

---

# プログラミング基礎演習

## 第5回

### C言語編

#### 【ポイント】

---

長谷川禎彦

---

# 前回課題1

math.hをinclude

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

```
double harmonic(int n) {
    int i;
    double sum = 0;
    for (i = 1; i <= n; i++) {
        sum += 1.0 / i;
    }

    return sum;
}
```

forの開始が1から.  
1/iではなく1.0/i.

sumの初期化を忘れずに

```
int main() {
    double sum = 0;
    int i;
    int upto = 5;
    double log_val;
    double harmonic_val;
    int N;

    for (i = 0; i <= upto; i++) {
        N = (int) pow(10, i);
        log_val = log(N + 1);
        harmonic_val = harmonic(N);
        printf("%f, %f\n", log_val, harmonic_val);
    }

    return 0;
}
```

ここはintではなくdouble

# 前回課題2

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

string.hをinclude

```
int main() {
    char s[] =
    "12934875628476528734692734697263475629101234100";
```

```
    int count[10];
    int len = strlen(s);
    int i, n;
```

配列のサイズは10

```
    for (n = 0; n <= 9; n++) {
        count[n] = 0;
    }
```

```
    for (i = 0; i < len; i++) {
        count[s[i] - '0'] = count[s[i] - '0'] + 1;
    }
```

```
int is_even;
for (n = 0; n <= 9; n++) {
    is_even = count[n] % 2;
    if(is_even == 0) {
        printf("%d is even\n", n);
    } else {
        printf("%d is odd\n", n);
    }
}

return 0;
}
```

=ではなく==

# 前回課題3

```
#include <stdio.h>

int main() {
    double sum = 0.0;
    double v = 0.0;
    while(v != 100.0) {
        sum += v;
        v += 0.1;
    }

    printf("Sum is %f\n", sum);
    return 0;
}
```

0.1は二進数では正確には表現できないので、0.1ずつ足していくとちょうど100にはならない。(修正案は後述)

# 補足: doubleの表現

---



$$value = (-1)^{\text{sign}} \left( 1 + \sum_{i=1}^{52} b_{-i} 2^{-i} \right) \times 2^{(e-1023)}$$

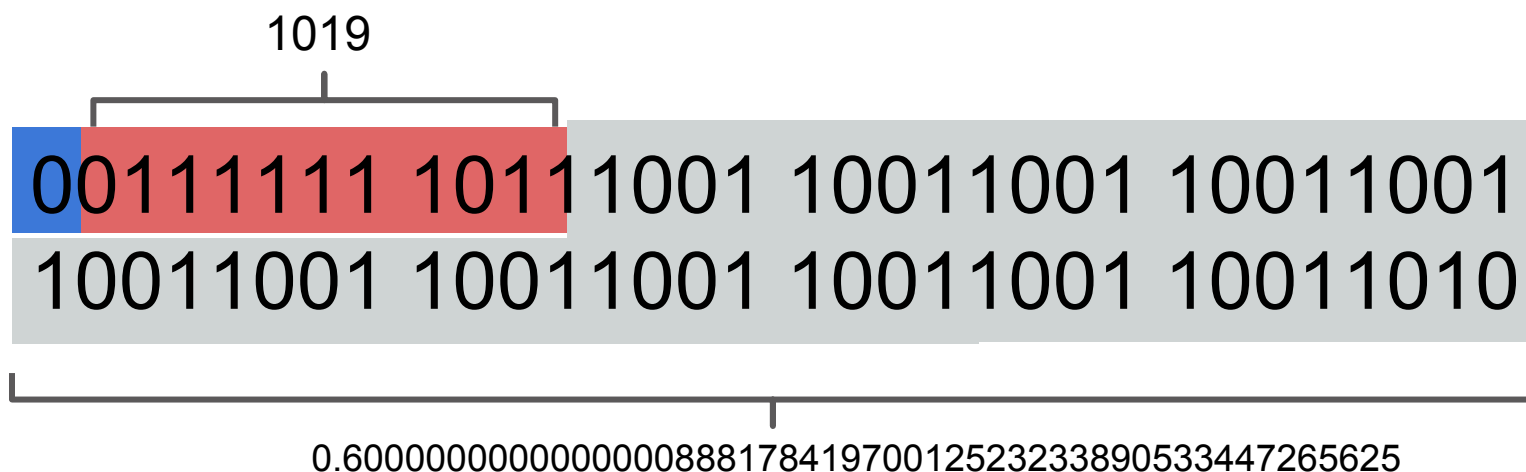
つまり, 10進数では正確に表せる数も, 二進数では正確に表せないため, doubleでは近似値となってしまうことがある

---

## 補足: doubleの表現

---

doubleでの0.1は, バイナリでは循環小数



$$= (-1)^0 \times 1.6000000000000000000000888 \dots \times 2^{-4}$$
$$\simeq 0.1$$

# 補足: doubleの表現

---

- 0.1はdoubleでは近似値である
- 0.1を1000個足し続けても, 100丁度にはならない!
- 0.1の場合に限らず, doubleにおいては等号の条件は用いない  
(悪い例)

```
double x;  
if (x == 100.0) { ... }
```

# 補足: doubleの表現

---

## よく使う方法

```
x == 100.0の代わりに  
fabs(x - 100.0) < e
```

```
x != 100.0の代わりに  
fabs(x - 100.0) >= e
```

(eは0でないとても小さい数)

---



# 第一回レポート課題

---

- ホームページに課題内容のPDFが掲示されている
  - 必須課題一問＋自由課題二問
    - 必須課題提出は必要条件なので、たとえ提出してもクオリティが著しく低い場合は提出とみなさない
-

