プログラミング2　第5回

EX-4.1について

```
int npc_rec(char ox, int turn, int i, int j){

  table[i][j] = ox;
  if(judge(ox)){
    table[i][j] = 0;
    return 1;
  }
  if(turn == 8){
    table[i][j] = 0;
    return 0;
  }
  char ox2 = (ox == 'O') ? 'X':'O';  //相手にターンを渡すので、oxをひっくり返す
  int ii,jj;
  for(ii = 0;ii < 3;ii++){
    for(jj = 0;jj < 3;jj++){
      if(table[ii][jj] == 0){
        int w_l = npc_rec(ox2,turn+1,ii,jj);
                  /* 相手の勝ち負け情報を集約して、自分が勝ちなのか負けなのか引き分けなのか判定する */
      }
    }
  }

  int rval = 0;
  /* rvalに勝ちなら1, 引き分けなら0, 負けなら-1を代入 */
  table[i][j] = 0;
  return rval;
}
```

//勝ちなら1, 負けなら-1, 分けなら0を返す関数と考える

講義計画（暫定）

- ~~第1回(11/28(金))　ガイダンス、Cプログラミングの基礎（データ型、変数、四則演算、制御構文）~~
- ~~第2回(12/05(金))　ビット演算、配列、関数~~
- ~~第3回(12/12(金))　配列と関数、文字列、ソート~~
- ~~第4回(12/19(金))　再帰関数、ポインタ(1)~~
- 第5回(12/26(金))　ポインタ(2)
- 第6回(01/ 9(金))　構造体・共用体、リスト
- 第7回(01/23(金))　Cプログラミング実用上必須のもの（defineマクロ、ファイル操作、複数ソースコンパイルなど）
- 第8回(01/30(金))　期末試験（**出席必須！**）

復習：メモリアドレス

- プログラム実行時、データはメモリに格納される
- メモリは位置情報（アドレス）によって管理される
- データへのアクセス方法
  1. 変数を使ってデータに直接アクセスする
  2. アドレス（ポインタ）を使って間接的にアクセスする

アドレス	データ
…	…
1000	8752
1004	-12898
1008	48
…	
100004	65536
100008	1008

復習：メモリと変数

- 変数は、データが格納される場所に名前をつけたもの
- コンパイラは「プログラム中の変数aはメモリの1008番地」などと対応づけて変数を扱う

```
int a = 48;
```

メモリの1008番地を48に書き換えよという意味になる

	アドレス	データ
	…	…
	1000	0
	1004	0
a	1008	48
	…	
	100004	0
	100008	0

int型のデータ

復習：ポインタ

- ポインタはアドレスを保存するための変数

```
int a = 48;
int *p;
p = &a;
```

ポインタ変数の宣言。コンパイラが「変数pはメモリの100008番地」と対応づけたとする。

メモリの100008番地をaのアドレス（1008)に書き換えよという意味になる。(&はアドレス演算子)

	アドレス	データ	
	…	…	
	1000	0	
	1004	0	
a	**1008**	48	int型のデータ
	…		
	100004	0	
p	100008	**1008**	int*型のデータ

復習：ポインタ

- ポインタ変数に保存されたアドレス経由で間接的にデータにアクセス

```
int a = 48;
int *p;
p = &a;
*p = 9999;
```

間接演算子*を使えば、pに保存された1008番地のデータを9999に書き換えよという意味になる。

	アドレス	データ	
	…	…	
	1000	0	
	1004	0	
a	**1008**	**9999**	int型のデータ
	…		
	100004	0	
p	100008	**1008**	int*型のデータ

pが指すアドレスのデータが変わる

pの値は変わらない

固定長配列によるメモリ確保

- 配列を宣言するときはプログラム中にサイズを明記する

```
int array[1000];
```

  - 配列はプログラム実行時に（スタック）メモリ上に確保される
- 固定長配列の問題点
  - コンパイル時にサイズ（配列長）を決めねばならない
    - 実行時にならないとサイズが決まらない場合は、大きめに確保する必要があり、無駄が生じる
  - 大きなサイズの配列を宣言すると（スタック領域の）メモリが不足する
    - 関数内で巨大な配列を宣言すると実行時にエラーするのはこのため
    - グローバル配列にすればより大きな配列が使える

可変サイズの動的メモリ確保：malloc()

- 必要なときに必要な分だけメモリを確保する方法

```
void *malloc(size_t numbytes);
```

  - 引数は割り当てたいメモリのバイト数
  - 戻り値は割り当てたメモリブロックの先頭を指すポインタ
    - void*は、型がまだ決まっていないアドレスと思えば良い
  - メモリ割り当てに失敗した場合はNULLが返る
  - ヘッダーファイルstdlib.hのincludeが必要
- 例
  - char型（1バイト）100個分、100バイトをメモリを確保

```
char *p;
p = malloc(100);
```

