

# C言語講習会

講師 江端

本講習会で使用したソースコードは, 以下にアップロードする予定です

[https://github.com/Tohoku-University-Takizawa-Lab/study\\_session\\_2020](https://github.com/Tohoku-University-Takizawa-Lab/study_session_2020)

# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

# 本日の流れ

時間的に全部はできないかも...

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

黄色い部分は確実にやりたい

# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

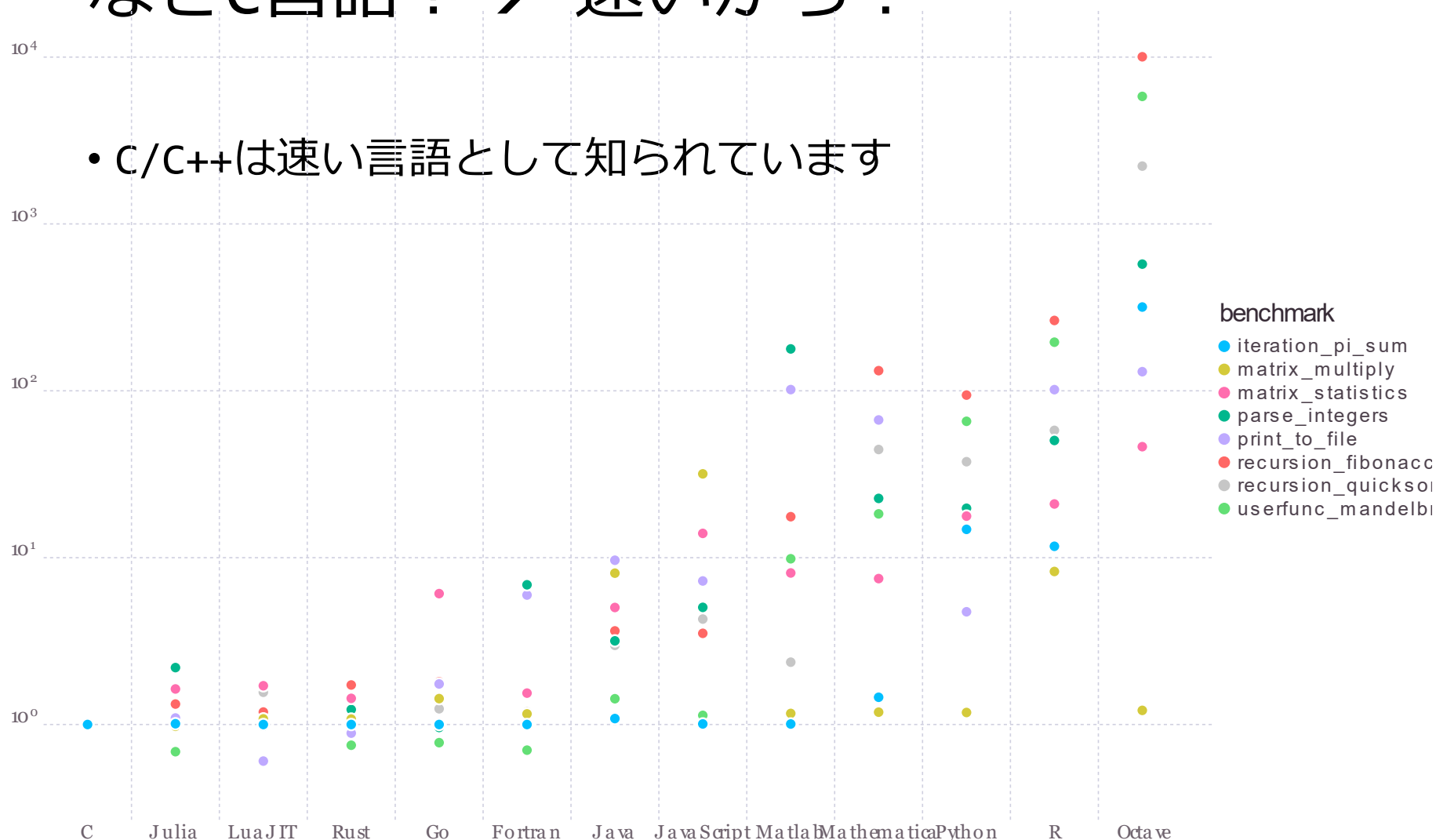
# 世界には便利な言語がたくさんあります

- 世の中にはプログラミング言語がたくさんあります
  - Python, Java, JavaScript, Go, Haskell, PHP, C/C++, ...
  - ちなみに僕のおすすめ → Python
- C言語は決して“便利な”言語ではありません
  - オブジェクト指向でない
  - Pythonなどに比べて, ライブラリで出来ることが少ない

あえて, Cを勉強する理由とは？

# なぜC言語？ → 速いから！

- C/C++は速い言語として知られています



# 速いので、HPC分野でよく使われる

- C言語で書かれたスパコン用のコードがたくさんある
  - 「コード資産が多い」などと言われます
- というか、SX-Aurora TSUBASAはFortranとC/C++しか動きません...
- +α：速度重視のPythonライブラリもFortranやC/C++で実装されていることが多い
  - Numpy：数値計算用ライブラリ（C/Fortran）
  - Tensorflow：機械学習ライブラリ（C++）

※「+α」は発展的な話題や、ちょっと脱線した話題を表すマークです。

# HPCの研究室にいるなら...

- スパコン向けに書かれたコードを読めるようになろう！
  - 言語：C/C++, Fortran
  - ライブラリ：OpenMP, MPI, CUDA, OpenCL, OpenACC
- 速いコードを書けるようになろう！
  - アーキテクチャに対する理解も必要

今日はその第一歩！



# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

# 事前アンケートの結果

- どれくらいC言語分かりますか？
  - 関数, 条件分岐, ループがわかる : **3人**
  - ポインタ, 構造体もわかる : **3人**

ウォーミングアップを兼ねて  
ちょっと復習！

# FizzBuzzを書こう！

- 次のようなプログラムを作ってください

- 1, 2, 3, ..., i, ..., N-1, Nについて
  - iが3で割り切れるなら Fizz
  - iが5で割り切れるなら Buzz
  - iが3と5の両方で割り切れるなら FizzBuzz
  - それ以外なら i

を出力してください

- 例：N=16 ( $1 \leq i \leq 16$ )

- 1, 2, Fizz, 4, Buzz, Fizz, 7, 8, Fizz, Buzz, 11, Fizz, 13, 14, FizzBuzz, 16

# 解答略

ウォーミングアップ完了！

# ポインタと構造体の説明に移る前に

- ここから、ポインタと構造体の説明をします
- しかし、受講者6人中3人はすでにポインタと構造体を完全理解しているようです...
- 知っていることを聞いても退屈だと思うので、演習課題を3個作りました
  - 課題①：P.59
  - 課題②：P.72
  - 課題③：P.77
  - 3個全部終わった人 → P.129
  - 参考までに、解き終わった人は「課題①終わりました」みたいにチャットで教えてほしいです

# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

# データとメモリアドレスの関係1

- ポインタの勉強を始めるまえに、データとメモリアドレスの関係を復習をします

- `int a;` ← これは何？

- 4byte整数型のデータ

- どこにあるの？

- メモリ上の連続する4byteに保存されています

- `+α` : 物理的にはレジスタにあるかもしれないし、キャッシュにあるかもしれないし、DRAMにあるかもしれないし、ディスクにあるかもしれない

memory

1byte
1byte
1byte
1byte
1byte
1byte
1byte
1byte
1byte
1byte
1byte
1byte

int a

# データとメモリアドレスの関係2

- メモリアドレスって何？ → 1byteごとに割り振られたメモリ上の住所のようなもの

- 本来アドレスは16進数ですが，便宜上本講習では10進数で表すことにします

- 右図では0004~0007の4byteがint aに利用されています

- +a : int型のような4byte型のアドレスは必ず4の倍数から始まる.  
double型のような8byte型では8の倍数.  
これを”アラインメント”と言います.

address	
0000	int a
0001	
0002	
0003	
0004	
0005	
0006	
0007	
0008	
0009	
0010	
0011	



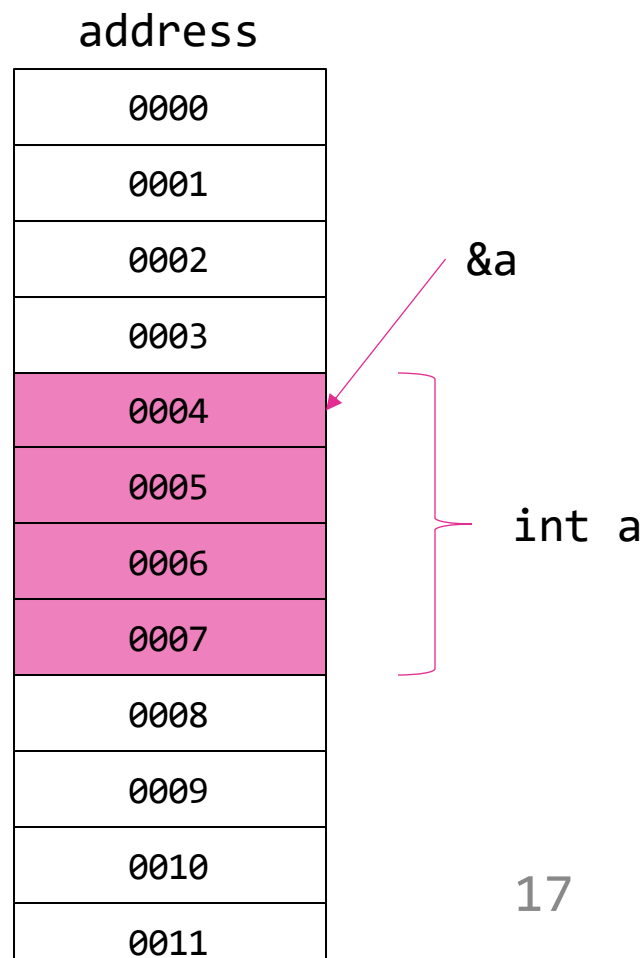
# データとメモリアドレスの関係3

- どうやればアドレスを見ることができるの？ → `&(変数名)`

- `printf("%p", &a);` → 0004

- `&(変数名)`は、連続する4byteの先頭のアドレスを示しています

ここまでは大丈夫でしょうか？  
それでは、ポインタです



# ポインタはアドレスを記憶する

- 結局ポインタって何なの？
  - ポインタはアドレスを保存するためのデータ型



# どうやって使うの？

- 宣言の仕方

- \* (ポインタ宣言子) をつけて宣言します.
- 例: int型のポインタ `int *pi;`
- 例: double型のポインタ `double *pd;`

- 値の代入

- ポインタに代入できるのはメモリアドレスです.
- ですので, 下のようにアドレスを代入できます.
- 注: int型のポインタにはint型以外のアドレスを代入しては, いけません. 他の型のポインタでも同様です.

```
int a;  
int *pi;  
pi = &a;
```

```
double b;  
double *pd;  
pd = &b;
```

# 間接参照

- ポインタの特筆すべき機能に「間接参照」があります.
- ポインタに \* (間接参照演算子)をつけることで, ポインタが保存しているアドレスにあるデータにアクセスすることができます.

```
int a, b;  
int *pa;
```

```
a = 5;  
pa = &a; // paにaのアドレスを代入  
b = *pa; // bにaの値(5)を代入
```

# ソースコード見た方が速い

```
#include <stdio.h>

int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);
}
```

時系列順に  
追っていこう



# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){
```

```
    int a = 8;
```

```
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

メモリ上に2つのデータが  
生まれた...

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

```
    pa = &a; // paにaのアドレスを代入
```

```
    printf("&a: %p\npa: %p\n", &a, pa);
```

```
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更
```

```
    printf("&a: %p\npa: %p\n", &a, pa);
```

```
    printf("a : %d\n*pa: %d\n", a, *pa);
```

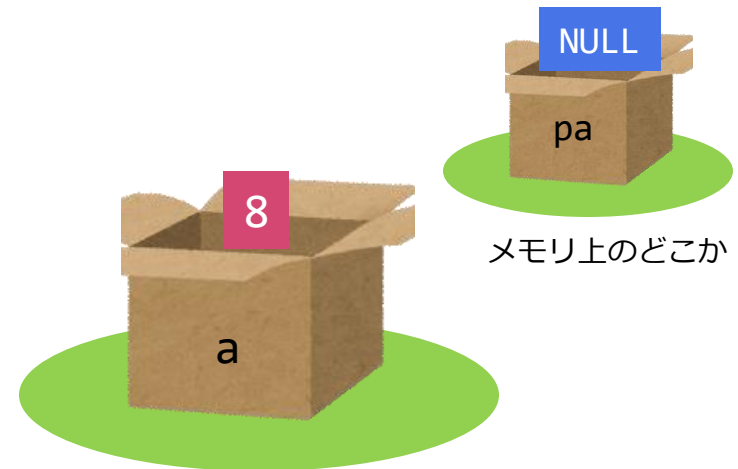
```
    a = 4;
```

```
    printf("&a: %p\npa: %p\n", &a, pa);
```

```
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

何も指していない



メモリ上のどこか

メモリ上のどこか

# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

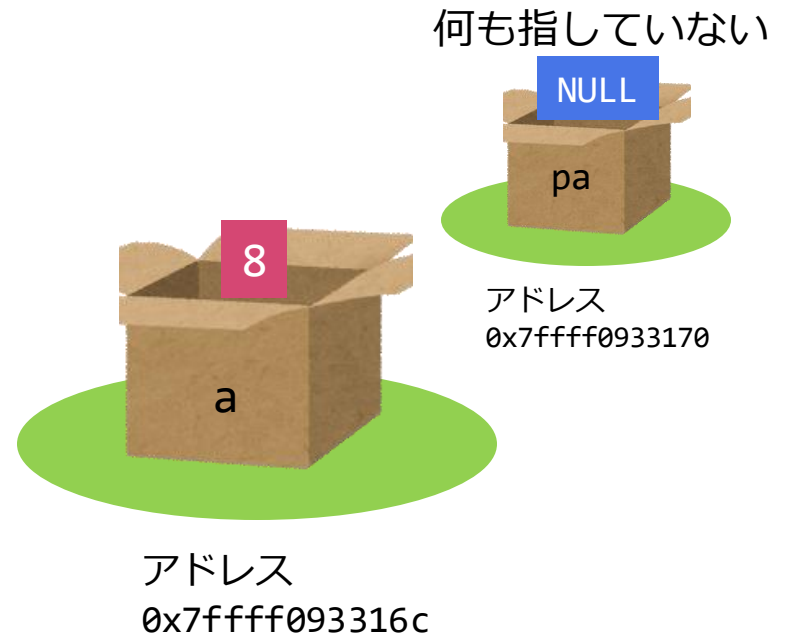
```
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

&a : 0x7ffff093316c  
&pa: 0x7ffff0933170  
pa : (nil)



# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

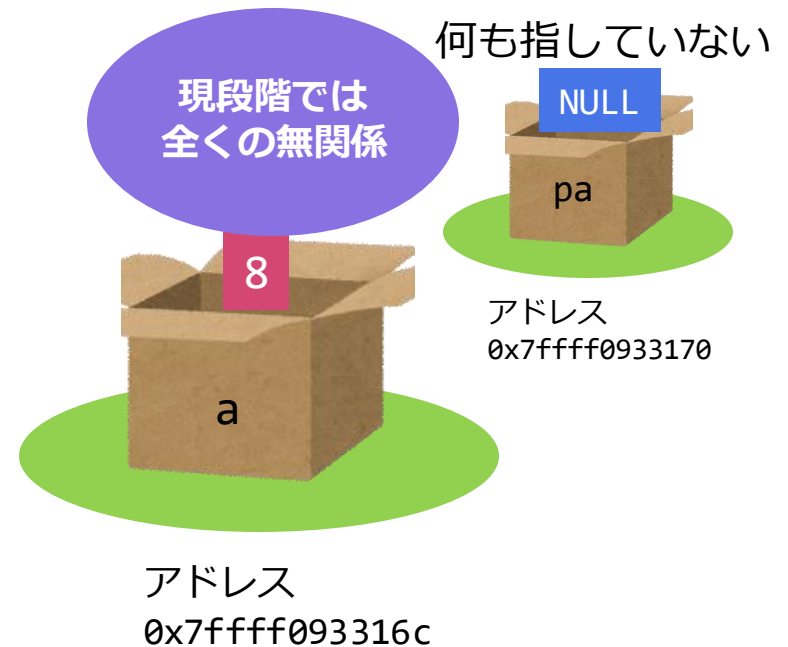
```
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

&a : 0x7ffff093316c  
&pa: 0x7ffff0933170  
pa : (nil)





# ソースコード見た方が速い