# ポインタ＝C言語の最重要点

---

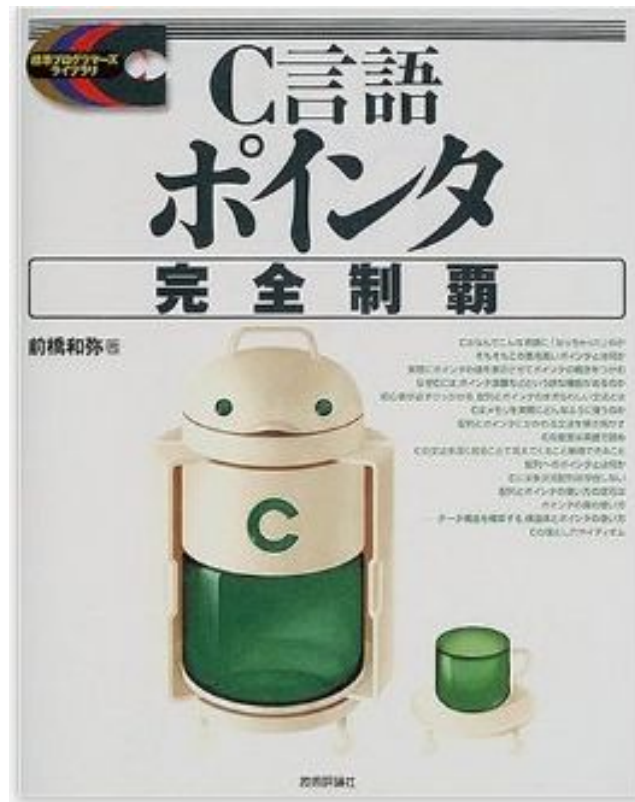
- ポインタの理解

- ポインタの概念はほとんど全ての手続き型言語に存在.  
Java, Ruby, Scalaなどは事実上ポインタしかない.
- ただし, ポインタとは言わずに「参照」という

- ポインタを理解せずして, これ以上の言語に進むことは不可能

# 極めたい人用のお勧めの教科書

---



C言語ポイント完全制覇(単行本)  
前橋和弥(著) 出版社: 技術評論社  
(2001/01)

# ポインタ

---

- コンピュータは, CPU内の記憶領域(レジスタ)と, CPU外の記憶領域(メインメモリ, 主記憶)を用いて計算を行う
    - 主記憶には, プログラムが使うあらゆるデータが格納されている
-

# ポインタとは？

---

- 「ポインタ (pointer)とは, あるオブジェクトがなんらかの論理的な位置情報でアクセスできるとき, それを参照するものである」 [Wikipedia]
  - 全てのデータはメモリ上にある
- ポインタはアドレスを通じて変数や関数にアクセスする方法
  - アドレス＝メモリ上の住所

# ポインタとは？

---

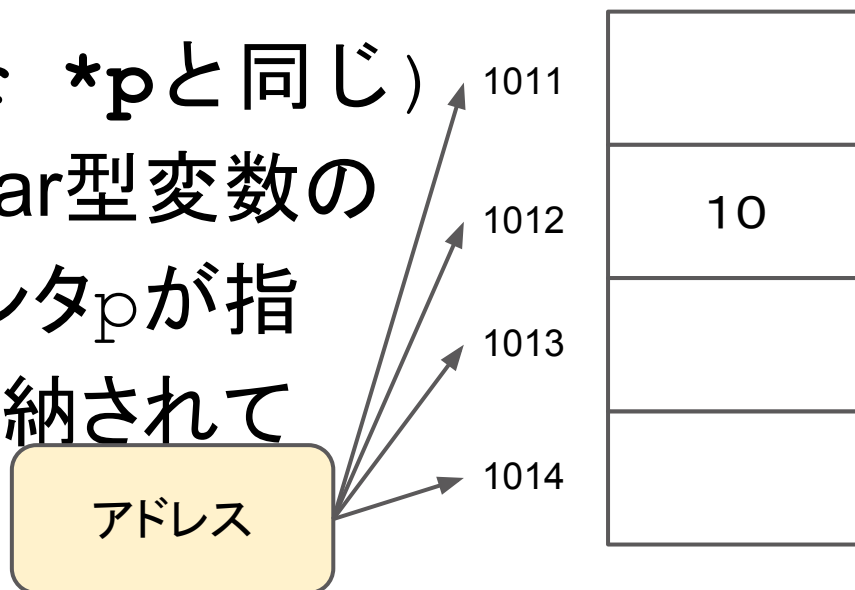
```
char a = 10;
```

とすると、メモリ上のある場所にchar型変数aの領域が確保され、そこに10を格納する。

ポインタは「アドレス」を格納する変数。

```
char* p; (char *pと同じ)
```

だとpに入るのはchar型変数のアドレス。\*pでポインタ<sub>p</sub>が指す先のアドレスに格納されている値を返す。



# ポインタとは？

```
char a = 10;
```

```
char* p;
```

```
p = &a;
```

```
printf("%p\n", p); => 1012
```

```
printf("%d\n", *p); => 10
```

%pはポインタ表示

&aはaのアドレスを返す  
(ここでは1012)