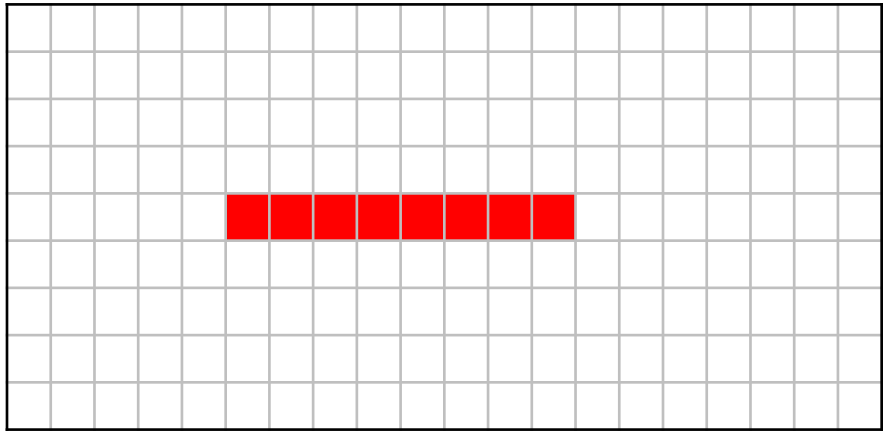
注：
char *p = **(char *)**malloc(100);
のように、代入時にキャストを行なっている資料がweb上に沢山あるが、C言語の仕様上は自動でキャストされるため不要。
C++でmallocを使う場合は必須。

malloc()による動的なメモリの割り当てのイメージ

malloc(8)

メモリ上のどこかに8バイト分を確保

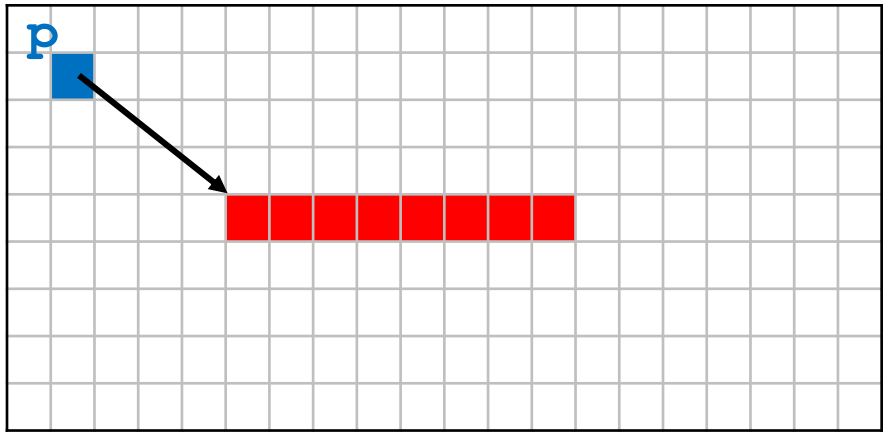
- このままだと、どこにメモリ割り当てが行われたのかわからない



メモリ空間のイメージ

malloc()の一般的な使い方

```
char *p;
p = malloc(8);
...
for(i=0; i<8; i++){
  p[i] = i;
}
```



メモリ空間のイメージ

- malloc()の戻り値＝確保したメモリの位置（先頭のアドレス）をポインタ変数に代入して使う
- 割り当て後のメモリは配列と同様に使える

malloc()で指定するバイト数について

- malloc()には割り当てたいメモリのバイト数を指定する
  - 一般的には基本データ型のサイズはsizeof演算子で指定する
    - 文字列100文字分　→　sizeof(char)*100　　＝100バイト
    - int型配列で要素数100　→　sizeof(int)*100　　＝400バイト
    - double型配列で要素数100　→　sizeof(double)*100　　＝800バイト
  - 具体的な書き方の例：double型でn個のメモリを割り当てる

```
double *dary;
dary = malloc(n*sizeof(double));
```

動的に確保したメモリの解放：free()

- 静的割り当てでは、プログラマがメモリ割り当てと解放のタイミングを気にする必要はない（制御できない）
- 動的割り当てでは、プログラマが明示的に解放する必要がある
  - ただし、プログラム終了時にはすべて自動で解放される
- メモリ解放を行う関数の一般的な形式

```
void free(void *ptr);
```