```
#include <stdio.h>

int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

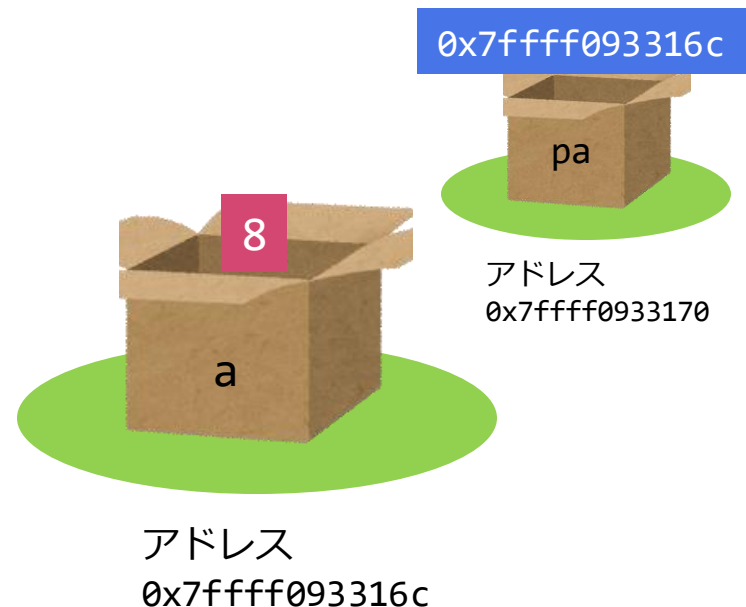
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);
}
```

paの値が&aに  
変更される



# ソースコード見た方が速い

```
#include <stdio.h>

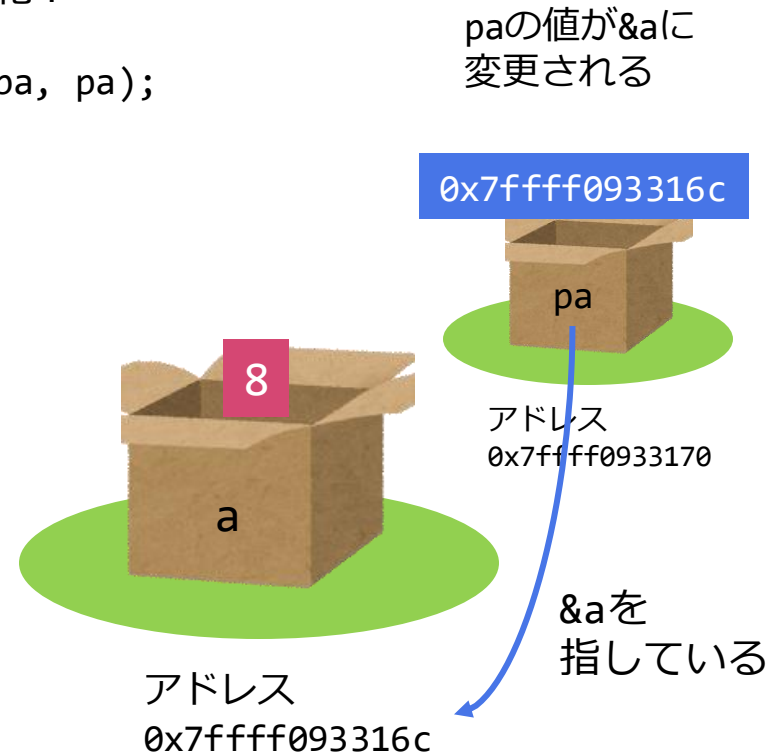
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a   : %d\n*pa: %d\n", a, *pa);
}
```



# ソースコード見た方が速い

```
#include <stdio.h>

int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

```
    pa = &a; // paにaのアドレスを代入
```

```
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

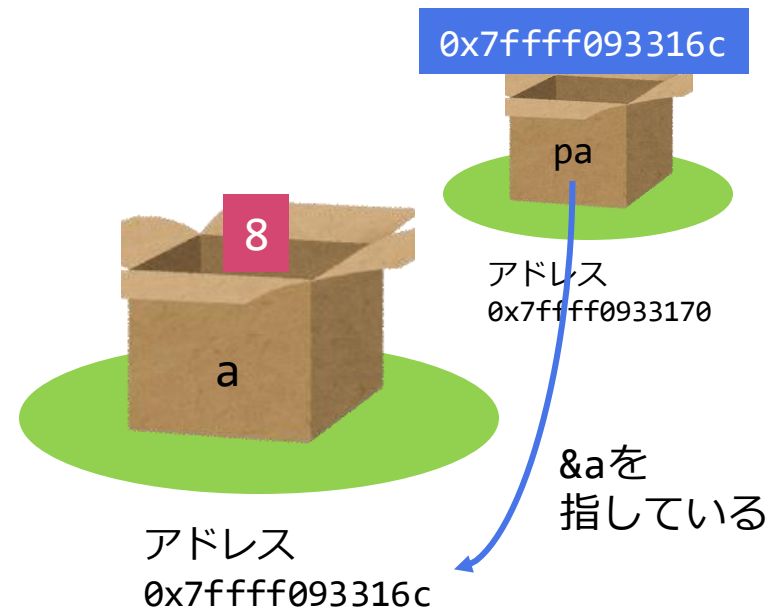
&a: 0x7ffff093316c

pa: 0x7ffff093316c

a : 8

\*pa: 8

同じ！



# ソースコード見た方が速い

```
#include <stdio.h>

int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```

&a: 0x7ffff093316c

pa: 0x7ffff093316c

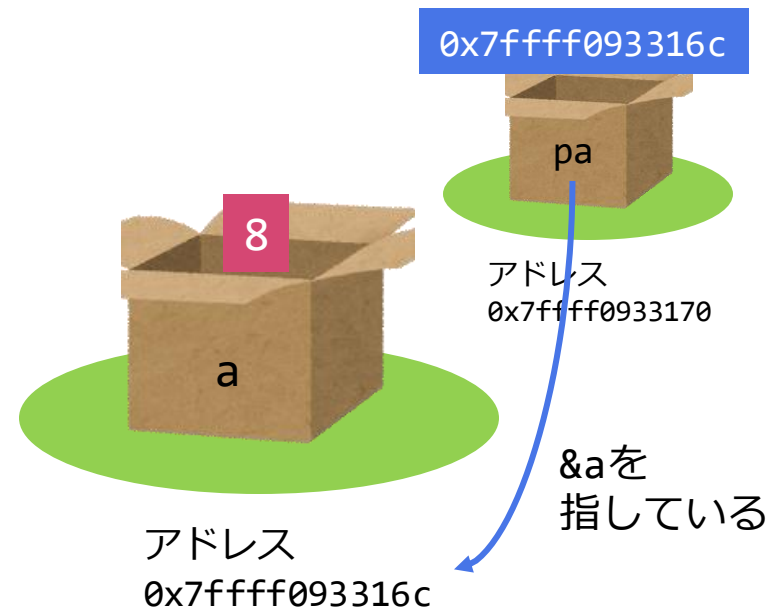
a : 8

\*pa: 8

同じ！

なぜ同じに？

\*paの値を出力する過程を追ってみよう！



# ソースコード見た方が速い

```
#include <stdio.h>

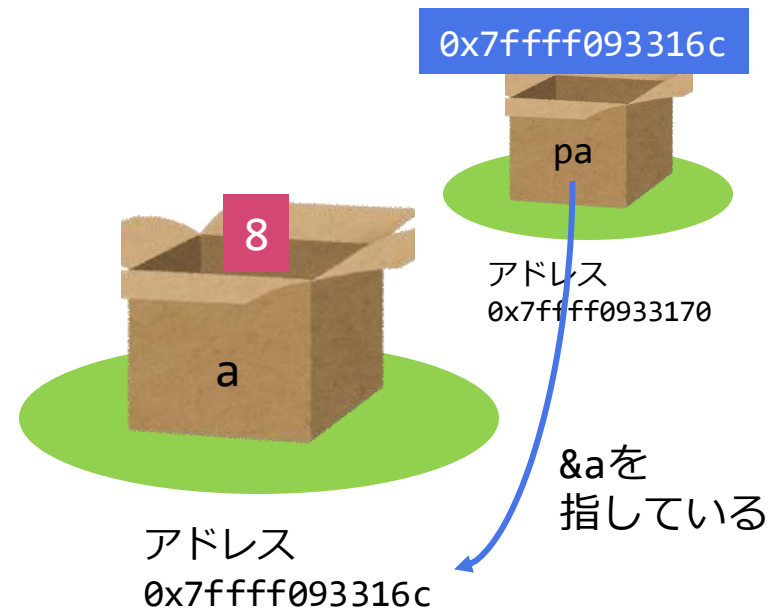
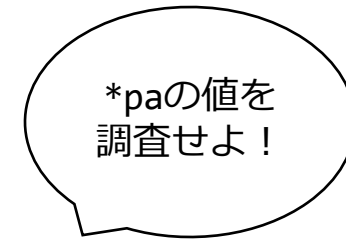
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```



# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

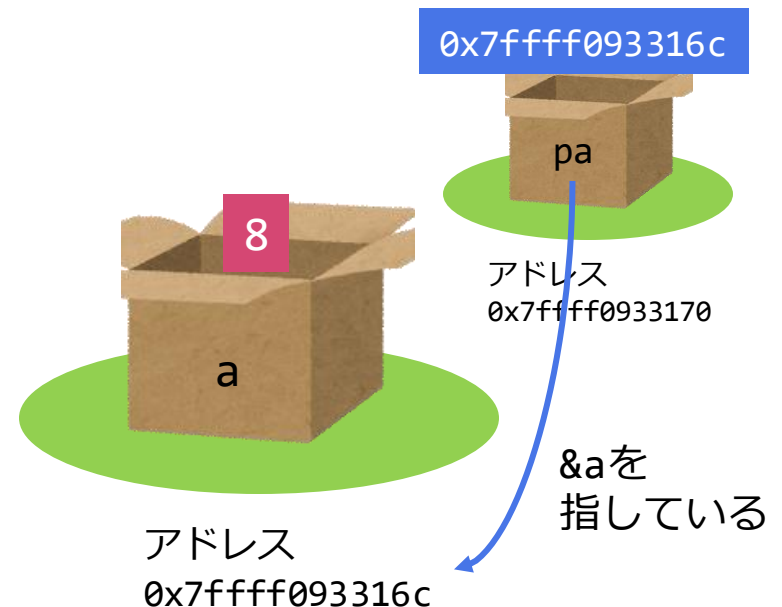
    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```

\*paの値を調査せよ！  
→ paが指している  
先にあるデータの  
値を調査せよ！



了解！

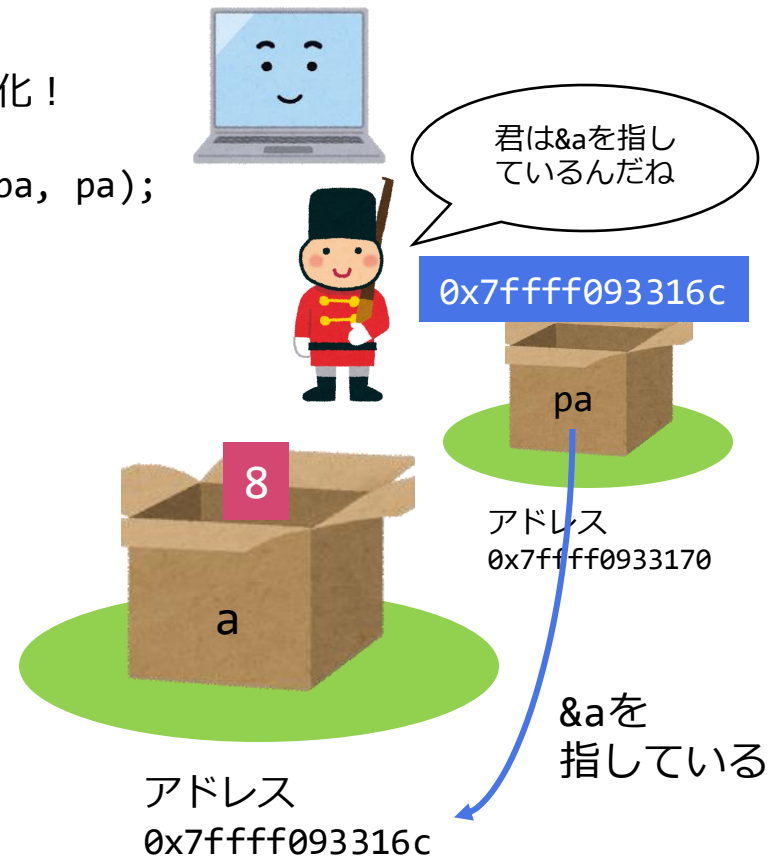


# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！  
  
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);  
  
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);  
  
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);  
  
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);  
}
```

\*paの値を調査せよ！  
→ paが指している  
先にあるデータの  
値を調査せよ！



# ソースコード見た方が速い

\*paの値を調査せよ！  
→ paが指している  
先にあるデータの  
値を調査せよ！

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```



```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

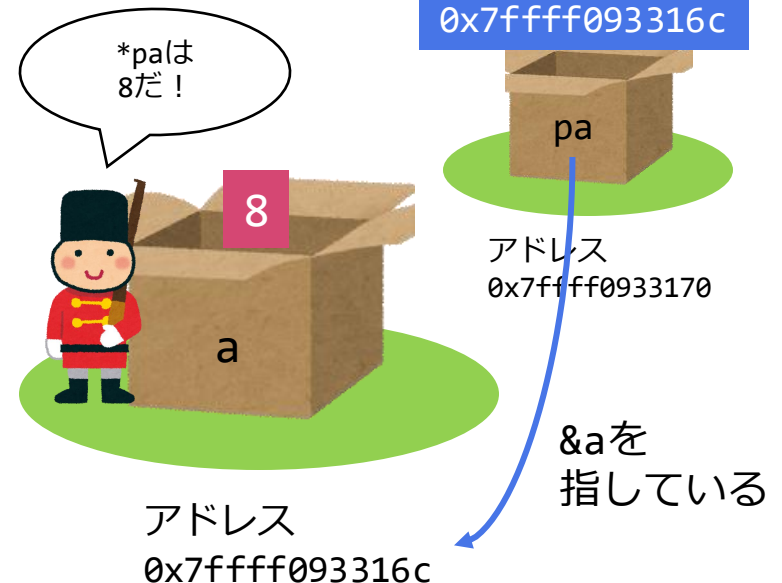
```
    pa = &a; // paにaのアドレスを代入
```

```
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```





# ソースコード見た方が速い

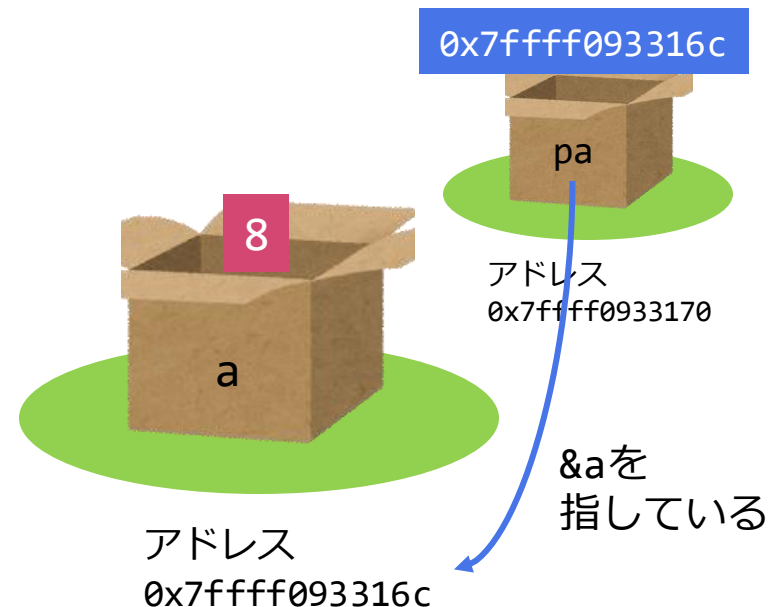
```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！  
  
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);  
  
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);  
  
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);  
  
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);  
}
```