\*pはポインタpが格納して  
るアドレスに格納されて  
る値を返す

アドレス

1011

1012

1013

1014

10

aはここに  
ある

# ポインタと配列

- 配列の変数名(例:“s”)は配列の先頭要素へのポインタ。

- `char s[] = 'tokyo';`
- `s` は先頭要素 (`'t'`) のアドレス
- `*s` (および `s[0]`) は `'t'`
- `s+2` は3番目要素 (`'k'`) のアドレス
- `*(s+2)` (および `s[2]`) は `'k'`
- `(s+2)` と `&s[2]` は同じアドレス

s → 1100  
s+1 → 1101  
s+2 → 1102

アドレス

t
o
k
y
o
\0

- 配列は先頭要素へのポインタ(重要)
- ポインタに「\*」を付ければ, アドレス上に格納されている値にアクセス出来る(読み書き)



# ポインタと配列

---

- 変数のアドレスにアクセスするには「&」を使う

```
printf("%p\n", &s[0]); => 1100  
printf("%p\n", &s[1]); => 1101
```

s → 1100  
s+1 → 1101  
s+2 → 1102

アドレス

t
o
k
y
o
\0

# ポインタと配列

## 例

```
printf("%p\n", s) => 1100
printf("%p\n", s+1) => 1101
printf("%c\n", *s) => 't'
printf("%c\n", *(s+1)) => 'o'
printf("%c\n", s[0]) => 't'
printf("%c\n", s[1]) => 'o'
printf("%p\n", &s[0]); => 1100
printf("%p\n", &s[1]); => 1101
```

s → 1100  
s+1 → 1101  
s+2 → 1102

アドレス

t
o
k
y
o
\0

# ポインタまとめ

---

- ポインタとはアドレスを格納する変数
  - `char* p = ...;` (`char`型変数のアドレスを格納する)
  - `p`に格納されているものはアドレス(整数)
  - `*p` は「`p`に格納されているアドレス」にアクセスする
  - `*(p+5)` は「`p`に格納されているアドレス+5」番地にアクセスする
  - `p[5]` も「`p`に格納されているアドレス+5」番地にアクセスする(シンタックスシュガー)
  - `p[0]` と`*p`は同じもの
  - 変数のアドレスは`&`によって求まる. `p == &p[0]`.
- `char *p` も `char* p` も同じ

# 型とポインタ

---

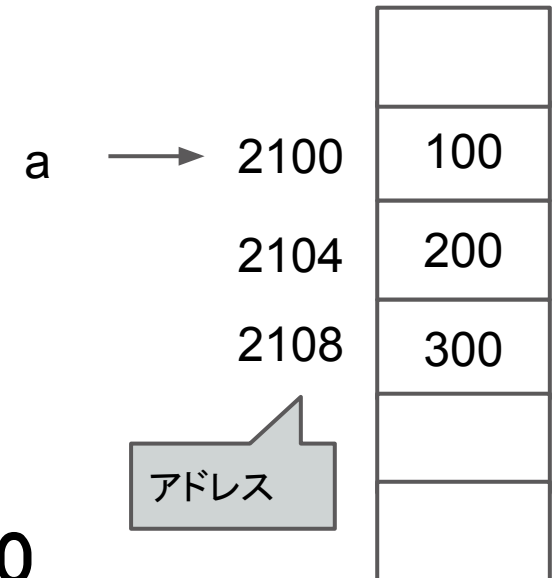
- さきほどの例ではcharなので, 1バイト
- 文字と+1のインクリメントが対応

- intでは？
  - intは4バイトの場合が多い

```
int a[] = {100, 200, 300};
```

```
printf("%p\n", a); => 2100
```

```
printf("%p\n", a+1); => 2101???
```



# 型とポインタ

```
int a[] = {100, 200, 300};
```

```
printf("%p\n", a); => 2100
```

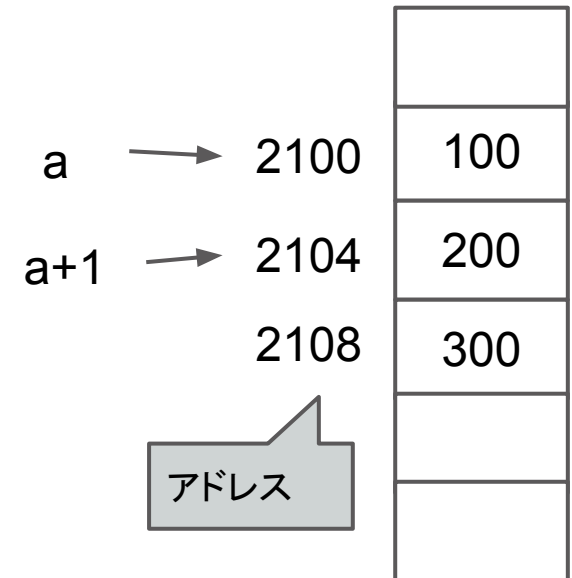
```
printf("%p\n", a+1); => 2101??
```

とはなりません.

アドレス + 1 は, 型一つ分 (重要)

```
printf("%p\n", a+1); => 2104
```

```
printf("%d\n", *(a+1)); => 200
```



つまり, `a[1]`と同じ物が返ってくる

# 次のコードを実行してみよう

---

```
#include <stdio.h>
int main(void){
    int a;
    int* b;
    a = 45;
    b = &a;
    printf("%d\n", *b);
    printf("%p\n", b);
    return 0;
}
```

&aはaのアドレスを返す

# 動的メモリ割り当て

---

- 今まで配列を定義するときにその大きさが既知とした
  - しかし、動的に(プログラム実行時に)しか取るべき配列の大きさが分からないという状況はよくある
  - その場合は「動的に」メモリ割り当てをしなければならない。

```
ポインタ = (型*)malloc(sizeof(型) * 大きさ);
```

# 動的メモリ割り当て

---

```
int *p=(int*)malloc(sizeof(int)*100);
```

mallocで確保したメモリの解放は

```
free(p);
```

で行う. 小さいメモリしか確保しないのならば`free`はなくても問題は起きない(プログラム終了時にOSが回収)が, 再帰的に`malloc`するようなプログラムでは必要となる(例: 木構造など). メモリが解放されずに増えることをメモリリークという.