  - 引数は、解放したいメモリブロックの先頭を指すポインタ
  - （malloc同様、）ヘッダーファイルstdlib.hのincludeが必要
- 例

```
char *p;
p = malloc(100);
...
free(p);
```

- 解放時にはvoid*型へのキャストやサイズは考えなくてよい
  （解放したいアドレス＝メモリブロックの先頭を指すだけでOK）

メモリリークとガベージコレクション

- C言語ではメモリ割り当て・解放といった操作をプログラマが明示的に行う
  - 利点：限られたメモリ資源を有効活用できる
  - 欠点：メモリを解放せずにいると（解放し忘れたままにすると）使える資源が不足する、多重解放などのミスをするとプログラムがエラー終了してしまう（メモリリーク）
- ガベージコレクション（Garbage Collection, GC）
  - Python, Ruby, Javaなどは未使用のメモリを勝手に解放してくれる
    - そのメモリを参照しているオブジェクトがなくなった時点で勝手に解放される
  - 利点：メモリ管理が不要
  - 欠点：GC処理の分だけプログラムの動作が遅くなる
    - 解放のタイミングによっては実行時間への影響が大きくなる

多重確保・多重解放に注意

- 同じポインタ変数に複数回malloc()したりfree()すると問題が起こりうる

- 多重確保
  - 行ってもすぐには問題が起きず、そのまま実行できることも多い
  - 最初に確保されたメモリ領域にアクセスする手段を失う
  - 何度も繰り返すとその分だけメモリを消費、最後にはメモリが枯渇し確保不可能になる

