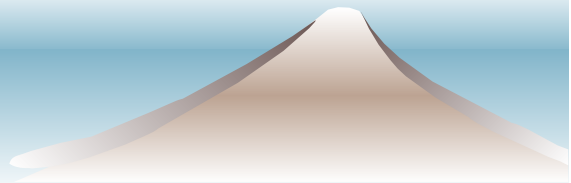


参考文献 1

- Rust programming language（公式doc）
  https://www.rust-lang.org/
- Stack Overflow Developer Survey 2020
  https://insights.stackoverflow.com/survey/2020

参考文献 2

- Deno
  https://deno.land/
- Verifying Invariants of Lock-Free Data Structures with Rely-Guarantee and Refinement Types
  https://dl.acm.org/doi/10.1145/3064850



Appendix

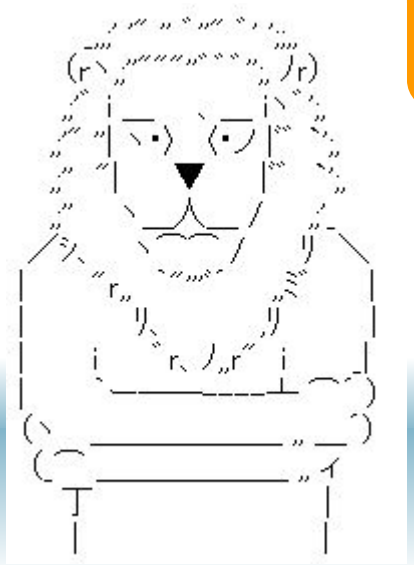
Rustの位置付け

- パラダイム
  - マルチパラダイムプログラミング言語
-

Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)



お前それC++の前でも同じこと言えんの？

Rustのコンセプト

効率的で信頼できるソフトウェアを誰もがつくれる言語

(A language empowering everyone to build reliable and efficient software.)

お前それC++の前でも同じこと言えんの？

ネイティブへのコンパイルと
ゼロコスト抽象化（後述）によって
大体CやC++と同じくらい速い！
将来的にはもっと早くなるかも…？

型付け警察24時

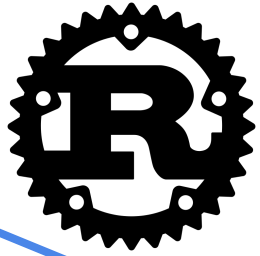
JavaScriptには型がないから💩！
TypeScriptには型があるから最高！

ﾋﾟﾋﾟｰｯ！！
JavaScriptにも型はあります！
ただ，動的型付けなJavaScriptと比べると，TypeScriptは静的型付けなので，実行時エラーを排除したり，型注釈やIDEによる開発支援を得たりと，より型のメリットを享受...

Rustで書かれたソフト

Rustの大規模プロジェクト

- Servo
  - Rustで開発されているHTMLレンダリングエンジン（FireFoxは一部Servoを使っている）
- Rustコンパイラ



これからが楽しみ！

Node.js

- サーバサイドのためのJavaScript実行環境
  - ○

Nodeを並び替えるとDeno，ロゴがkawaii～！

Deno（ディーノ）

- 新しいJavaScriptのランタイム
  - Node.jsの開発者が，
    Node.jsでの反省を生かして開発した
    （強くてニューゲーム）

Nodeを並び替えるとDeno，ロゴがkawaii～！

C++との対応(1)

C++		Rust
void myfunc (int foo){}	≒	fn myfunc (foo: String){}
void myfunc (int foo){}	＝	fn myfunc (mut foo: String){}
void myfunc (const int& foo){}	≒	fn myfunc (foo: &String){}
void myfunc (int& foo){}	＝	fn myfunc (foo: &mut String){}

C++との対応(2)

C++		Rust
void myfunc (std::string foo){}	≠	fn myfunc (foo: String){}
void myfunc (std::string foo){}	≠	fn myfunc (mut foo: String){}
void myfunc (const std::string& foo){}	≒	fn myfunc (foo: &String){}
void myfunc (std::string & foo){}	=	fn myfunc (foo: &mut String){}

型の役割 (1) 抽象化，隠蔽

- 複数の値をまとめてひとつの値と見なせる (一つの変数で「持ち歩ける」) ようにする

```
1 typedef struct  {
2   float x; float y;
3 } vec2;
4 vec2 add(vec2 a, vec2 b);
```

- 型の「中身」をわかっていなくても使え

```
1 FILE * fp = fopen("foo.txt", "rb");
2 fread(buf, n, 1, fp);
```