---



# 動的メモリ割り当ての例

```
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int arraySize = 5;
    int* p;
    p = (int*)malloc(sizeof(int) * arraySize);
    int i;
    for(i = 0; i < arraySize; i++) {
        p[i] = 2 * i;
    }
    for(i = 0; i < arraySize; i++) {
        printf("p[%d] = %d\n", i, p[i]);
    }
    free(p);
    return 0;
}
```

mallocはstdlib.hに定義されている

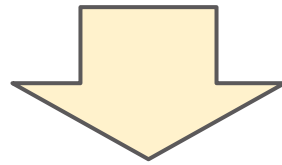
sizeofはデータ型の大きさ（バイトで）返す関数（stdlib.hで定義されている）

ここでメモリを解放

# なぜポインタが必要か？

---

- ポインタの概念はまだ比較的分かりやすい
- しかし, なんのために使うのかが分からない場合がある



- swap関数はポインタなしでは作れない
  - swap関数: `swap(x, y)` に値を入れると, xとyの値を交換する

# 上手いかないソースコード

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

```
void swap(int n1, int n2) {  
    int tmp = n1;  
    n1 = n2;  
    n2 = tmp;  
}
```

この中で値を交換する(はず)

```
int main () {  
    int n1 = 10;  
    int n2 = 1000;  
    printf("n1 = %d, n2 = %d\n", n1, n2);  
    swap(n1, n2);  
    printf("n1 = %d, n2 = %d\n", n1, n2);  
    return 0;  
}
```

この出力は  
n1 = 10, n2 = 1000  
n1 = 10, n2 = 1000  
でスワップされていない...

# 上手いくソースコード

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

```
void swap(int *n1, int *n2) {
```

```
    int tmp = *n1;
```

```
    *n1 = *n2;
```

```
    *n2 = tmp;
```

```
}
```

ここがさっきのコードとは違う

```
int main () {
```

```
    int n1 = 10;
```

```
    int n2 = 1000;
```

```
    printf("n1 = %d, n2 = %d\n", n1, n2);
```

```
    swap(&n1, &n2);
```

```
    printf("n1 = %d, n2 = %d\n", n1, n2);
```

```
    return 0;
```

```
}
```

この出力は

n1 = 10, n2 = 1000

n1 = 1000, n2 = 10

でスワップされる

# なぜ最初のではswapされないか？

---

関数の引数は呼び出し時に渡されたもののコピー  
(全ての言語がそういう訳ではないが、C派生言語  
はみんなあてはまる)

`int swap(n1, n2)` の中で, `n1`, `n2` はコピーさ  
れてから, 関数内ではそのコピーに対して操作さ  
れている (つまり別に `n1`, `n2` のための領域がメモリ  
上に確保され, `swap` から抜けると解放される)

---

# なぜ最初のではswapされないか？

---

```
➡ n1 = 10, n2 = 100;  
   swap(n1, n2);  
   printf("%d, %d\n", n1, n2);
```