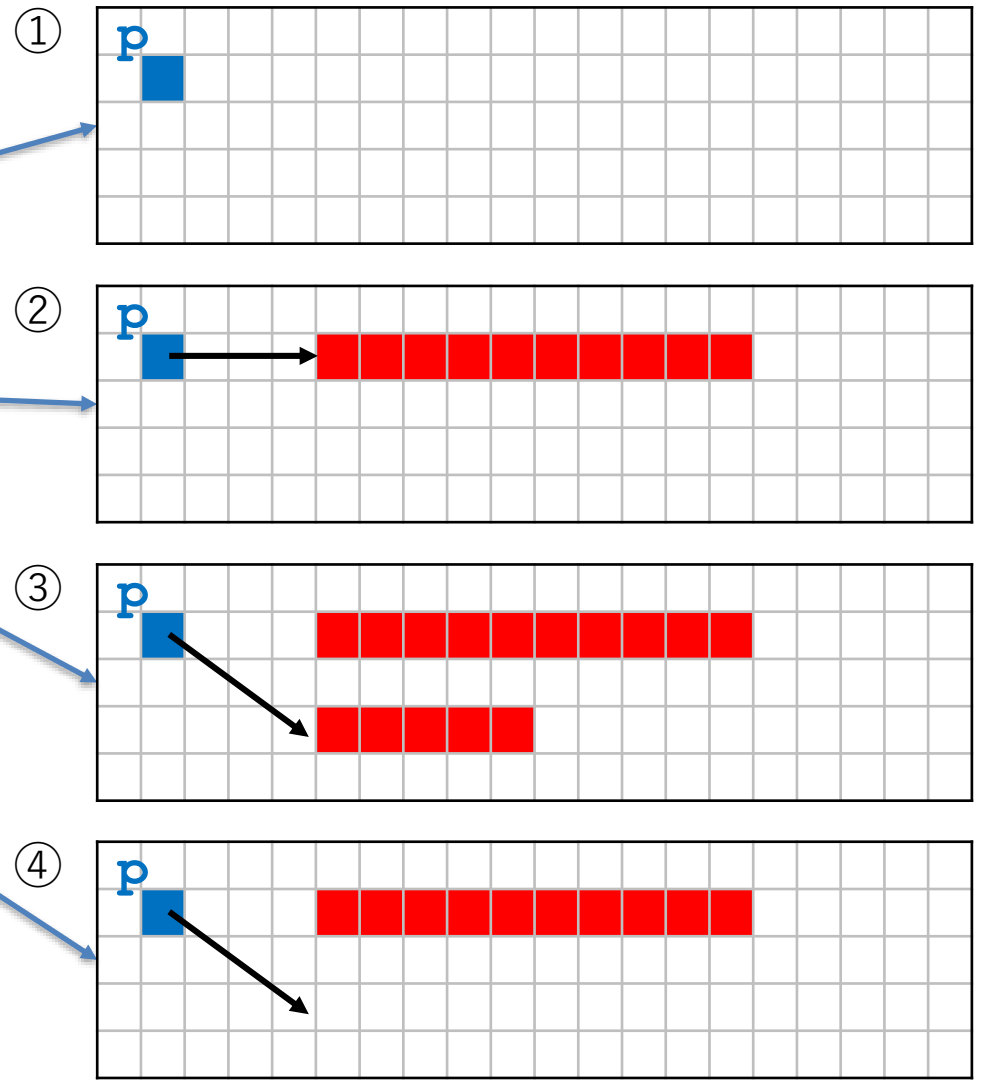
- 多重解放
  - 多重解放は未定義動作。エラーする場合も、しない場合も。

多重確保の例

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

int main(int argc, char *argv[])
{
  double *p;              ①
  printf("p %p¥n", p);
  p = malloc(10*sizeof(double)); ②
  printf("p %p¥n", p);
  p = malloc(5*sizeof(double));  ③
  printf("p %p¥n", p);
  free(p);                ④
  printf("p %p¥n", p);
  return 0;
}
```

メモリ空間のイメージ



多重mallocするとpに保存されたアドレスが変わるがエラーはしない

freeしてもpに保存されたアドレスは元に戻らず、最初に確保した領域はアクセス不能になる

多重確保の防ぎ方の例

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

int main(int argc, char *argv[])
{
  double *p;
  printf("p %p¥n", p);
  p = malloc(10*sizeof(double));
  printf("p %p¥n", p);
  p = malloc(5*sizeof(double));
  printf("p %p¥n", p);
  free(p);
  printf("p %p¥n", p);
  return 0;
}
```



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

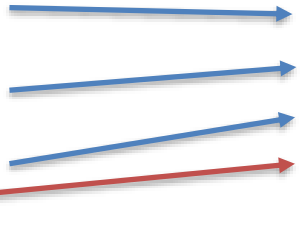
int main(int argc, char *argv[])
{
  double *p = NULL;
  printf("p %p¥n", p);
  if(p==NULL) p = malloc(10*sizeof(double));
  printf("p %p¥n", p);
  if(p==NULL) p = malloc(5*sizeof(double));
  printf("p %p¥n", p);
  free(p);
  printf("p %p¥n", p);
  return 0;
}
```

- 初期値としてNULLを入れておき、malloc済（NULLではなくなっている）かどうかを判定してからmallocすれば多重確保を防ぐことができる
- 実務上は推奨だが、この講義の課題では左側の書き方でも減点しない

多重解放の例

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

int main(int argc, char *argv[])
{
  double *p = NULL;
  printf("p %p¥n", p);
  p = malloc(100*sizeof(double));
  printf("p %p¥n", p);
  free(p);
  printf("p %p¥n", p);
  free(p);
  printf("p %p¥n", p);
  return 0;
}
```



```
$ ./a.out
p 0x0
p 0x800058570
p 0x800058570
Aborted (core dumped)
```

- printf()で%pを使うと
  16進数でアドレスを出力できる

多重解放すると即時エラー

多重解放の防ぎ方の例

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

int main(int argc, char *argv[])
{
  double *p = NULL;
  printf("p %p¥n", p);
  p = malloc(100*sizeof(double));
  printf("p %p¥n", p);
  free(p);
  printf("p %p¥n", p);
  free(p);
  printf("p %p¥n", p);
  return 0;
}
```



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

int main(int argc, char *argv[])
{
  double *p = NULL;
  printf("p %p¥n", p);
  p = malloc(100*sizeof(double));
  printf("p %p¥n", p);
  if(p!=NULL) { free(p); p=NULL; }
  printf("p %p¥n", p);
  if(p!=NULL) { free(p); p=NULL; }
  printf("p %p¥n", p);
  return 0;
}
```

- free時にNULL判定　＋　freeしたらNULLを代入
  により多重解放を防ぐことができる
- さらにメモリ確保時にNULL判定をすることでプログラムの安全性を高めることができる
- 実務上は推奨だが、この講義の課題では左側の書き方でも減点しない

Static変数

- 関数内で作られたローカル変数は、関数終了時になくなってしまう
  - 関数の引数も、関数内のみのローカル変数として扱われる
- 次回の関数実行時にも残っていて欲しい場合は、グローバル変数やstatic変数を使ったりする必要がある

- static変数
  - 通常のローカル変数は存在する関数の実行が終わると破棄されてしまうが、static変数は値が残る
  - 寿命はグローバル変数と同じだが、スコープはローカル変数のまま

こう書けば
初期化時にのみ0が代入される

Static変数の使用例

```
int sub(int count)
{
  static int val=0;
  val++;
  printf("val = %d¥n", val);
  return 0;
}

int main(int argc, char *argv[])
{
  sub(0);
  sub(1);
  return 0;
}
```