\*paの値を調査せよ！  
→ paが指している  
先にあるデータの  
値を調査せよ！



\*paは  
8です



# ソースコード見た方が速い

```
#include <stdio.h>

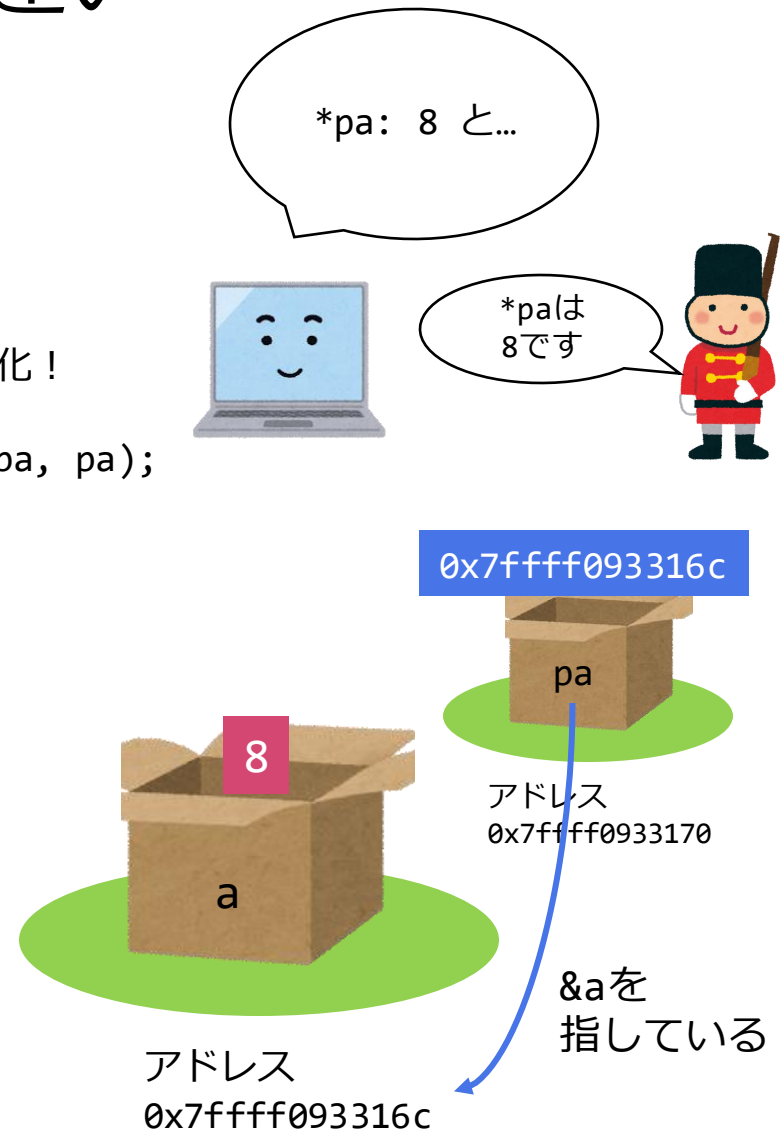
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```



# ソースコード見た方が速い

```
#include <stdio.h>

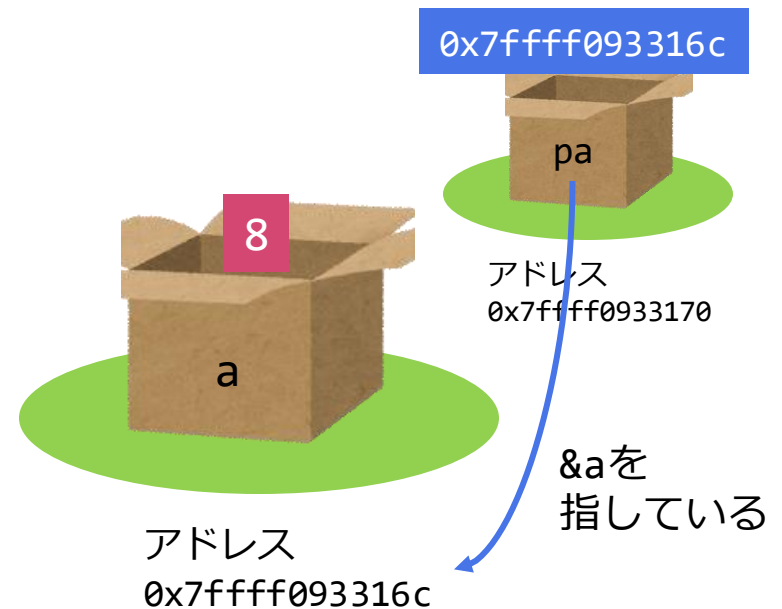
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```



# ソースコード見た方が速い

```
#include <stdio.h>

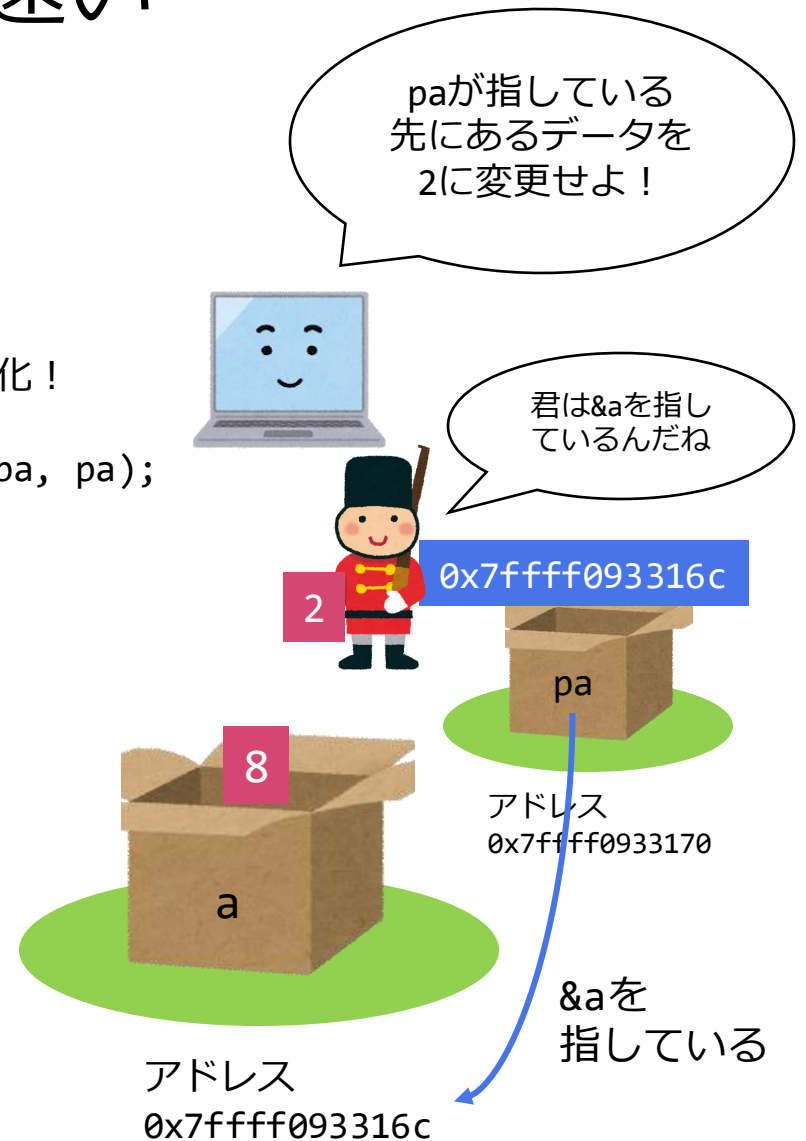
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```



# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

```
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

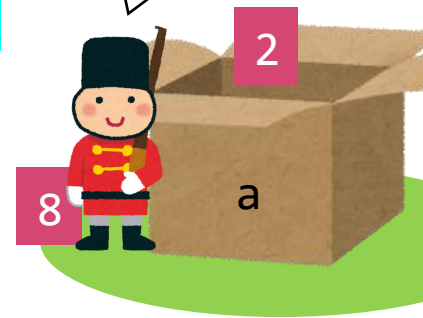
```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

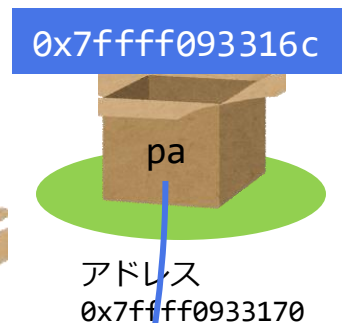
paが指している  
先にあるデータを  
2に変更せよ！



変更完了！



アドレス  
0x7ffff093316c



アドレス  
0x7ffff0933170

&aを  
指している

# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

```
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

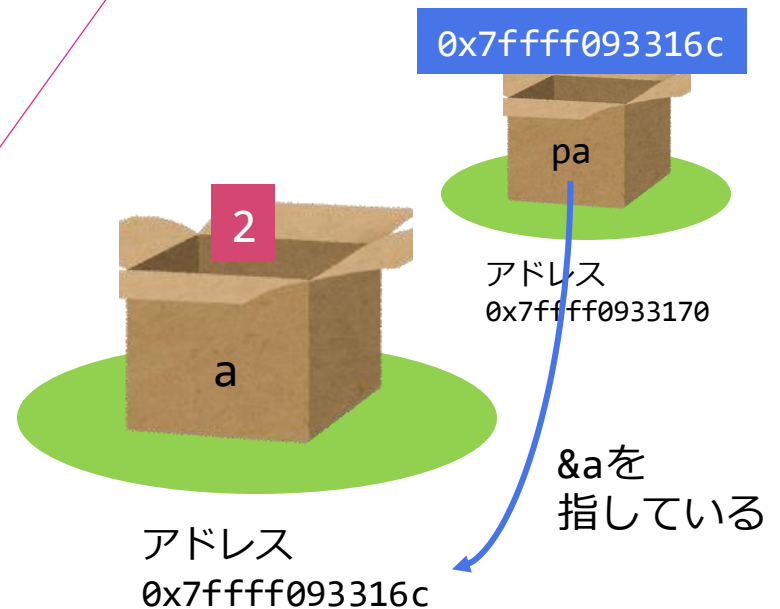
&a: 0x7ffff093316c

pa: 0x7ffff093316c

a : 2

\*pa: 2

同じ！



# ソースコード見た方が速い

```
#include <stdio.h>

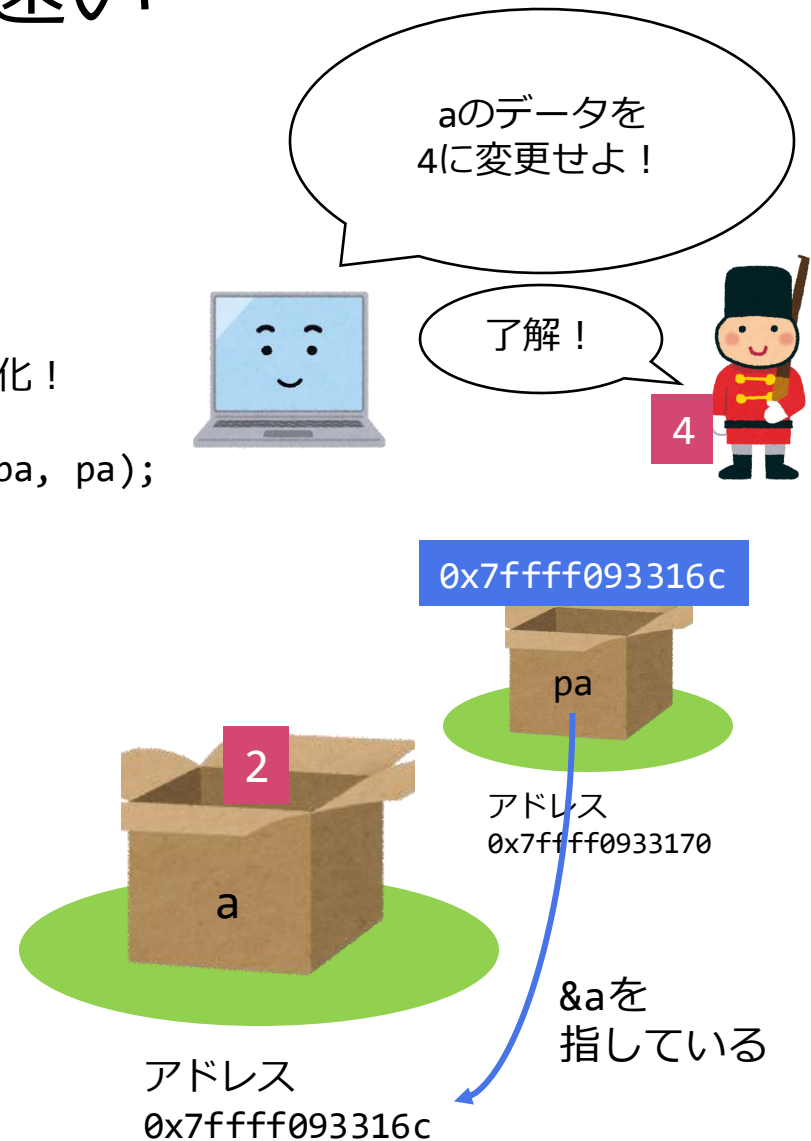
int main(){
    int a = 8;
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！

    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);

    pa = &a; // paにaのアドレスを代入
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    *pa = 2; // paが指している先のデータを2に変更
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);

    a = 4;
    printf("&a: %p\npa: %p\n", &a, pa);
    printf("a : %d\n*pa: %d\n", a, *pa);
}
```



# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

```
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

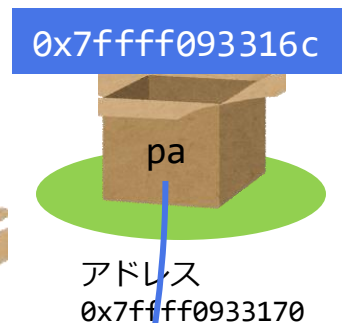
aのデータを  
4に変更せよ！



変更完了！



アドレス  
0x7ffff093316c



アドレス  
0x7ffff0933170

&aを  
指している



# ソースコード見た方が速い

```
#include <stdio.h>
```

```
int main(){  
    int a = 8;  
    int *pa = NULL; // ポインタ宣言時はNULLで初期化！
```

```
    printf("&a : %p\n&pa: %p\npa : %p\n", &a, &pa, pa);
```

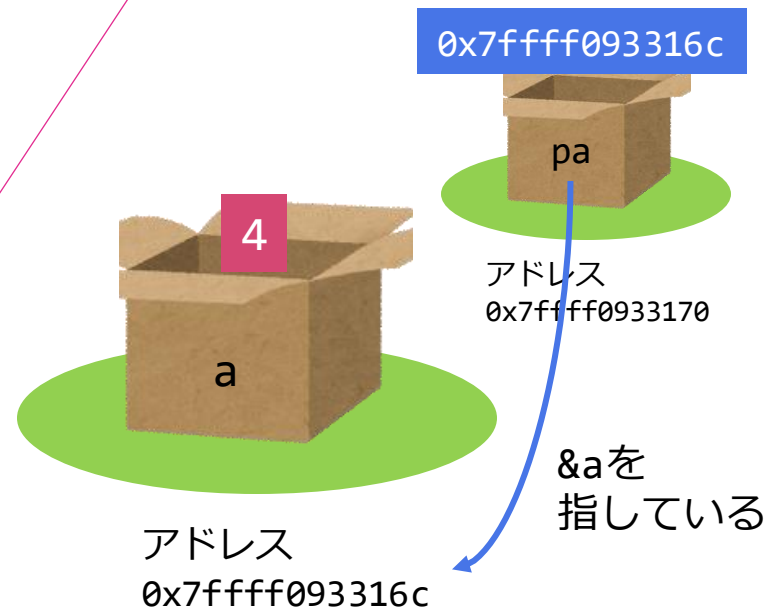
```
    pa = &a; // paにaのアドレスを代入  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    *pa = 2; // paが指している先のデータを2に変更  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
    a = 4;  
    printf("&a: %p\npa: %p\n", &a, pa);  
    printf("a : %d\n*pa: %d\n", a, *pa);
```