1020	
1024	10
1028	100
1032	
1036	

# なぜ最初のではswapされないか？

```
n1 = 10, n2 = 100;
```

```
➡ swap(n1, n2);  
printf("%d, %d\n", n1, n2);
```

main関数内のn1, n2

1020

1024

1028

1032

1036

10

100

呼び出し

1060

1064

1068

1072

1076

10

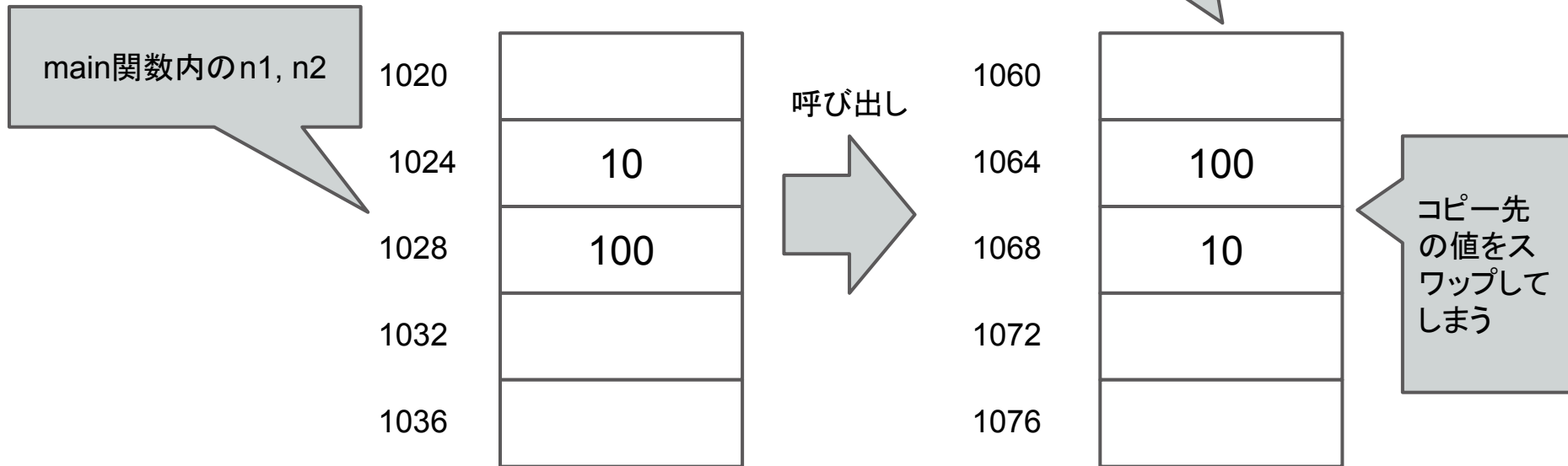
100

swap関数内のn1, n2

# なぜ最初のではswapされないか？

```
n1 = 10, n2 = 100;
```

```
➡ swap(n1, n2);  
printf("%d,%d\n", n1, n2);
```





# なぜ最初のではswapされないか？

```
n1 = 10, n2 = 100;
```

```
swap(n1, n2);
```

```
printf("%d, %d\n", n1, n2);
```

main関数内のn1, n2

1020

1024

1028

1032

1036

10

100

1060

1064

1068

100

10

もともとの値はなんの  
変化もない

# なぜ後ではswapされるか？

---

➡ `n1=10, n2=100;`  
`swap(&n1, &n2);`

main関数内のn1, n2

1020

1024

1028

1032

1036

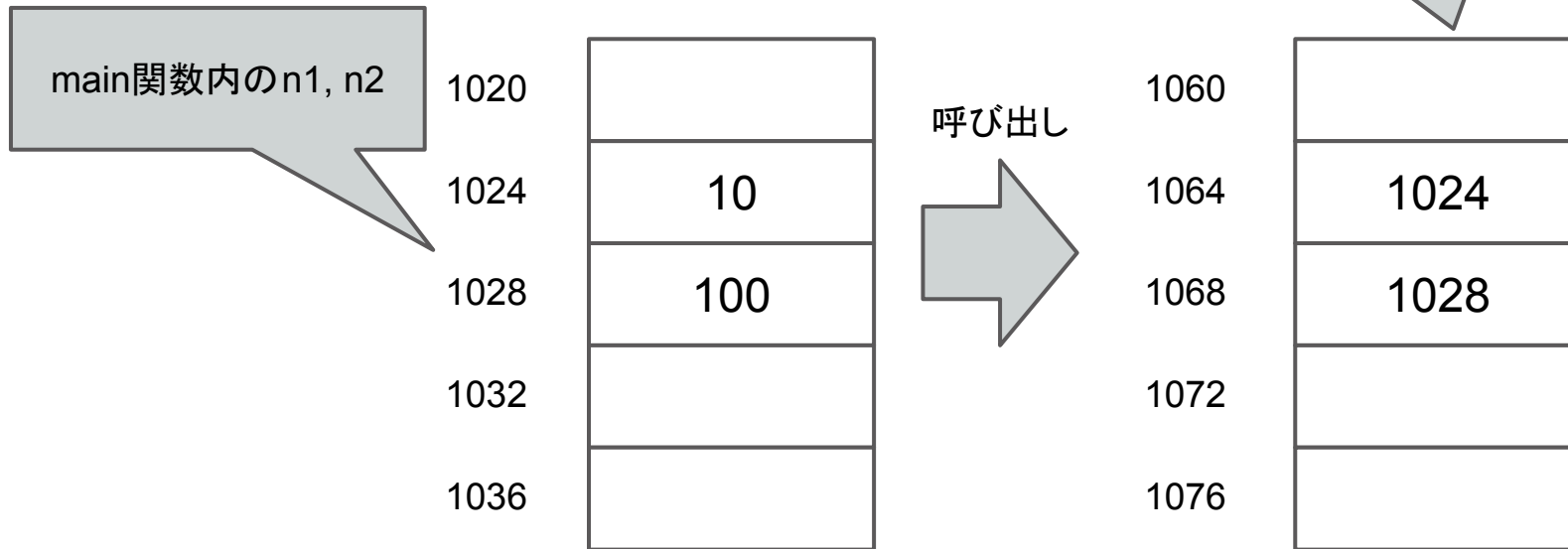
10

100

# なぜ後のではswapされるか？

`n1=10, n2=100;`

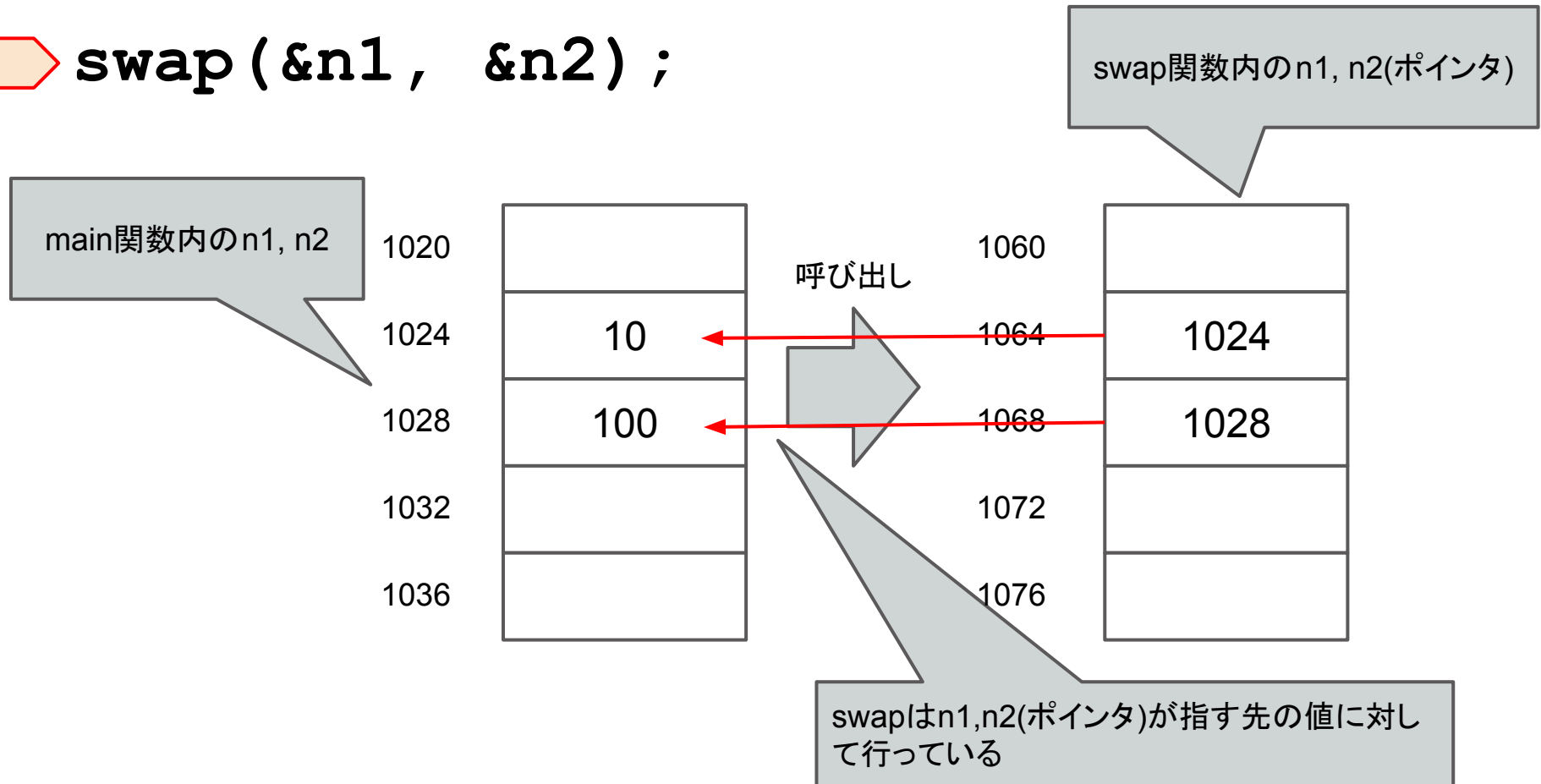
➡ `swap(&n1, &n2);`



# なぜ後ではswapされるか？

`n1=10, n2=100;`

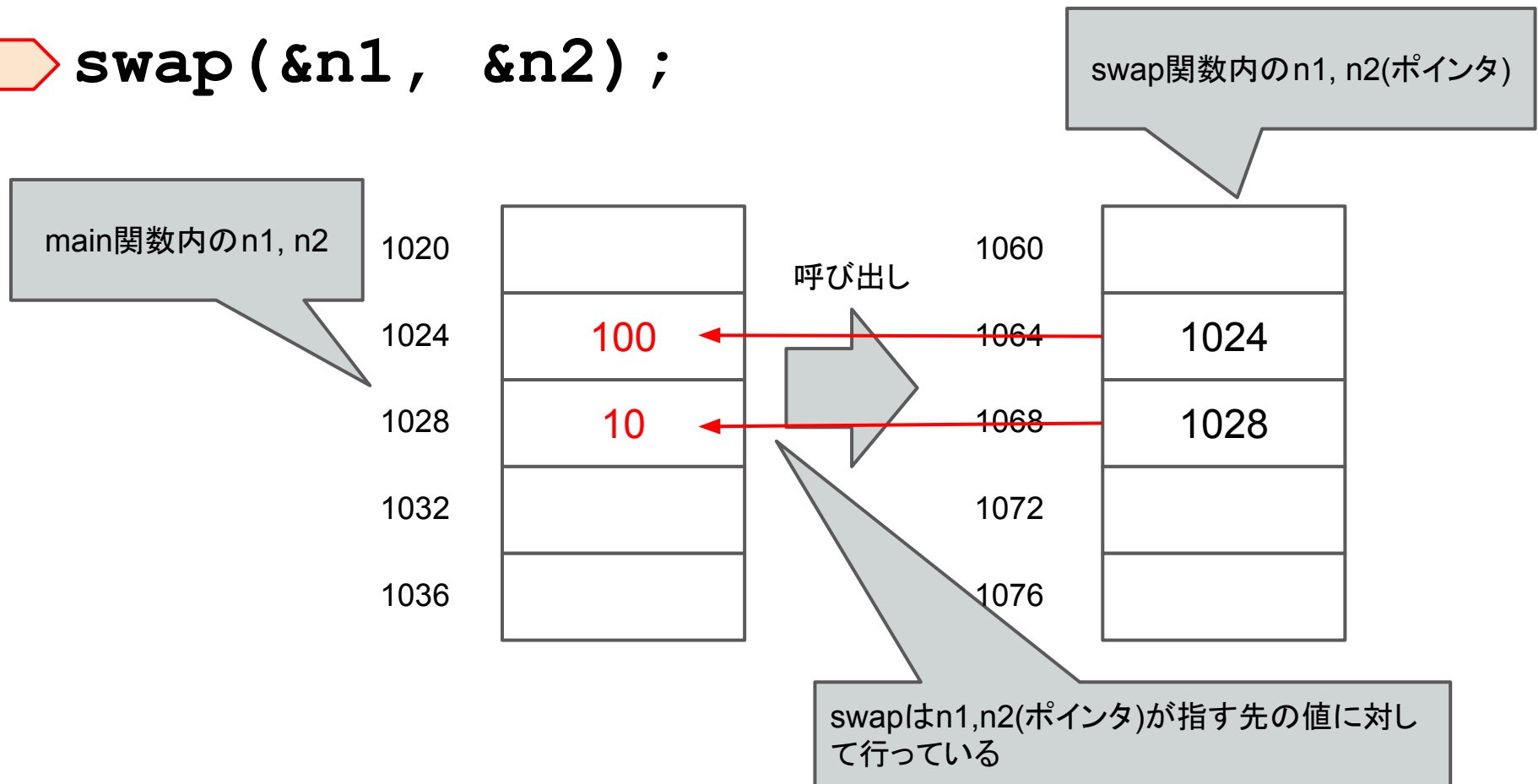
➡ `swap(&n1, &n2);`



# なぜ後ではswapされるか？

`n1=10, n2=100;`

`swap(&n1, &n2);`



# なぜ後ではswapされるか？

`n1=10, n2=100;`

➡ `swap(&n1, &n2);`

main関数内のn1, n2

1020

1024

1028

1032

1036

100

10

関数から戻ってきても、swapされた状態は保持されている

1060

1064

1068

1072

1076

1024

1028

# swapまとめ

---

- ポインタ版でコピーされるのは「アドレス」
  - `swap(int*n1, int*n2)` 関数内ではアドレスの示す値に対して操作される