「型」はいくつかのデータを一つにまとめることで
「抽象化」している（データ抽象化）

オブジェクト指向的な考え方

「オブジェクト指向言語，Python / Python : An Object-Oriented Language」より

1. モジュール化・抽象化:
   - 「データ型＋外から呼ばれる手続き (インタフェース)」をワンセットの部品としてソフトを構築していく ⇒ クラス
   - 「外から呼ばれる手続き」の意味にだけ依存したコードは，部品の中身 (実装) が変わっても動きつづける
2. 多相性の利用:
   - クラスが異なれば同じ名前で違うメソッ ⇒ 「同じ使い方の部品」をたくさん作れる

オブジェクト指向では，クラスなどを利用することで，
具体的な実装を捨象（抽象化）する！

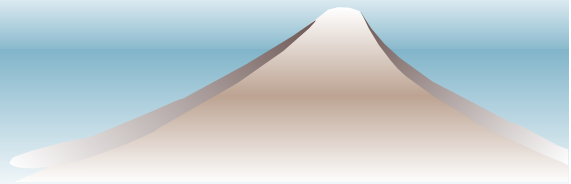
関数型言語では，Lambda計算や高階関数など

Rustacean in 10 min !!

Rustの非公式マスコット(Ferris)
crustacean（甲殻類の）が由来.

Rustのインストール

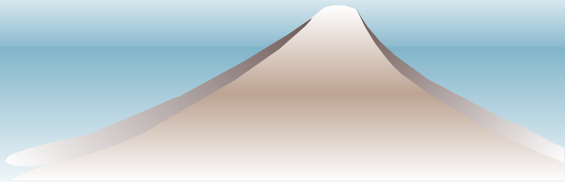
```
$ curl https://sh.rustup.rs -sSf | sh
```



Cargo

Rustのビルドツール兼パッケージマネージャ

- ビルドツール
  - makeとか，Pythonの
- パッケージマネージャ
  - Pythonのpip，Rubyのbundle

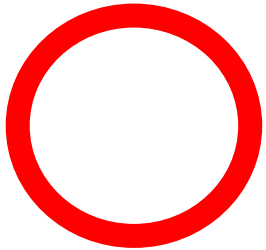


シャドーイング (Shadowing)

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

    let x = x + 1;

    let x = x * 2;

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

シャドーイング (Shadowing)

```
let mut spaces = "    ";
spaces = spaces.len();
```



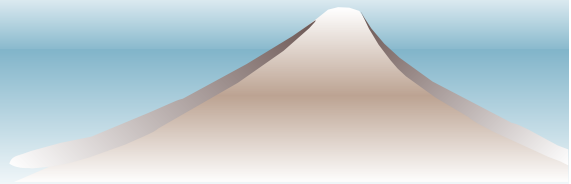
```
let spaces = "    ";
let spaces = spaces.len();
```

基準型

- String
- f64

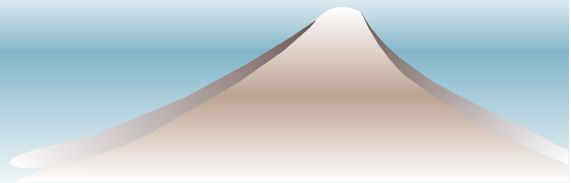
index out of bounds → panic

```
fn main() {
    let a = [1, 2, 3, 4, 5];
    let index = 10;
    let element = a[index];
    println!("The value of element is: {}", element);
}
```



index out of bounds → panic

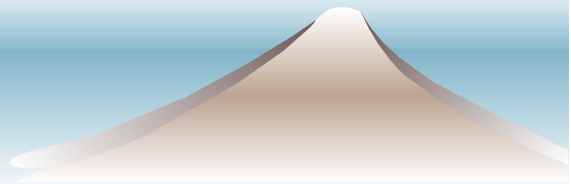
```
$ cargo run
   Compiling arrays v0.1.0 (file:///projects/arrays)
    Finished dev [unoptimized + debuginfo] target(s) in 0.31 secs
     Running `target/debug/arrays`
thread '<main>' panicked at 'index out of bounds: the len is 5 but the index is
 10', src/main.rs:6
note: Run with `RUST_BACKTRACE=1` for a backtrace.
```



多相性??

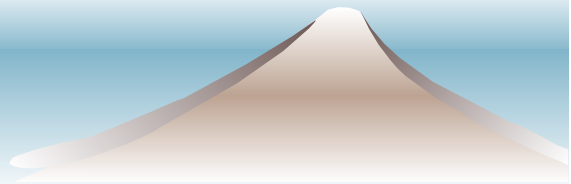
```
fn main() {
    let x: (String, f64, u8) = (1, 2.0, 3);
    let tuple_first = x.0;

    let y = [1, 2, 3, 4, 5];
    let array_first = y[0];
}
```

なぜn番目の要素への
アクセスの仕方が異なる？



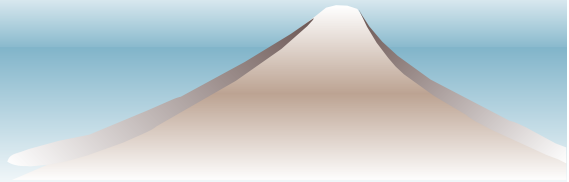
ヒープとスタック

- スタック
  - int hoge = 123;
- ヒープ
  - int *hoge = (int*)malloc(sizeof(int) * 1)
    *hoge = 123