```
}
```

&a: 0x7ffff093316c  
pa: 0x7ffff093316c  
a : 4  
\*pa: 4      同じ！



雰囲気掴んで頂けたでしょうか...？



## + α : 直接参照と間接参照

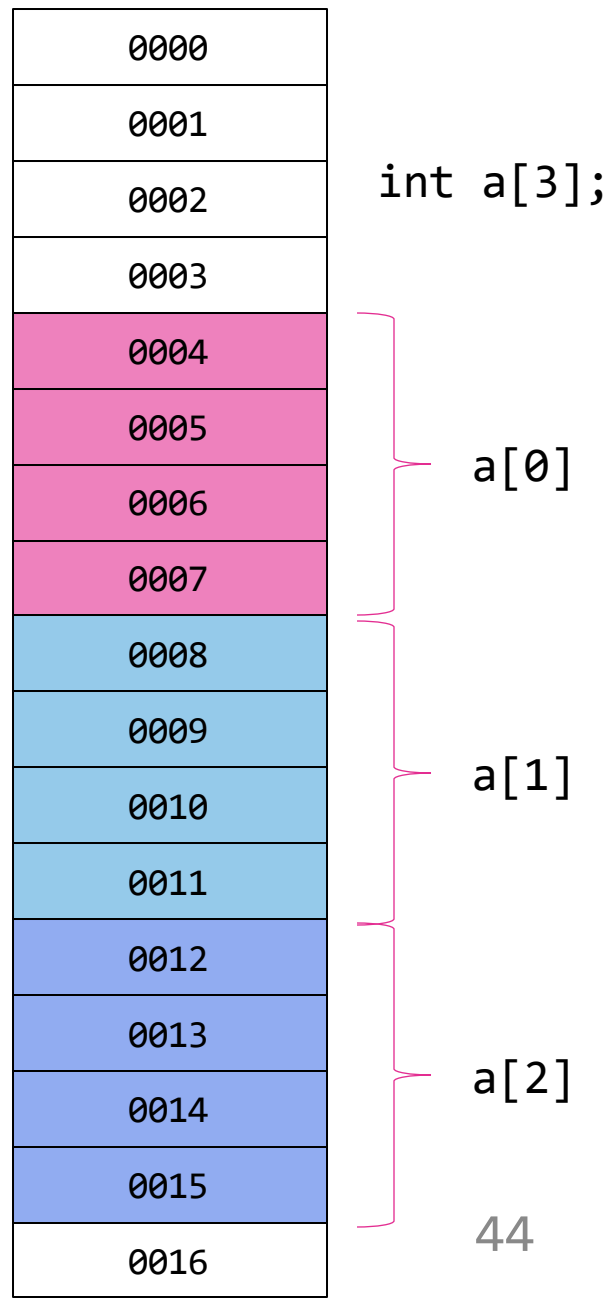
- `*pa = 4;` の方が `a = 2;` より兵隊さんの動きの数が多かったことにお気づき頂けたらどうか.
- これは, `a = 2;` が `a` に対する直接参照であり,  
`*pa = 4;` が `a` に対する間接参照だからである.
- 当然, 直接参照の方が速い

# 配列とポインタの関係1

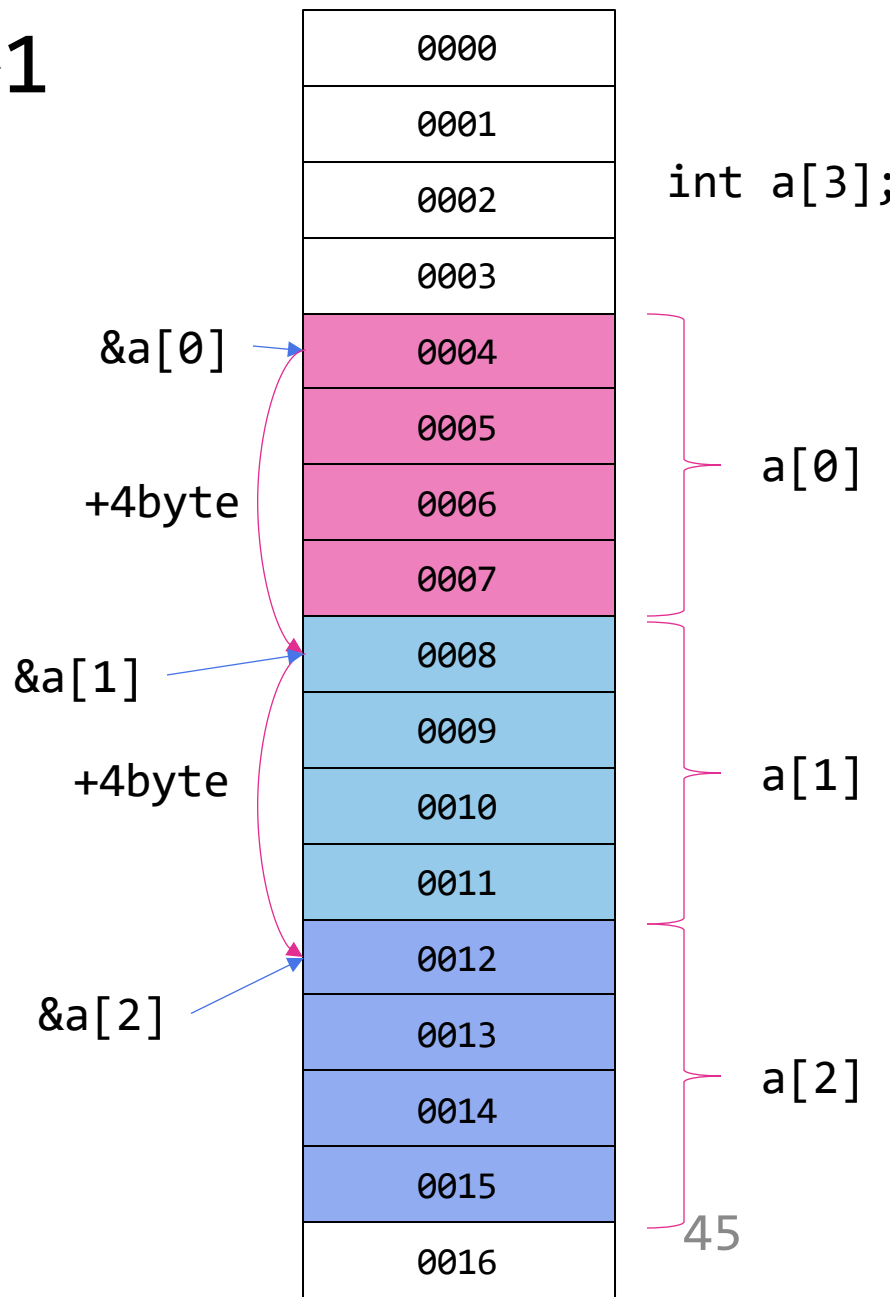
- 実はポインタを使って配列と同じようなことができます
- ところで、配列はメモリ上で連続していることを知っていますか？
  - 知らなかった人も、もう知ってるから大丈夫
  - 右図：要素数3のint型の配列a



何か気づきませんか？



# 配列とポインタの関係1



`&a[0]`, `&a[1]`, `&a[2]`は  
4byteずつずれている...?



# 配列とポインタの関係2

- &a[0]を起点として, ポインタを使って配列の各要素にアクセスできるのでは?

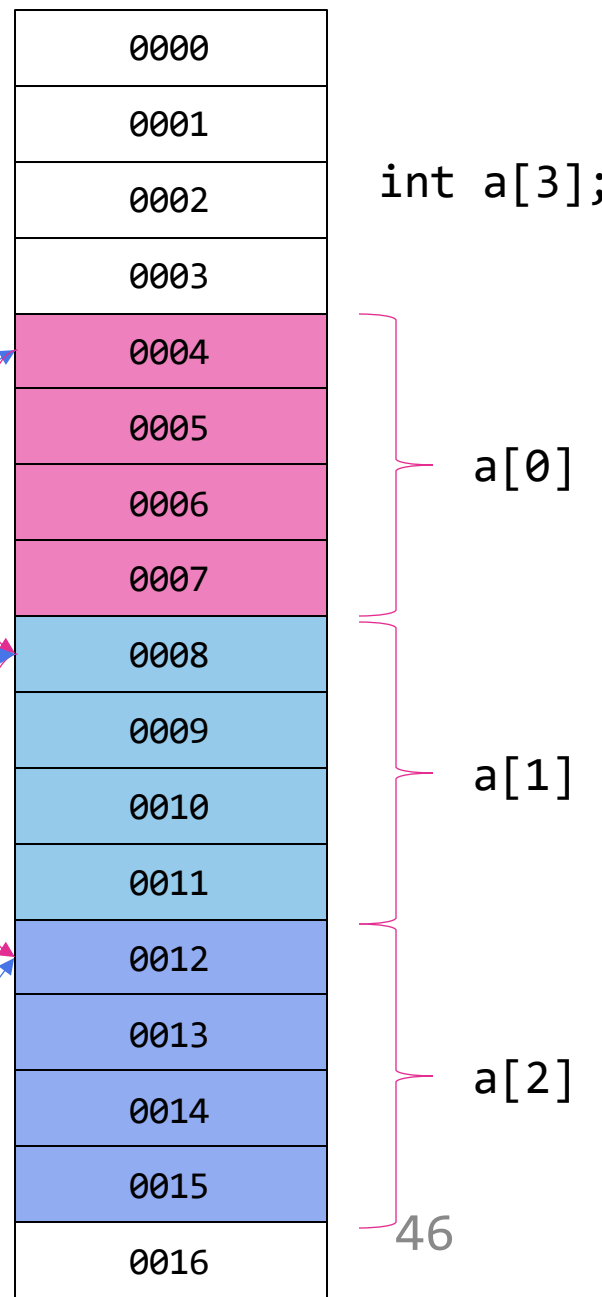
ptr = &a[0]

- たとえば...

```
int a[3] = {0, 1, 2};  
int *ptr = &a[0];  
printf("%d", *ptr); → 0  
printf("%d", *(ptr+4)); → 1  
printf("%d", *(ptr+8)); → 2
```

となるのでは?

ptr+8 = &a[2]



# 配列とポインタの関係2

- &a[0]を起点として, ポインタを使って配列の各要素にアクセスできるのでは?

- たとえば...

```
int a[3] = {0, 1, 2};
```

```
int *ptr = &a[0];
```

```
printf("%d\n", *ptr);
```

```
printf("%d\n", *ptr + 1);
```

```
printf("%d\n", *ptr + 2);
```

よくある  
間違い!

ptr = &a[0]

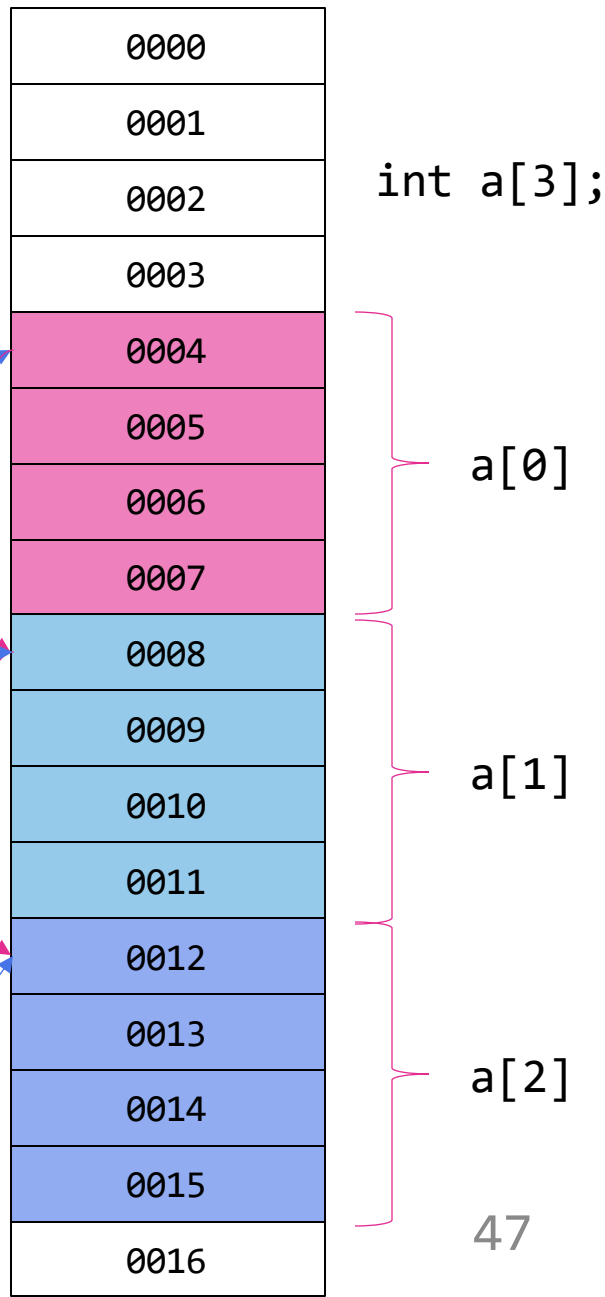
+4byte

&a[1]

→ 1 +4byte

&a[2]

とな...は?



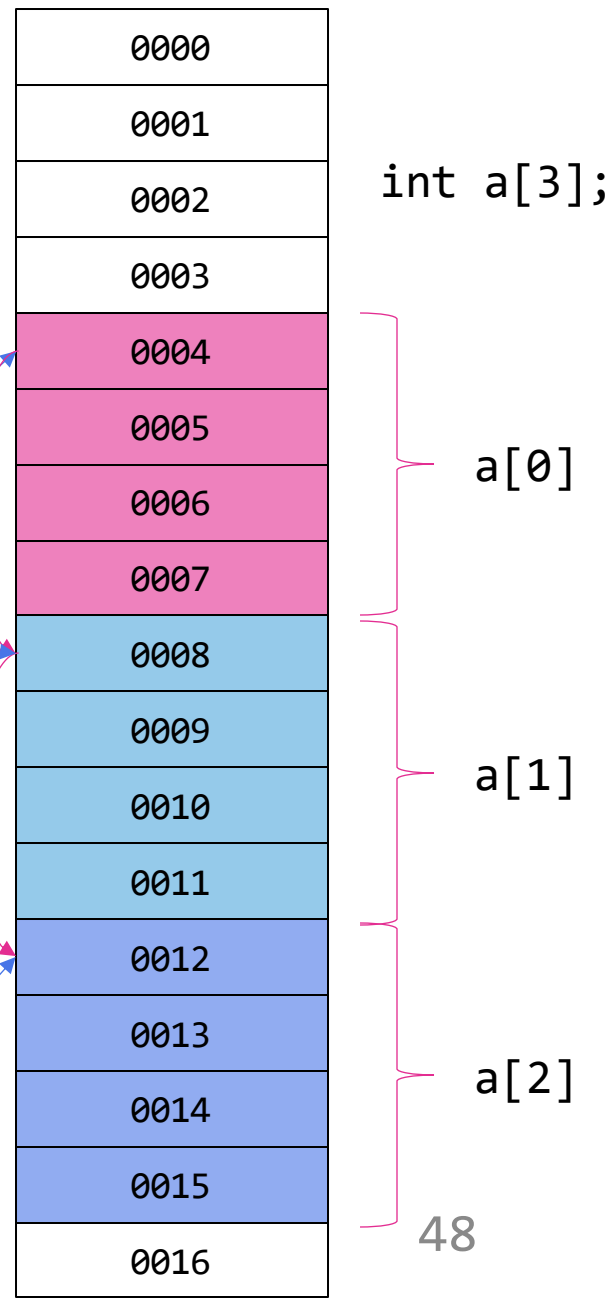
# 配列とポインタの関係3

- C言語は親切な言語なので(ほんとか?)  
int型のポインタで+1すると,  
4byteずらしてくれます.

- つまりこれが正解!