-

# なぜポインタが必要か2？

---

- 多値を返す関数
  - C言語ではreturnでひとつの値しか返せない
  - 配列をreturnすると1種類しか返せない
    - `int, double, char s[]`を一気に返すには？
- 例：データの平均，最大値，最小値を一度に返す関数statsを作りたい
  - 別々に計算すると計算の無駄



```
#include <stdio.h>

void stats(double v[], int n, double *mean, double *pmax, double *pmin) {

    double sum = 0;
    double max = -1000000;
    double min = 1000000;
    int i;

    for (i = 0; i < n; i++) {
        sum = sum + v[i];
        if (max < v[i]) {
            max = v[i];
        }
        if (min > v[i]) {
            min = v[i];
        }
    }
    *mean = sum / n;
    *pmax = max;
    *pmin = min;
}
```

計算した結果を mean, pmax, pmin のアドレスの  
指す変数に格納している

```
int main() {  
    double v[] = {1.1, -0.4, 3.2, 1.9, 4.0};  
    int size = 5;  
    double mean = 0;  
    double max = 0;  
    double min = 0;  
  
    stats(v, size, &mean, &max, &min);  
  
    printf("mean = %f, max = %f, min = %f\n", mean, max, min);  
    return 0;  
}
```

---

# 補足: 配列のコピー

---

`memcpy(dest, src, size)`

配列srcを配列destにコピーする. sizeはコピーするバイト数  
使うには`#include <string.h>`が必要.

例

```
int a[] = {1,2,3,4,5};  
int* b = (int*)malloc(sizeof(int) * 5);  
memcpy(b, a, sizeof(int) * 5);
```

# 補足: 他の言語では？

---

- Java, Python, Rubyでのオブジェクトは全てポインタ(参照という)
  - そして, これらの言語では実質参照しかない
    - 参照以外のオブジェクトは作れない