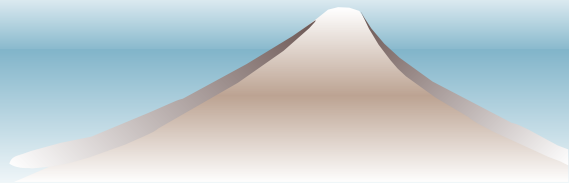
Tips: 実引数と仮引数

- 実引数 (argument)
  - fn another_function(**x**: String) { …
- 仮引数 (parameter)
  - another_function(**5**)



Tips: 文と式

- 文 (sentence)
  - fn another_function(**x**: String) { …
- 式 (expression)
  - another_function(**5**)



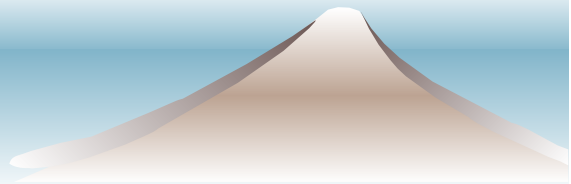
Tips: 式指向言語

- 文 (sentence)
  - fn another_function(**x**: String) { …

- 式 (expression)
  - another_function(**5**)

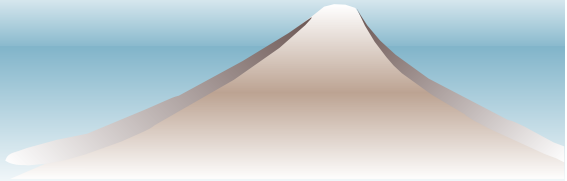
式指向言語

```
fn main() {
    let x = plus_one(5);
    println!("The value of x is: {}", x);
}
fn plus_one(x: String) -> String {
    x + 1;
}
```



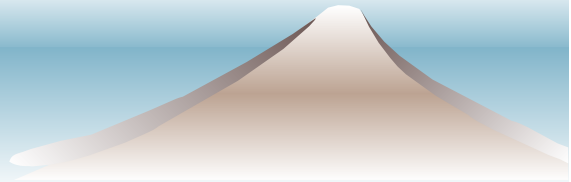
変数のスコープ

```
{                                   // sのスコープ外
    let s = "hello";                // sのスコープ内
}                                   // sのスコープ外
```



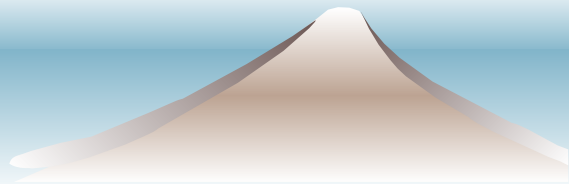
StringのMutableとImmutable

```
let s_1 = "hello";

let mut s_2 = String::from("hello");
```

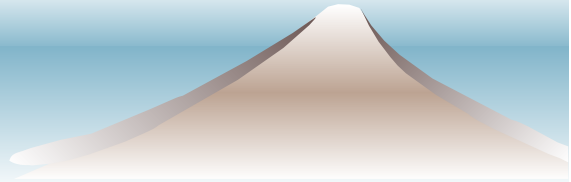


Memory と Allocation

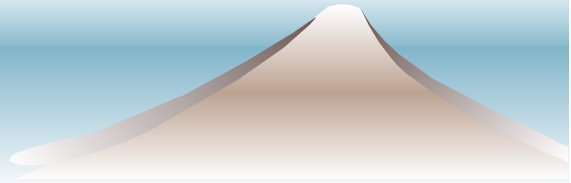
- Immutable
  - コンパイル実行時にメモリ確保のサイズが明らか
- Mutable
  - プログラム実行中に必要なメモリ確保のサイズが変わる
  - コンパイルの時点では



リソースの管理



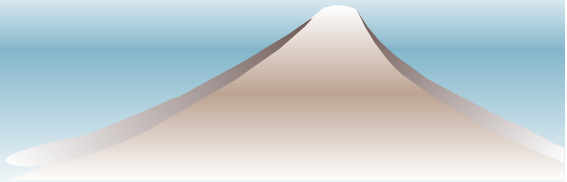
値と変数



Rustの変数と参照

Rustの変数と参照は、可変と不変の2種類がある
変数の指す値を変更できるかどうかに影響する

- 変数
  - 不変変数
  - 可変変数
- 参照
  - 不変参照
  - 可変参照



スマートポインタ

- スマートポインタはヒープ領域の値に、
  直接もしくは間接的に紐づけられる

- オブジェクト同士が相互参照するような場合には、
  スマートポインタを用いて実装する