```
int a[3] = {0, 1, 2};  
int *ptr = &a[0];  
printf("%d", *ptr); → 0  
printf("%d", *(ptr+1)); → 1  
printf("%d", *(ptr+2)); → 2
```

ptr+2 = &a[2]





# 重要スライド！

## 配列とポインタの関係4

- 実は配列とポインタは同じではないですが、同じようなことができます.
- 例えば, こういうことが可能

```
int a[3] = {0, 1, 2};  
int i;  
for(i = 0; i < 3; i++){  
    printf("%d", *(a+i));  
}
```

aは&a[0]を指すポインタのようなもの

$a[i] \Leftrightarrow *(a+i)$

- 逆に, ポインタを一次元配列として使うことも可能

```
int a[3] = {0, 1, 2};  
int *p = &a[0];  
int i;  
for(i = 0; i < 3; i++){  
    printf("%d", p[i]);  
}
```

$*(p+i) \Leftrightarrow p[i]$

# 豆知識

- $a[i] = *(a+i)$  が成り立つ.
- $*(a+i) = *(i+a)$  が成り立つ.
- $*(i+a) = i[a]$  が成り立つ.
- よって,
- $a[i] = i[a]$  が成り立つ.

# 手を動かさないと覚ええない

- 下のプログラムをポインタを使って書いてみよう！
  - 題意が広いですが、ポインタを使っていればなんでも良いです

```
int main(){  
    int a[5] = {1, 2, 3, 4, 5};  
    int b[5] = {6, 7, 8, 9, 10};  
    int i;  
    for(i=0; i<5; i++) b[i] += a[i];  
}
```

まあ、言いたいことは分かったけど...

- 普通に `int a;` とか, `int a[10]` でよくない？
- そんなことはありません.
- 次ページからポインタならではの使い方を説明していきます.

# 関数に引数としてポインタを渡す

- 値渡しとポインタ渡しを知っていますか？

値渡し

```
int hoge(int a, int b){  
    a = 3;  
    return (a + b);  
}  
  
int main(){  
    int a = 1, b = 2;  
    int c = hoge(a, b);  
}
```

ポインタ渡し

```
int hoge(int *pa, int *pb){  
    *pa = 3;  
    return (*pa + *pb);  
}  
  
int main(){  
    int a = 1, b = 2;  
    int c = hoge(&a, &b);  
}
```

**決定的な違いがあります。何が違うでしょう？**

# こたえ

- 関数呼び出し後のaの値が違ふ

値渡し

```
int hoge(int a, int b){  
    a = 3;  
    return (a + b);  
}  
  
int main(){  
    int a = 1, b = 2;  
    int c = hoge(a, b);  
    printf("%d", a); → 1  
}
```

ポインタ渡し

```
int hoge(int *pa, int *pb){  
    *pa = 3;  
    return (*pa + *pb);  
}  
  
int main(){  
    int a = 1, b = 2;  
    int c = hoge(&a, &b);  
    printf("%d", a); → 3  
}
```

# 値渡しについて

- main関数のa,bとhoge関数のa,bは名前こそ同じですが、まったく別の変数です
- メモリ上でも全く別の場所に保管されています。
- そのため、hoge関数内でaに何をしようとmain関数のaは影響を受けません  
値渡し

```
int hoge(int a, int b){  
    a = 3;  
    return (a + b);  
}  
  
int main(){  
    int a = 1, b = 2;  
    int c = hoge(a, b);  
    printf("%d", a); → 1  
}
```

# ポインタ渡しについて

- 一方、ポインタ渡しでは、hoge関数のポインタpa, pbが指しているのはmain関数のa, bです
- すなわち、\*paに書き込むと、main関数のaの値が変更されます。

## ポインタ渡し

```
int hoge(int *pa, int *pb){
    *pa = 3;
    return (*pa + *pb);
}

int main(){
    int a = 1, b = 2;
    int c = hoge(&a, &b);
    printf("%d", a); → 3
}
```



# ポインタ渡しの使用例1

scanfもこのパターン  
scanf(“%d”, &n);

- 関数内で引数の値を変更したい場合
- Swap : aとbの値を入れ替える操作

```
void swap(int *pa, int *pb){  
    int tmp = *pa;  
    *pa = *pb;  
    *pb = tmp;  
}  
  
int main(){  
    int a = 1, b = 2;  
    swap(&a, &b);  
    printf(“a:%d,b:%d”, a, b) → a:2,b:1  
}
```

# ポインタ渡しの使用例2

- 配列を関数に渡す場合

```
void vec_add(int *a, int *b, int *c){
    for(int i = 0; i < 3; i++){
        c[i] = a[i] + b[i];
    }
}

int main(){
    int a[3] = {1, 2, 3};
    int b[3] = {4, 5, 6};
    int c[3];
    vec_add(a, b, c); // c = a + b
}
```

# 演習課題①

- 2つの配列a,bがmain関数内で宣言されています.
- a,bの各要素について次の操作を行う関数を作成してください.
  - $a[i] < b[i]$  ならば,  $a[i]$ と $b[i]$ を交換する.

- 例 :

```
int main(){
    int a[5] = {1, 2, 4, 6, 9};
    int b[5] = {2, 1, 3, 7, 9};
    func(?);
    // a = {2, 2, 4, 7, 9}
    // b = {1, 1, 3, 6, 9}
}
```

解答はあとでアップロードします

ポインタ完全理解おめでとう！

# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

# まえおき

- 構造体はポインタほど難しい概念ではありません。
- その割には便利な機能なので、ぜひ使い方を覚えましょう！

# 構造体とは？

- 複数のデータをひとまとめにして管理できるもの
- 例：学生データとして名前，学籍番号，年齢を管理したいとき

```
char student_name[NMAX][NAME_LENGTH];  
char student_id[NMAX][ID_LENGTH]  
int student_age[NMAX];
```

配列を3つ用意しますか？



# 構造体とは？

- 複数のデータをひとまとめにして管理できるもの
- 例：学生データとして名前，学籍番号，年齢を管理したいとき

```
char student_name[NMAX][NAME_LENGTH];  
char student_id[NMAX][ID_LENGTH]  
int student_age[NMAX];
```

配列を3つ用意しますか？

時と場合に応じてはその方が良い場合もあるけど  
構造体を使う方がスマート

# 構造体の宣言の仕方

- 構造体は次のような形で宣言します

```
struct 構造体タグ{  
    メンバ1;  
    メンバ2;  
}; ←忘れやすい
```

構造体タグ：構造体の名前  
メンバ：構造体が管理するデータ

- この時点では構造体を定義しただけで、まだ変数は存在しません.
- 変数は次のように宣言します.

```
struct 構造体タグ a;
```

# メンバへのアクセス

- 構造体のメンバにはメンバ演算子「.」を使ってアクセスできます.

```
struct data a;  
a.(メンバ名) = 1;
```

- また, 構造体のポインタからはアロー演算子「->」を使ってアクセスできます.

```
struct data a;  
struct data *ptr = &a;  
ptr->(メンバ名) = 1;
```

# ソースコード見た方が速い

```
#include<stdio.h>
#include<string.h>

struct StudentInfo{
    char name[100];
    char id[10];
    int age;
};

int main(){
    struct StudentInfo me;
    me.age = 23;
    strcpy(me.name, "Ebata Naoki");
    strcpy(me.id, "B9IMXXXX");

    printf("Name: ¥t%s¥n", me.name);
    printf("ID: ¥t%s¥n", me.id);
    printf("AGE: ¥t%d¥n", me.age);
    return 0;
}
```

+a : strcpyは文字列を代入する関数 (string.hにあるよ)  
C++だとstring型があって便利だね

# 手を動かさないと覚えない

- あなたは都市のデータを管理したいです.
- 次のような情報を持つ構造体を作成してください.
  - 都市の名前：文字列
  - 緯度：浮動小数点数
  - 経度：浮動小数点数
- 実際に仙台市のデータを代入して, 出力してみましょう
  - 名前：Sendai
  - 緯度：38.263033
  - 経度：140.871437

# 新たな型の定義

- 構造体を新しい型として定義することも可能です.

```
struct StudentInfo{  
    char name[100];  
    char id[10];  
    int age;  
};
```

```
typedef StudentInfo student;  
student me;
```

# 構造体の配列

- また、構造体の配列を作ることにも可能です。

```
struct StudentInfo M2[5];  
strcpy(M2[0].name, "Ebata Naoki");  
strcpy(M2[1].name, "Watanabe Koki");
```

## 演習課題②

- 100都市の名前/緯度/経度が記されたtxtファイルがあります.
  - それぞれ, 文字列, 浮動小数点数, 浮動小数点数です
  - txtファイルのLink ↓  
[https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise2/test.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise2/test.txt)
- この100都市の名前を, 緯度が小さい順に出力してください.
  - ただし, 緯度が等しい場合は, より経度が小さいものを先に出力するものとします.
  - 緯度と経度の両方が等しいデータの組は存在しません(たぶん)
  - 北緯/南緯, 東経/西経を気にする必要はありません.
- txtファイルには次のようなフォーマットで情報が記述されています.