    - これらの言語では, Cのポインタの面倒くささや, メモリがらみのバグはほとんど起きない
  - ポインタを知らずにこれらの言語を使うと, とんでもないバグの温床になったり, 理解できない挙動を示すように見える
-

# 今日の課題

---

- 課題1, 2, 3提出で出席扱い
- 課題4, 5は自由課題
- 課題6はおまけ(解説しない予定)

# 課題1

---

数字を一つ増やす関数 `increment` を作成せよ.

```
void increment(int*);  
int main() {  
    int a = 10;  
    increment(&a);  
    printf("%d\n", a); //出力は11  
    return 0;  
}  
  
void increment(int *n) {  
    /*ここに内容を書く*/  
}
```

# 課題1のヒント

---

## ポインタを使わない「失敗」バージョン

```
void increment(int);  
int main() {  
    int a = 10;  
    increment(a);  
    printf("%d\n", a); //出力は10  
    return 0;  
}  
void increment(int n) {  
    n = n + 1;  
}
```

## 課題2

---

- $2 \times 2$ 行列を与えると, 固有値と固有ベクトルを同時に計算する関数を作成せよ(固有値は実数に限る)
  - 複素数になる場合は, 計算しなくて良い
-



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

void eig(double[2][2], double*, double*, double*, double*);

int main() {
    double val1;
    double val2;
    double vec1[2];
    double vec2[2];

    double mat[2][2] = {{1, 5},{6, 2}};
    eig(mat, &val1, &val2, vec1, vec2);
    printf("[%f, %f]\n", mat[0][0], mat[0][1]);
    printf("[%f, %f]\n", mat[1][0], mat[1][1]);
    printf("eigenvalue = %f, eigenvector = (%f, %f)\n", val1, vec1[0], vec1[1]);
    printf("eigenvalue = %f, eigenvector = (%f, %f)\n", val2, vec2[0], vec2[1]);
    return 0;
}