```
都市名1 緯度1 経度1  
都市名2 緯度2 経度2  
...  
都市名100 緯度100 経度100
```



解答はあとでアップロードします

構造体完全理解おめでとう！

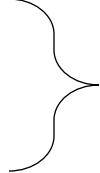
# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

# 実践あるのみ！

- ポインタと構造体を完全理解した読者諸兄には、その両方を使ったプログラムを書いてもらおうと思います.
- 具体的には、Stackを実装してもらいます

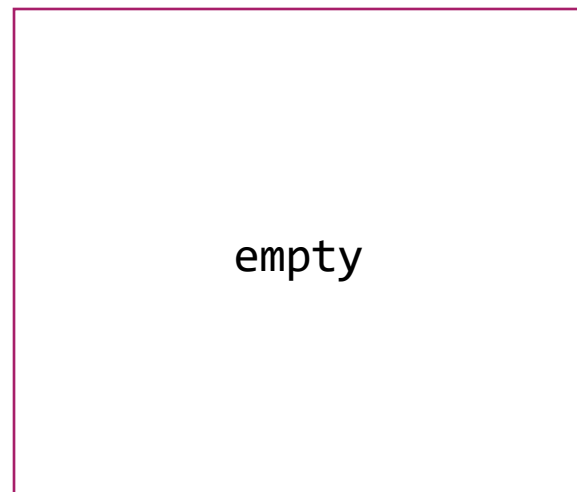
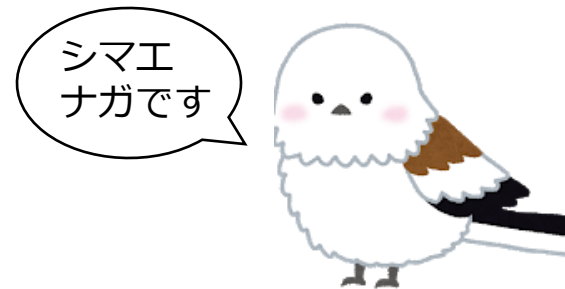
## 演習課題③

- 次の3機能を持つint型のStackを実装して下さい.
  - Push
  - Pop
  - Top

次ページ以降で説明
- 実装したStackを用いて, 次ページ以降で説明される「シマエナガくんの指示」を遂行するプログラムを作成して下さい

# Stackを知っていますか？

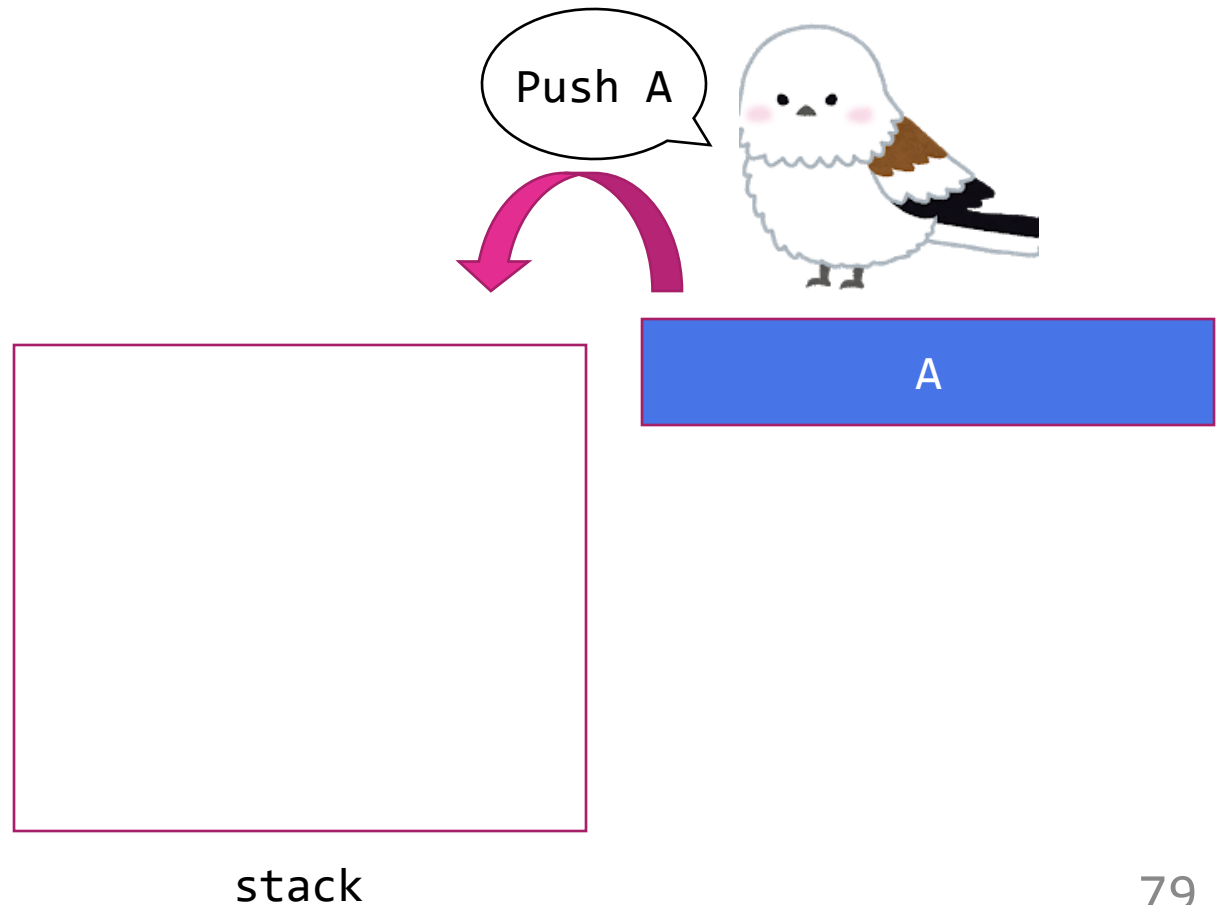
- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



stack

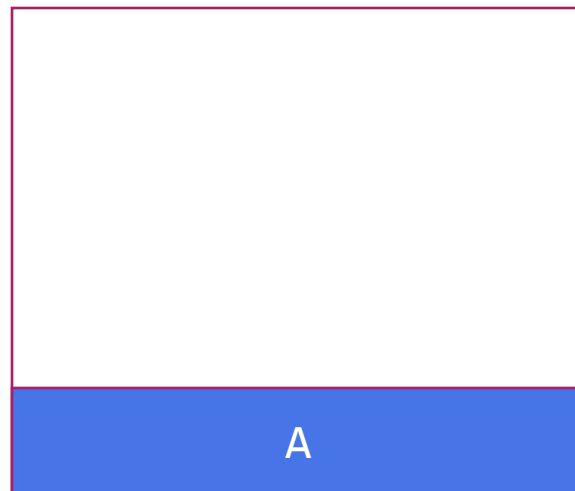
# Stackを知っていますか？

- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



# Stackを知っていますか？

- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...

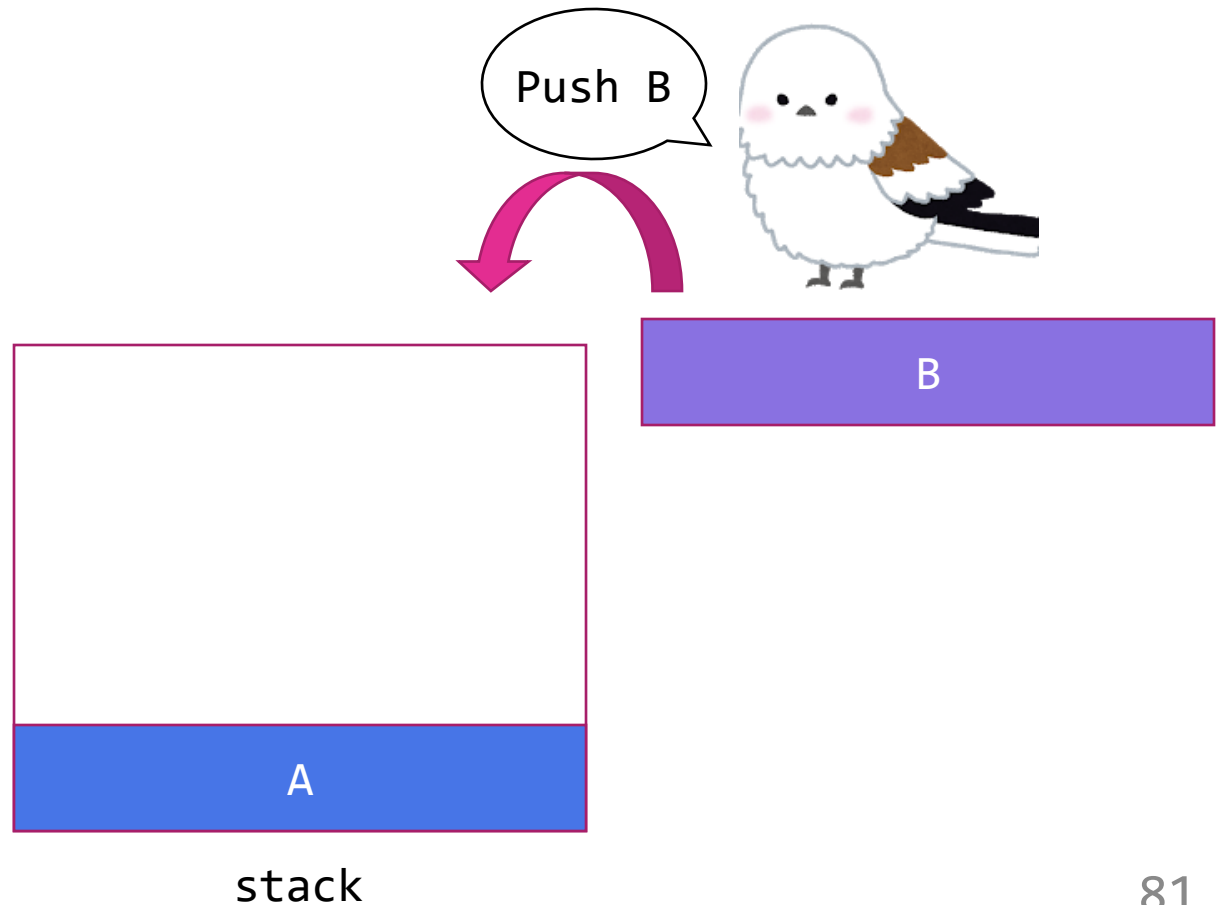


stack



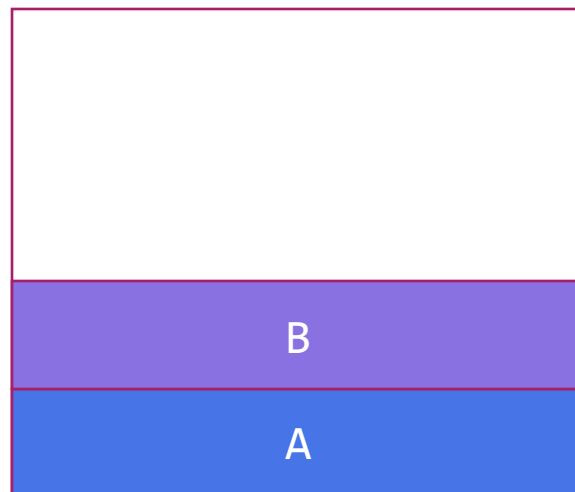
# Stackを知っていますか？

- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



# Stackを知っていますか？

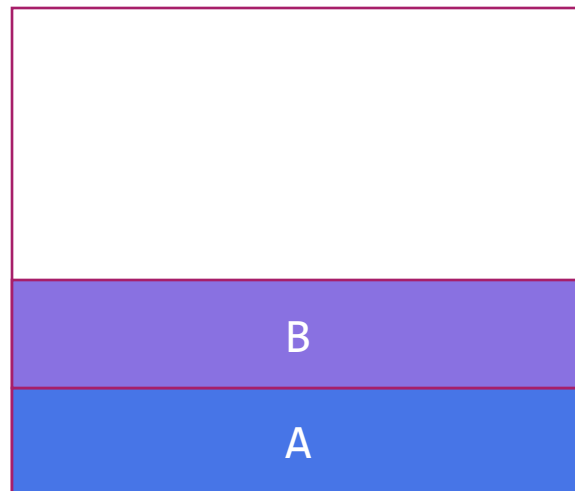
- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



stack

# Stackを知っていますか？

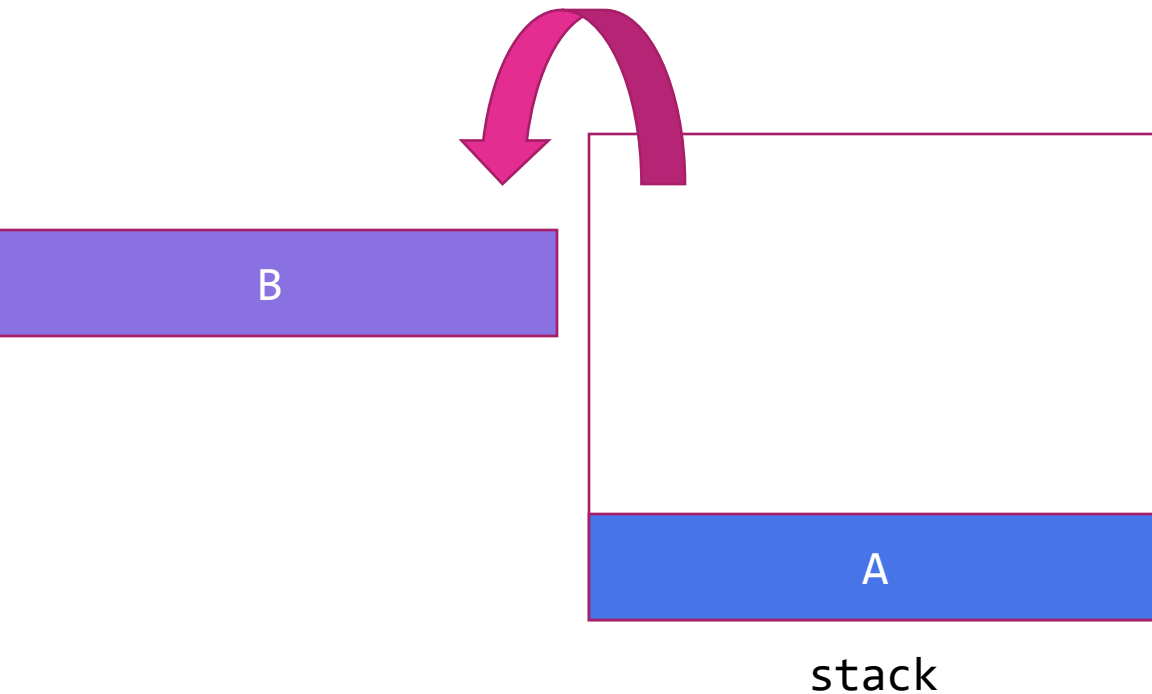
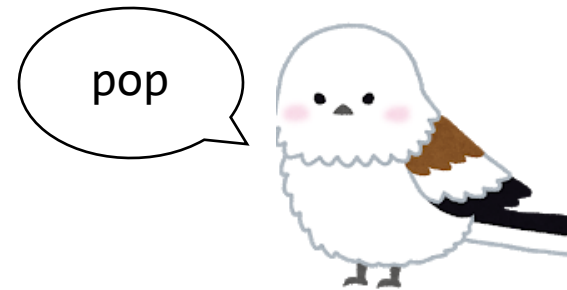
- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



stack

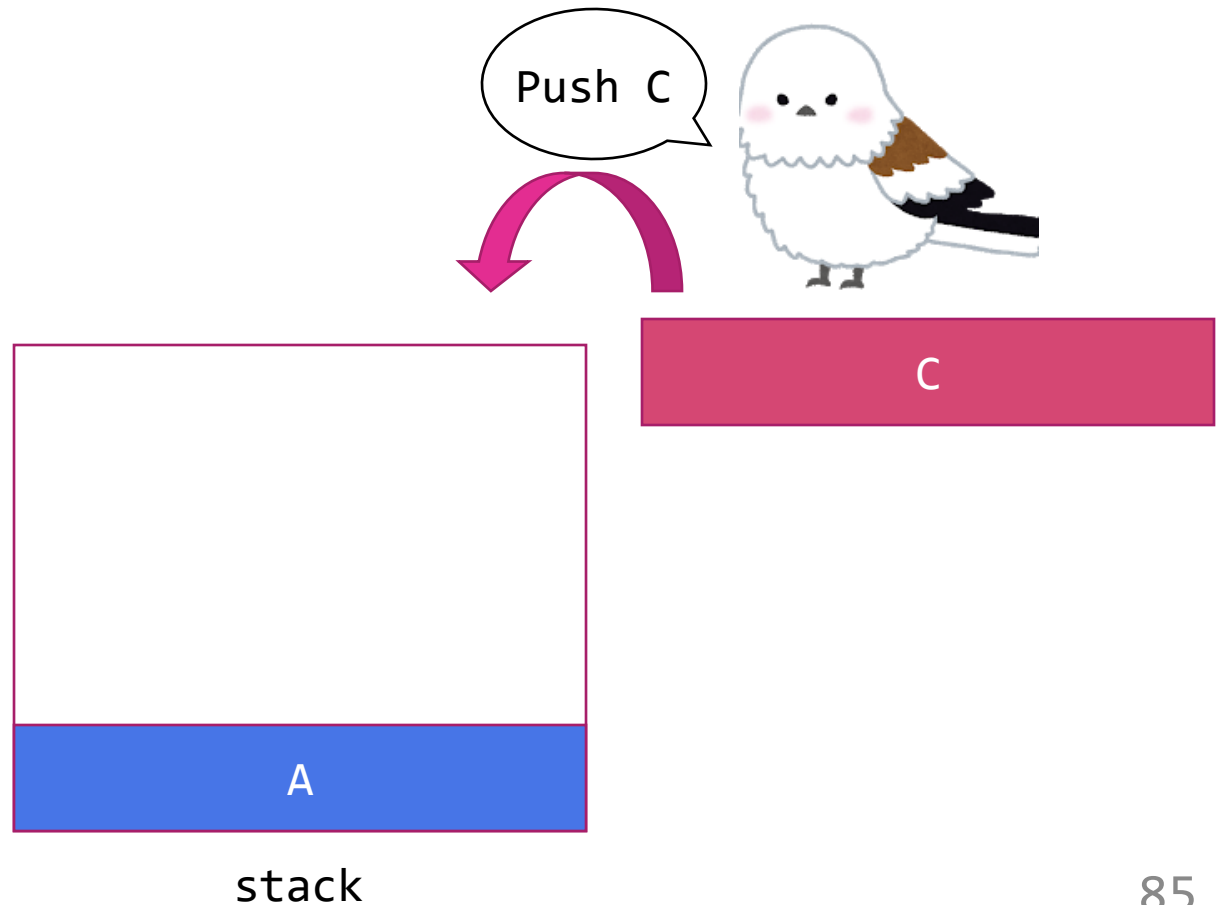
# Stackを知っていますか？

- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



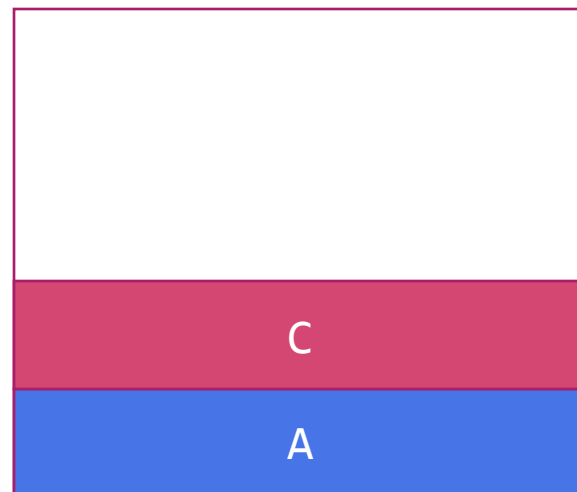
# Stackを知っていますか？

- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



# Stackを知っていますか？

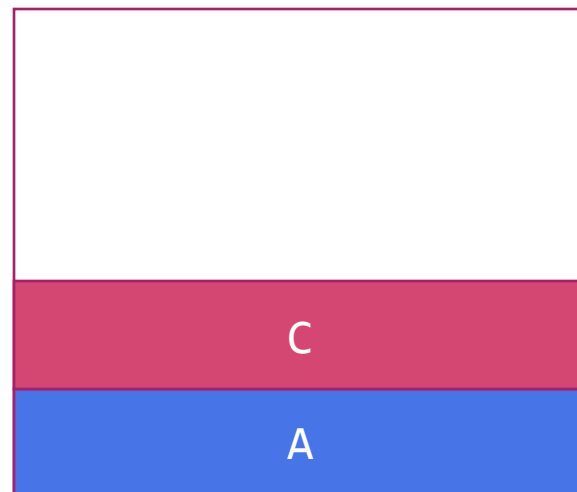
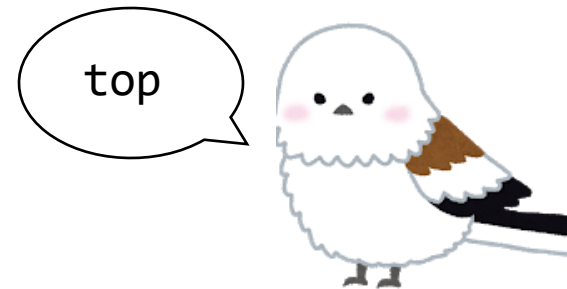
- Stackは先入れ後出し(FILO)のデータ構造です.
- 例えば...



stack

# Top

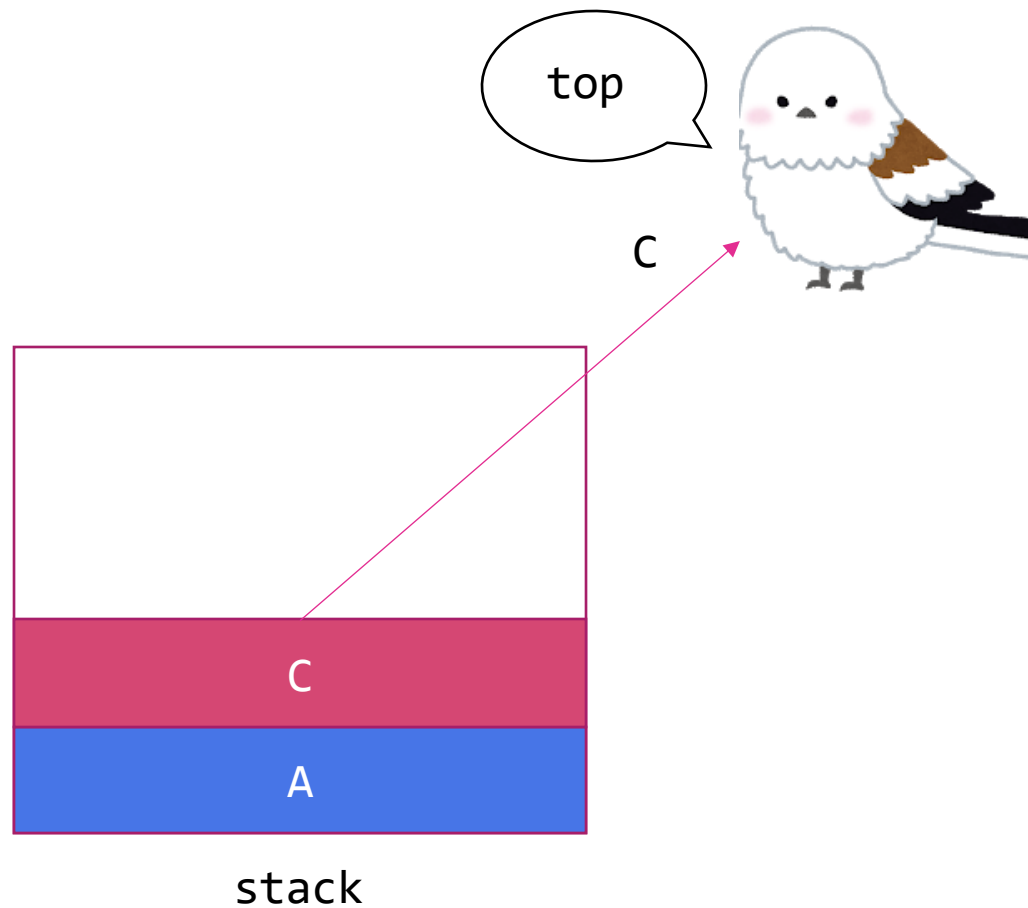
- Topはstackの一番上にあるデータを返す関数です
- 例えば...



stack

# Top

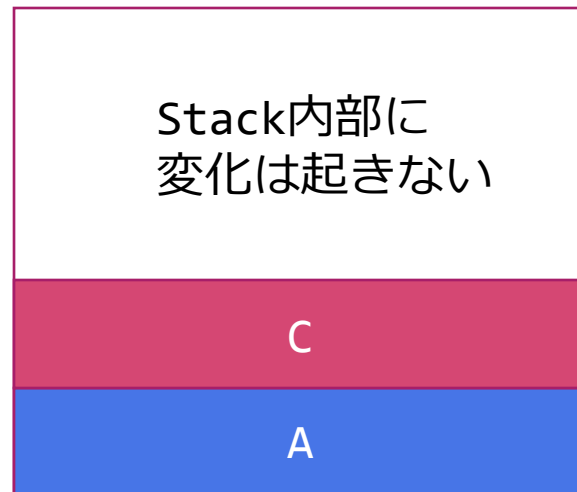
- Topはstackの一番上にあるデータを返す関数です
- 例えば...





# Top

- Topはstackの一番上にあるデータを返す関数です
- 例えば...



stack

# シマエナガくんの指示に従おう！



- シマエナガくんは繰り返し{push N, pop, top}のいずれかの操作を指示します。それぞれ、次の操作をしてください。
  - push N : 整数Nをpushしてください
  - pop : popしてください
  - top : Stackの一番上にある値を標準出力に表示してください。
- はじめ, stackは空です。

# 例



push 1

push 2

top

pop

top



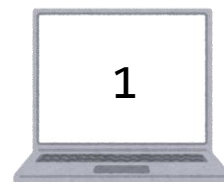
stack



stack



stack



# シマエナガさんの指示の受け取り方1

- シマエナガさんの指示はtxtファイルに記録されています.
  - 1行目に指示の数
  - 2行目以降に指示が書いてあります.
- あなたはこれをファイルから直接読み込んでも良いですし、標準入力から読み込んでも構いません.
  - 標準入力から読み込む場合：
    - `$ cat [file name] | ./a.out`

```
5
push 1
push 2
top
pop
top
```



# シマエナガくんの指示の受け取り方2

- ファイルは全部で3つあります.
  - [https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise3/test1.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise3/test1.txt)
  - [https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise3/test2.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise3/test2.txt)
  - [https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise3/test3.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise3/test3.txt)
- また、想定する出力が下記のファイルに書いてあるので、diffコマンドを使って各自答え合わせしよう！
  - [https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise3/ans1.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise3/ans1.txt)
  - [https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise3/ans2.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise3/ans2.txt)
  - [https://github.com/EbataNaoki/C\\_training/blob/master/test\\_case/exercise3/ans3.txt](https://github.com/EbataNaoki/C_training/blob/master/test_case/exercise3/ans3.txt)

# 注意点

- あくまで、ポインタと構造体の実践課題です.
  - ポインタと構造体を使って、汎用性のあるコードを書こう！
- シマエナガくんは親切なので、stackが空のときには、popやtopを指示しません.
- また、指示の総数  $\leq 100$  です.

解答はあとでアップロードします

# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門



# 教えるもの

- 動的メモリ確保
- プリプロセッサ
- Makefile
- Shell Script

ざっくりと出来ることを教えていきます。  
細かな使い方はGoogleで検索してね！

# 教えるもの

- 動的メモリ確保
- プリプロセッサ
- Makefile
- Shell Script

ざっくりと出来ることを教えていきます。  
細かな使い方はGoogleで検索してね！

# 実行時まで配列サイズが分からないとき

- プログラム実行時まで配列サイズが分からない場合があります
- そんなときは, malloc関数を使うのが一般的です.
  - 例: `int *a = (int *)malloc(N * sizeof(int));`
- malloc関数は引数byteメモリを確保し, その先頭を指すvoid型のポインタを返します.
  - なので, 適切に型キャストしてあげる必要があります.

# mallocのお作法1

- 引数のサイズなどによってはメモリの確保に失敗する場合があります.
- その場合, mallocはNULLを返します.
- そのため, お作法として, 次のような書き方をするのが良いとされています (要出典)

```
int *a = (int *)malloc(N * sizeof(int));  
if(a == NULL){  
    printf("ERROR: Failed to allocate memory¥n");  
    exit(1);  
}
```

# mallocのお作法2

- 確保したメモリ領域は使い終わったら解放するのが良いとされています（要出典）。
- なので、次のようなコードを書きましょう。

```
int *a = (int *)malloc(N * sizeof(int));  
if(a == NULL){  
    printf("ERROR: Failed to allocate memory¥n");  
    exit(1);  
}
```

```
/* do something... */
```

```
free(a);  
a = NULL;
```

# 教えるもの

- 動的メモリ確保
- プリプロセッサ
- Makefile
- Shell Script

ざっくりと出来ることを教えていきます。  
細かな使い方はGoogleで検索してね！

# プリプロセッサを知っていますか？

- 見たことがない人はいないと思います.
- こういうやつです.
  - `#include`
  - `#define`
  - `#ifdef`
  - `#pragma`

# プリプロセッサとは

- コンパイルの前段階として処理されるもの
- `#define NMAX 500` と書いてあれば, プログラム中のNMAXが500に置換されたあと, コンパイルが始まる

```
#define NMAX 500
int main(){
    int a[NMAX];
    int i;
    for(i=0; i<NMAX; i++) a[i] = i;
}
```



# #define

- コンパイル前に置換される

マクロと呼びます

```
#define NMAX 500
int main(){
    int a[NMAX];
    int i;
    for(i=0; i<NMAX; i++) a[i] = i;
}
```



```
int main(){
    int a[500];
    int i;
    for(i=0; i<500; i++) a[i] = i;
}
```

# #define

- defineはコンパイルオプションとして定義することも可能
- D(マクロ名)=(指定値)
  - gcc -DNMAX=500 main.c

# #define

- また, #defineには引数を持たせることができる

```
#define REP(i,start,end) for((i)=(start); (i)<(end); (i)++)

int main(){
    int sum = 0;
    int i;
    REP(i, 1, 101) sum += i;
}
```



```
int main(){
    int sum = 0;
    int i;
    for(i=1; i<101; i++) sum += i;
}
```

# #ifdef

- 次のように使います.

```
#define DEBUG
```

```
#define NMAX 500
```

```
int main(){
```

```
    int a[NMAX], b[NMAX];
```

```
    int i;
```

```
    for(i=0; i<NMAX; i++){
```

```
        a[i] = i;
```

```
        b[i] = -i;
```

```
#ifdef DEBUG
```

```
    printf("a[%d]: %d\n", i, a[i]);
```


```
#endif
```

```
}
```

```
    for(i=0; i<NMAX; i++) a[i] += b[i];
```

```
}
```

DEBUGが定義されているときのみ  
コンパイルされる




# #ifdef

- 次のように使います.

```
#define NMAX 500
int main(){
    int a[NMAX], b[NMAX];
    int i;
    for(i=0; i<NMAX; i++){
        a[i] = i;
        b[i] = -i;
#ifdef DEBUG
        printf("a[%d]: %d\n", i, a[i]);
#endif
    }
    for(i=0; i<NMAX; i++) a[i] += b[i];
}
```

DEBUGが定義されていないときは  
コンパイルされない



# まあ、これくらい覚えておけば大丈夫

- #から始まる知らないものの遭遇したら、適宜Googleで検索してください.

# 教えるもの

- 動的メモリ確保
- プリプロセッサ
- Makefile
- Shell Script

ざっくりと出来ることを教えていきます。  
細かな使い方はGoogleで検索してね！

# Makefileを作ろうね！

- Makefileを作るのはちょっとだけ面倒くさいです.
- しかし, 後から見返すときのため, あるいは, あなたが書いたプログラムを使う他の人のために, Makefileを作ることを習慣づけましょう！



# で, Makefileって何？

- プログラムのコンパイルの仕方を記述してあるもの
  - `$ make` と叩くだけで, 記述に従ってコンパイルしてくれる.

# どんなときに使うの？

- こんなコンパイルオプションをいちいち書きたくない...
  - `gcc -O3 -fopenmp -DNMAX=50 -DDEBUG -mavx -lm -o exe main.c`
- 分割してコンパイルしてリンクする場合, 1つずつコンパイルするのはめんどくさい
  - `gcc -c func1.c`
  - `gcc -c func2.c`
  - `gcc -c func3.c`
  - `gcc -o exe func1.o func2.o func3.o main.c`
- 「あれ, これどうやってコンパイルするんだっけ？」となりそうなとき

# Makefileの書き方

## Makefile

```
CC=gcc
CFLAGS=-O3 -fopenmp
SRC=main.c
EXE=exe

default: SRC
    $(CC) $(CFLAGS) $(SRC) -o $(EXE)

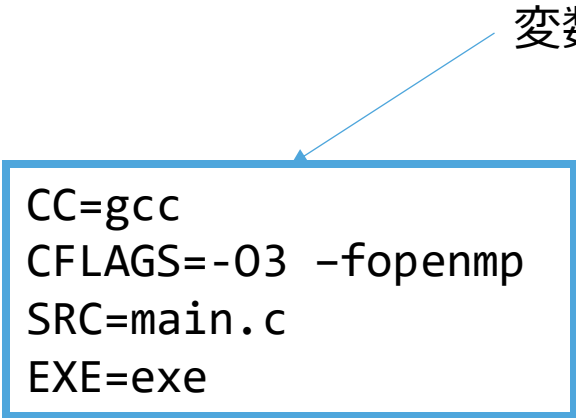
clean:
    rm -f $(EXE)
```

- 普通, Makefileは, Makefileかmakefileという名前のファイルに記述します.

# Makefileの書き方

- 例

```
CC=gcc  
CFLAGS=-O3 -fopenmp  
SRC=main.c  
EXE=exe
```



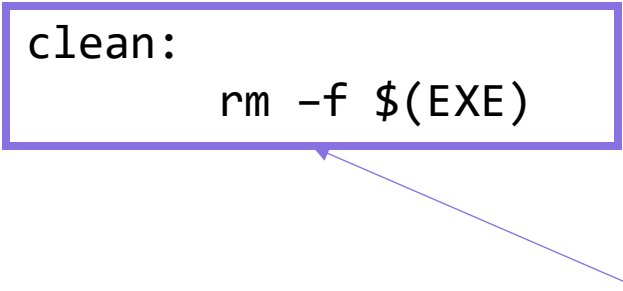
変数

```
default: SRC  
        $(CC) $(CFLAGS) $(SRC) -o $(EXE)
```



コンパイルの仕方

```
clean:  
        rm -f $(EXE)
```



コンパイルの結果できた実行ファイルを削除する方法

# Makefileの書き方

- 例

```
CC=gcc  
CFLAGS=-O3 -fopenmp  
SRC=main.c  
EXE=exe
```

変数

コンパイルの仕方

```
default: main.c  
          gcc -O3 -fopenmp main.c exe  
          $(CC) $(CFLAGS) $(SRC) -o $(EXE)
```

```
clean:  
      rm -f exe  
      $(RM) $(EXE)
```

コンパイルの結果できた実行ファイルを削除する方法 117

# Makefileの書き方

- 例

```
CC=gcc
CFLAGS=-O3 -fopenmp
SRC=main.c
EXE=exe
```

変数

コンパイルの仕方

依存するファイル

ターゲット

```
default: main.c$(SRC)
        gcc -O3 -fopenmp main.c$(SRC) -o exe$(EXE)
```

```
clean:
```

```
        rm -f exe$(EXE)
```

コンパイルの結果できた実行ファイルを削除する方法

# コンパイルの仕方

- `$ make default` で, defaultに記述されたコンパイルの方法に従ってコンパイルが行われる.
  - `$ make` とだけ叩くと, Makefileの一番上に記述されているターゲットのルールに従ってコンパイルされる.
- ただし, 常にコンパイルが行われるわけではない.
- 前回makeしたとき以降に, 依存するファイルに変更があった場合のみコンパイルする.

# make clean

- `clean` には, `$ make` によって生成された実行ファイルを削除する方法が記述されている.



# 他にもいろいろな書き方があるけど...

- 適宜Googleで検索してください.

# 教えるもの

- 動的メモリ確保
- プリプロセッサ
- Makefile
- Shell Script

ざっくりと出来ることを教えていきます。  
細かな使い方はGoogleで検索してね！

# Shell Scriptを，教えたかった...

- この講習で教えるには，教えることが多すぎるので断念します
- Terminal上での一連の動作を記述できるプログラムだと考えてよいです.

hello.sh

```
#!/bin/bash  
  
echo "Hello world" > hello.txt  
cp hello.txt world.txt  
cat world.txt
```

```
$ ./hello.sh  
と叩くと,  
$ Hello World  
と表示され, さらに,  
hello.txtとworld.txtが  
できています.
```

めっちゃめっちゃ便利なので，ぜひ勉強してみてください...

# 本日の流れ

1. なぜC言語？
2. ウォーミングアップ
3. ポインタ
4. 構造体
5. 実践：stackの実装
6. 覚えておくと便利な機能
7. プログラム高速化入門

# プログラムを高速化しよう！

- プログラム高速化はHPC分野における最も重要なテーマの1つ
- HPCのアプリケーションは総じて実行時間が非常に長いいため高速化の恩恵が著しい
  - 例：プログラムの実行時間を1/2にできた！
    - 10秒かかるプログラムなら5秒の節約
    - 1時間かかるプログラムなら30分の節約
    - 1日かかるプログラムなら12時間の節約
    - 1か月かかるプログラムなら2週間の節約...

# どうやって高速化するの？

## 1. まず、アルゴリズムを再考察しましょう

- $O(N^2)$ を $O(N\log N)$ にできるなら、それが最も効果的
- これは、その道の専門家のお仕事
  - 流体シミュレーションなら流体力学の専門家
  - 機械学習なら機械学習の専門家

## 2. それでも遅いなら、あの手この手で定数倍高速化

- 我々のお仕事
- やること
  - 並列化
  - ベクトル化
  - ハードウェアの特性を活かしたプログラム最適化
  - 無駄な演算の削除
  - 演算精度を落とす
  - アクセラレータにオフロード
  - などなどなど...

# どうやって高速化するの？

## 1. まず、アルゴリズムを再考察しましょう

- $O(N^2)$ を $O(N \log N)$ にできるなら、それが最も効果的
- これは、その道の専門家のお仕事
  - 流体シミュレーションなら流体力学の専門家
  - 機械学習なら機械学習の専門家

## 2. それでも遅いなら、あの手この手で定数倍高速化

- 我々のお仕事
- やること
  - 並列化
  - ベクトル化

**コンピュータに優しいコードを書く！！**

- 演算精度を落とす
- アクセラレータにオフロード
- などなどなど...

# 今日やること

- 今日学んだ範囲で出来る高速化テクニックの実践
- 残念ながら、時間の都合上、ほんの少しのテクニックしか教えられません...
  - 今日はお試しです
- 並列化は佐々木くんの講義を待たれよ



# 茶番

- あなたのもとに以下の依頼が届きました.

私は卒業研究で以下のプログラムを利用しています.

[https://github.com/EbataNaoki/C\\_training/tree/master/acceleration/0\\_original](https://github.com/EbataNaoki/C_training/tree/master/acceleration/0_original)

このプログラムの実行にはおおよそ $T$ 秒かかることが分かっています.

あと1回, このプログラムを実行すれば, 私の卒業研究は完成します.

しかし, 困ったことに卒業研究の提出締め切りは $T/2$ 秒後です.

このままでは, 私は留年してしまいます.

そこで, あなたにお願いです.

このプログラムを高速化して,  $T/2$ 秒以内に終了するようにして下さい.

多少は精度が落ちても構いません.

私の卒業がかかっています. よろしくお願いします.

# プログラムの説明1

- 5万個の天体の加速度を計算するプログラム
- 各天体の加速度は次の式で導出できます.
  - 衝突は考慮しない.

$$\mathbf{a}_i = \sum_{j=1, j \neq i}^{50000} m_j \mathbf{g} \frac{\mathbf{r}_{ij}}{|\mathbf{r}_{ij}|^3}$$

$\mathbf{a}_i$  : 天体の加速度

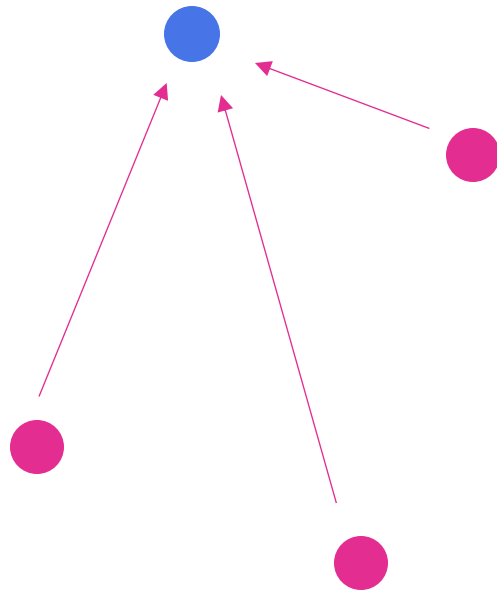
$m_j$  : 質量

$\mathbf{g}$  : 重力加速度

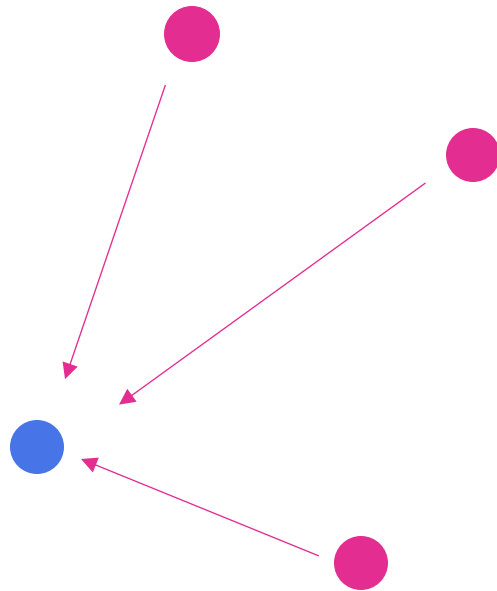
$\mathbf{r}_{ij}$  : 天体の相対的な位置ベクトル

- このプログラムでは、各天体が他の天体から受ける引力を累積計算し、各天体の加速度を導出します.

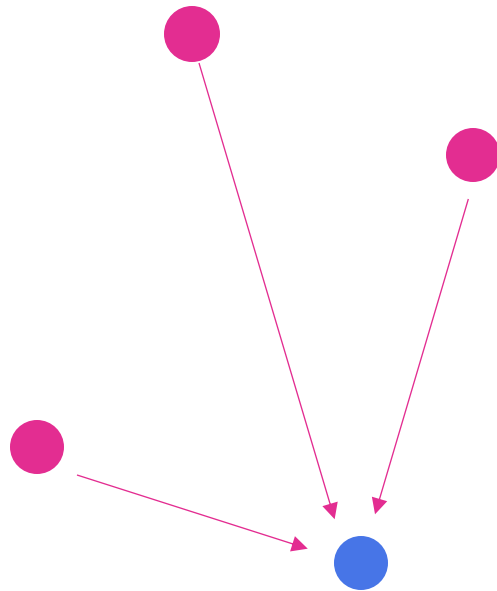
つまり...



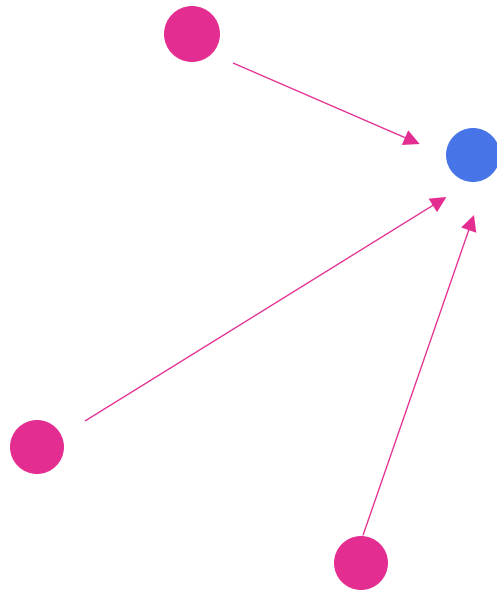
つまり...



つまり...



つまり...



こんな感じ

# プログラムの説明2

- 入力

- N個の天体の情報はtxtファイルに以下のフォーマットで記述されています.
- ファイル名 : input.txt