void eig(double mat[2][2], double *val1, double *val2, double *vec1, double *vec2) {
    //ここを埋める
}
```

固有値・固有ベクトルを求め  
たい行列 (2次元配列)

配列名だけで配列の先頭要  
素へのポインタであることを  
思い出す ("&"は必要ない)

## 課題2

---

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$



固有値

固有ベクトル

$$\left\{ \begin{array}{l} \frac{1}{2} \left( -\sqrt{a^2 - 2ad + 4bc + d^2} + a + d \right) \Rightarrow \begin{pmatrix} -\frac{\sqrt{a^2 - 2ad + 4bc + d^2} - a + d}{2c} \\ 1 \end{pmatrix} \\ \frac{1}{2} \left( \sqrt{a^2 - 2ad + 4bc + d^2} + a + d \right) \Rightarrow \begin{pmatrix} -\frac{-\sqrt{a^2 - 2ad + 4bc + d^2} - a + d}{2c} \\ 1 \end{pmatrix} \end{array} \right.$$

## 課題2

---

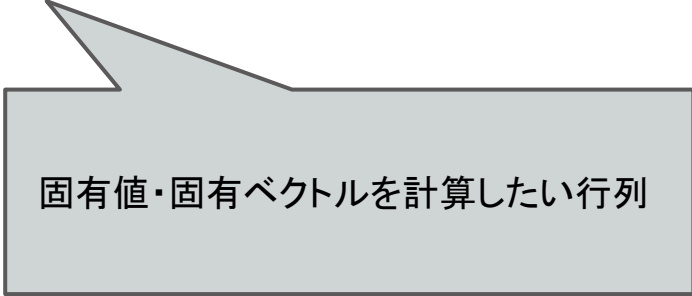
なお, チェックのための固有値・固有ベクトルは

<http://www.wolframalpha.com/>

で

`Eigensystems[{ {1, 5}, {6, 2} }]`

と入れると計算できる.



固有値・固有ベクトルを計算したい行列

# 課題3

---

文字列を逆順にする関数**reverse**を作れ.

```
void reverse(char *s);  
int main() {  
    char s[] = "hello";  
    reverse(s);  
    printf("%s\n", s); // 出力は"olleh"  
    return 0;  
}
```

注:char \*s = "hello"では動かないので注意.  
(文字列リテラルは書き込み禁止領域にメモリ確保され, ポインタsはその領域へのポインタになってしまうため)

# 参考: 文字列に関する関数

---

- **strlen(str)**

- 文字列の長さを返す. ' \0 ' までの長さで, ' \0 ' は含まない( ' \0 ' 以降に何があっても関係なし)

strlen("abc") は 3

- **strcpy(dest, source)**

- 文字列をsourceからdestにコピー. これも ' \0 ' までの範囲( ' \0 ' 含む)をコピーする.

```
char a[20] = "hellohello";
```

```
char b[20] = "world"
```

```
strcpy(a,b)
```

a は "world", b は "world"

これらの関数を使うには string.h  
が必要

## 課題4 (自由課題)

---

入力アルファベットを大文字にする関数  
**uppercase**を書け.

```
void uppercase(char *s);  
int main() {  
    char s[] = "hello world";  
    uppercase(s);  
    printf("%s\n", s); // 出力は"HELLO WORLD"  
    return 0;  
}
```

注:char \*s = "hello world"  
では動かないので注意  
(さっきと同じ理由)

## 課題4 (自由課題)

---

小文字と大文字のcharの値はどのアルファベットでも一定.

例えば以下を実行してみる.

```
printf("%d\n", 'a' - 'A')  
printf("%d\n", 'q' - 'Q')
```

# 課題5（自由課題）

---

ローマ数字(I, II, IV, XIIなど)を数字に変換するプログラムを書け.

```
#include <stdio.h>
int roman2num(char*);
int main() {
    // Examples
    printf("XIV = %d\n", roman2num("XIV")); //14
    printf("CDXCV = %d\n", roman2num("CDXCV")); //495
    printf("MCMXLV = %d\n", roman2num("MCMXLV")); //1945
    printf("MMMCMXCIX = %d\n", roman2num("MMMCMXCIX")); //3999
    return 0;
}
int roman2num(char *s) {
    /*内容を埋める*/
}
```



# ローマ数字 (Wikipediaより)

---

- ★ 「I」が 1、「V」が 5、「X」が 10、「L」が 50、「C」が 100、「D」が 500、「M」が 1000 を意味する。
- ★ 加算すべき数を、できるだけ使う文字数が少なくなるように選び、左から大きい順に並べて書く。例えば 3 は「III」、7 は「VII」、20 は「XX」、23 は「XXIII」となる。
- ★ ただし、同じ文字を4つ以上連続で並べることはできない。そのため、例えば 4 は「IIII」、9は「VIIII」とは表現できない。この場合は小さい数を大きい数の左に書き、右から左を減ずることを意味する。これを減算則という。

[http://www.asahi-net.or.jp/~ax2s-kmttn/ref/roman\\_num.html](http://www.asahi-net.or.jp/~ax2s-kmttn/ref/roman_num.html)

---

# ローマ数字の例 (Wikipediaより)

---

- ❖ XI = 11
  - ❖ XII = 12
  - ❖ XIV = 14
  - ❖ XVIII = 18
  - ❖ XXIV = 24
  - ❖ XLIII = 43
  - ❖ XCIX = 99
  - ❖ CDXCV = 495
  - ❖ MDCCCLXXXVIII = 1888
  - ❖ MCMXLV = 1945
  - ❖ MMMCMXCIX = 3999
-

# 課題提出方法

---

- 課題xの答えをkadai0x.cファイルに書く.
  - xは1,2,3,...
- 締め切りは日曜日の23:59
- ファイルをzipファイルにまとめる
  - ファイル名: 学籍番号.zip
  - 例: 学籍番号が641234の場合, 641234.zipとする.
  - 圧縮方法はホームページに記載してある
- 学籍番号.zipファイルのみを提出する
  - 提出は <https://goo.gl/QJ3LMP>