```
天体ID1 質量1 x座標1 y座標1 z座標1  
天体ID2 質量2 x座標2 y座標2 z座標2  
天体ID3 質量3 x座標3 y座標3 z座標3  
...
```

- 出力

- 天体IDとx,y,z方向の加速度をそれぞれファイルに書き出しします.
- ファイル名 : output.txt

# まず、実行時間を計測してみる

- まず、オリジナルコードの実行時間を計測してみましょう。

## 1. コンパイルする

- `$ make`
- スパコンで実行するようなプログラムにはだいたいMakefileが付いています

## 2. 実行する

- `exe`という名前の実行ファイルができたと思います
- `time`コマンドを使って実行時間を計測してみましょう
  - `$ time ./exe`



# 計測結果

- 次のような結果が出力されたかと思います.

real	0m13.022s	← 実際の経過時間
user	0m12.995s	
sys	0m0.008s	

- 今回の目標は実行時間を1/2にすることだったので,  
目標時間は6.5secです.
  - 注: 目標時間は実行環境によって異なります.

演習課題が早く終わって来た人は, 自力で高速化してみよう!

並列化しなくても(たぶん)達成できます.

(sharkでは, 並列化なしで4.5秒に短縮できることを確認済)

# Step1:演算の共通化

- これは分かりやすい発想だと思います.
- オリジナルでは, 質点間の距離計算が $N(N-1)$ 回行われていますが, これは本当に必要でしょうか?
- 実はこれを半分に削減することが可能です.
  - 2質点の組み合わせは $N(N-1)/2$ 通りしかありません.

# こんな感じ

```
// main loop
for(i = 0; i < NMAX; i++){
    for(j = 0; j < NMAX; j++){
        if(i == j) continue;
        float dr = 0;
        for(k = 0; k < 3; k++){
            dr += (mp[i].r[k] - mp[j].r[k]) * (mp[i].r[k] - mp[j].r[k]);
        }
        dr = sqrtf(dr);
        for(k = 0; k < 3; k++){
            mp[i].a[k] += mp[j].m * (mp[j].r[k] - mp[i].r[k]) / (dr * dr * dr);
        }
    }
}
```



```
// main loop
for(i = 0; i < NMAX-1; i++){
    for(j = i+1; j < NMAX; j++){
        float dr = 0;
        for(k = 0; k < 3; k++){
            dr += (mp[i].r[k] - mp[j].r[k]) * (mp[i].r[k] - mp[j].r[k]);
        }
        dr = sqrtf(dr);
        for(k = 0; k < 3; k++){
            mp[i].a[k] += mp[j].m * (mp[j].r[k] - mp[i].r[k]) / (dr * dr * dr);
            mp[j].a[k] += mp[i].m * (mp[i].r[k] - mp[j].r[k]) / (dr * dr * dr);
        }
    }
}
```

# 実行結果

	実行時間 sec
オリジナル	13.0
演算の共通化	8.4

- まあ, こんなもんでしょう

## Step2: AoSからSoAへ

- これはメジャーな高速化テクニックです
- データの空間的局所性を考えたとき, AoSはあまりCPU向きのデータ構造ではありません
  - AoS : Array of Structure
  - SoA : Structure of Array
- 詳しく説明したいという気持ちはあるが, スライドつくるのめんどくさい...
- 今回は特に, 質点IDとかいう意味不明データが構造体メンバにあるので, SoAにするべき

# こんな感じ

```
struct Mass_Point{  
    float r[3]; // r[0]:x, r[1]:y, r[2]:z  
    float a[3];  
    float m;  
    char id[100];  
};  
struct Mass_Point mp[NMAX];
```



```
float r[NMAX][3];  
float a[NMAX][3];  
float m[NMAX];  
char id[NMAX][100];
```

ソースコード

[https://github.com/EbataNaoki/C\\_training/tree/master/acceleration/2\\_AoS\\_to\\_SoA](https://github.com/EbataNaoki/C_training/tree/master/acceleration/2_AoS_to_SoA)

# 実行結果

	実行時間 sec
オリジナル	13.0
演算の共通化	8.4
AoS → SoA	8.1

- 正直，渋いですね...
- これは，このプログラムは演算がボトルネックになっているからです.
- メモリ転送がボトルネックになっているプログラムではAoSからSoAの変更は効果的です（たぶん）

## Step3 : コンパイルオプションの暴力

- ここで, Makefileを見てみると, 最適化系のコンパイルオプションが-O2だけなことが分かります.
- 多少, 精度は落ちてても構わないとあるので以下を追加しました
- -mavx
  - ベクトル命令の仕様を許可
- -ffast-math
  - 高速算術演算を使用 (精度が落ちる場合あり)
- -O2 → -O3
  - 副作用を伴う最適化を許可
- gccは優秀なので, これだけで結構性能が改善します

ソースコード

[https://github.com/EbataNaoki/C\\_training/tree/master/acceleration/3\\_compile\\_option](https://github.com/EbataNaoki/C_training/tree/master/acceleration/3_compile_option)



# 実行結果

	実行時間 sec
オリジナル	13.0
演算の共通化	8.4
AoS → SoA	8.1
コンパイルオプション	5.8

- GCC優秀ですね
- 早くもノルマ達成できてしまった...

## Step4 : ループ分割

- ループを分割すると、ベクトル化やパイプライン効率が改善する場合があります。
- これは、コンパイラがループ単位でベクトル化やパイプラインスケジューリングを行うからです。
  - 1つのループが複雑になりすぎていると、コンパイラがうまく最適化できないことがあります。

# こんな感じ

```
// main loop
for(i = 0; i < NMAX-1; i++){
    for(j = i+1; j < NMAX; j++){
        float dr = 0;
        for(k = 0; k < 3; k++){
            dr += (r[i][k] - r[j][k]) * (r[i][k] - r[j][k]);
        }
        dr = sqrtf(dr);
        for(k = 0; k < 3; k++){
            a[i][k] += m[j] * (r[j][k] - r[i][k]) / (dr * dr * dr);
            a[j][k] += m[i] * (r[i][k] - r[j][k]) / (dr * dr * dr);
        }
    }
}
```



```
// main loop
float dr[NMAX];
for(i = 0; i < NMAX-1; i++){
    for(j = i+1; j < NMAX; j++){
        dr[j] = 0;
        for(k = 0; k < 3; k++){
            dr[j] += (r[i][k] - r[j][k]) * (r[i][k] - r[j][k]);
        }
        dr[j] = sqrtf(dr[j]);
        for(k = 0; k < 3; k++){
            a[i][k] += m[j] * (r[j][k] - r[i][k]) / (dr[j] * dr[j] * dr[j]);
            a[j][k] += m[i] * (r[i][k] - r[j][k]) / (dr[j] * dr[j] * dr[j]);
        }
    }
}
```

# 実行結果

	実行時間 sec
オリジナル	13.0
演算の共通化	8.4
AoS → SoA	8.1
コンパイルオプション	5.8
<b>ループ分割</b>	<b>5.0</b>

- 解析してないので、なぜ速くなったか分かりません.
  - パイプラインを上手に埋めれるようになった？
  - 上手にベクトル化できるようになった？

## Step5 : 除算の削減

- 和, 差, 積の計算にくらべて, 除算の計算は非常に時間がかかることはご存知ですよね
  - ヘネパタに書いてあるので, 知っているはず
  - 除算や平方根があるコードでは, 和差積はノーコストと考えて良いくらいです.
- なので, ここでは, 除算を減らすことを考えます.

# こんな感じ

```
// main loop
float dr[NMAX];
for(i = 0; i < NMAX-1; i++){
    for(j = i+1; j < NMAX; j++){
        dr[j] = 0;
        for(k = 0; k < 3; k++){
            dr[j] += (r[i][k] - r[j][k]) * (r[i][k] - r[j][k]);
        }
        dr[j] = sqrtf(dr[j]);
    }
    for(j = i+1; j < NMAX; j++){
        for(k = 0; k < 3; k++){
            a[i][k] += m[j] * (r[j][k] - r[i][k]) / (dr[j] * dr[j] * dr[j]);
            a[j][k] += m[i] * (r[i][k] - r[j][k]) / (dr[j] * dr[j] * dr[j]);
        }
    }
}
```



```
// main loop
float dr[NMAX];
for(i = 0; i < NMAX-1; i++){
    for(j = i+1; j < NMAX; j++){
        dr[j] = 0;
        for(k = 0; k < 3; k++){
            dr[j] += (r[i][k] - r[j][k]) * (r[i][k] - r[j][k]);
        }
        dr[j] = sqrtf(1.0/dr[j]);
    }
    for(j = i+1; j < NMAX; j++){
        for(k = 0; k < 3; k++){
            a[i][k] += m[j] * (r[j][k] - r[i][k]) * (dr[j] * dr[j] * dr[j]);
            a[j][k] += m[i] * (r[i][k] - r[j][k]) * (dr[j] * dr[j] * dr[j]);
        }
    }
}
```

# 実行結果

	実行時間 sec
オリジナル	13.0
演算の共通化	8.4
AoS → SoA	8.1
コンパイルオプション	5.8
ループ分割	5.0
除算の削減	4.5

- とりあえず，高速化はここまで！
- 元が13secだったので， $13/4.5=2.8$ 倍高速化することができました！

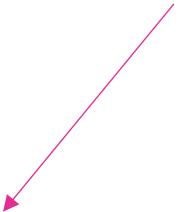
# おまけ：てか，並列化すればよくね？

- このレベルのコードであれば簡単に並列化することができます.
- そのうえ，高速化率も高いです.
- コードの可読性も失われません.
- 今までの高速化より，よっぽどコスパが良いです...



# こんな感じ

この1行を加えるだけ！



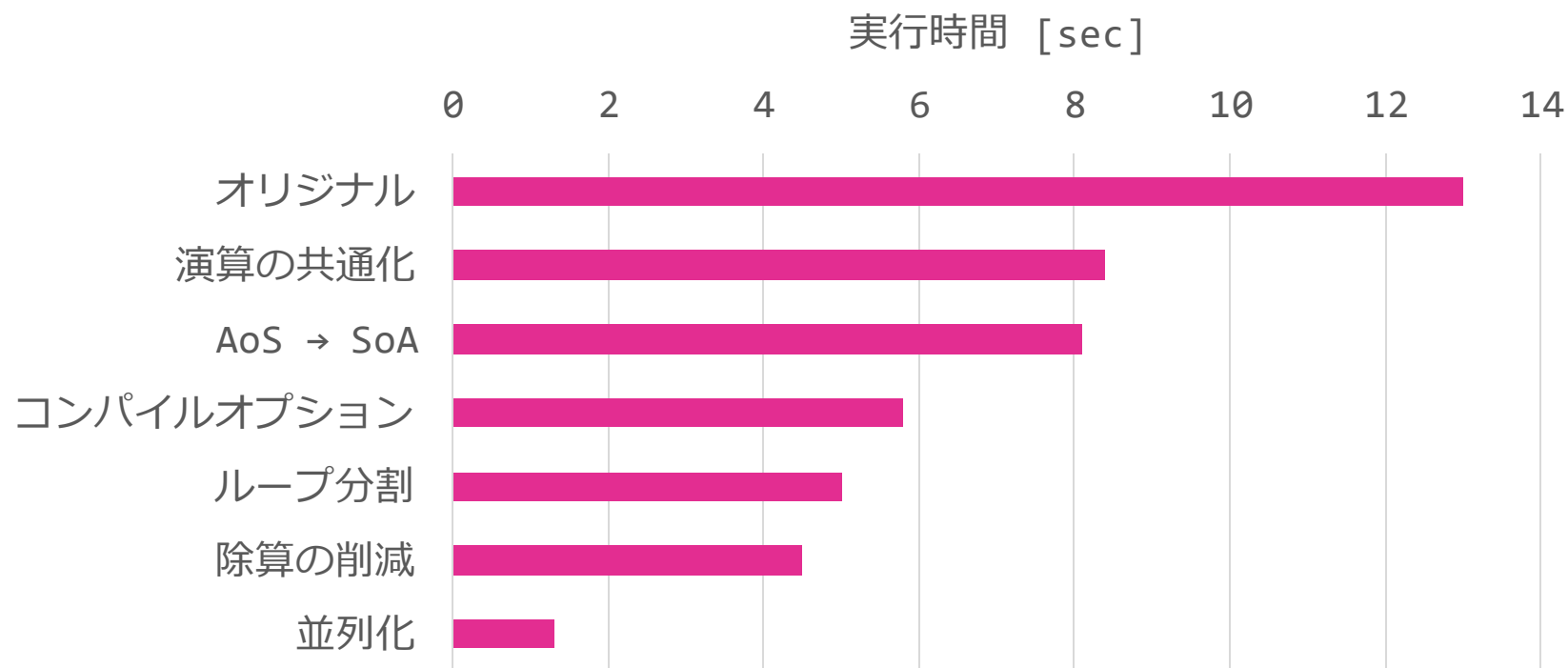
```
// main loop
float dr[NMAX];
#pragma omp parallel for shared(a) private(dr, i, j)
for(i = 0; i < NMAX-1; i++){
    for(j = i+1; j < NMAX; j++){
        dr[j] = 0;
        for(k = 0; k < 3; k++){
            dr[j] += (r[i][k] - r[j][k]) * (r[i][k] - r[j][k]);
        }
        dr[j] = sqrtf(1.0/dr[j]);
    }
    for(j = i+1; j < NMAX; j++){
        for(k = 0; k < 3; k++){
            a[i][k] += m[j] * (r[j][k] - r[i][k]) * (dr[j] * dr[j] * dr[j]);
            a[j][k] += m[i] * (r[i][k] - r[j][k]) * (dr[j] * dr[j] * dr[j]);
        }
    }
}
```

# 実行結果

	実行時間 sec
オリジナル	13.0
演算の共通化	8.4
AoS → SoA	8.1
コンパイルオプション	5.8
ループ分割	5.0
除算の削減	4.5
<b>並列化</b>	<b>1.3</b>

- まあ、速いよね.

# 実行結果まとめ



# 高速化まとめ

- 高速化はやめどきも肝心です.
  - 一般に高速化すればするほど、コードは複雑になります.
  - 埋め込みでアセンブリを書くことさえ可能です.
- 可能であれば、単純に並列数を増やすのが簡単な場合もあります
- もっと高速化を勉強したいという方へ
  - [https://www.slideshare.net/KMC\\_JP/ss-45855264](https://www.slideshare.net/KMC_JP/ss-45855264)

# 最後に

- これにて完結です！
- いかがでしたか？

# え？こんな不便な言語使ってられない？

- わかる
- そんなあなたにはPythonがおすすめです.
- Pythonは無限のライブラリを持っています.
- Pythonを勉強したい！
  - 布川くんの講習を待て

# え？もっといろんなコードを書きたい？

そんなあなたには競技プログラミングがおすすめです！



AtCoder

<https://atcoder.jp/>



<https://codeforces.com/>

興味がある人は研究室Slackの [#atcoder](#) に参加してみよう！

# 本当の本当に終わり

参考：猫でも分かるc言語プログラミング第3版（SB Creative）