関数の引数に対してmallocする時

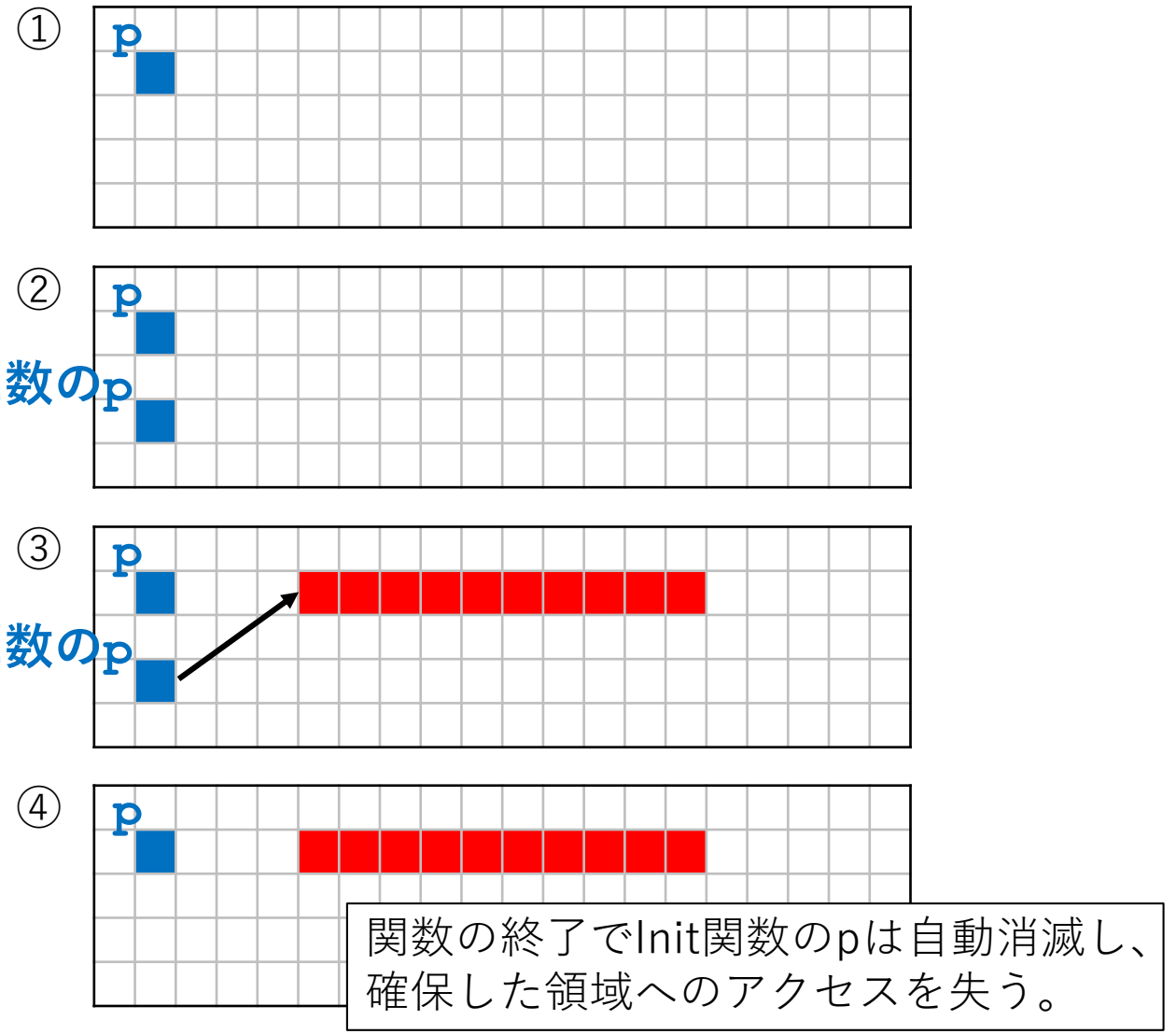
- 次のコードは期待通り動かない

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

void init(int n, int *p) {
③  p = malloc(sizeof(int)*n);
  int i;
  for(i=0; i<n; i++){
    p[i] = i;
  }
④ }

int main(int argc, char *argv[]) {
  int i;
①  int *p = NULL;
②  init(10, p);
  for(i=0; i<10; i++){
    printf("%d %d\n", i, p[i]);
  }
  free(p);
  return 0;
}
```

メモリ空間のイメージ



関数の引数に対してmallocする時

- ポインタのポインタ(後述)にする

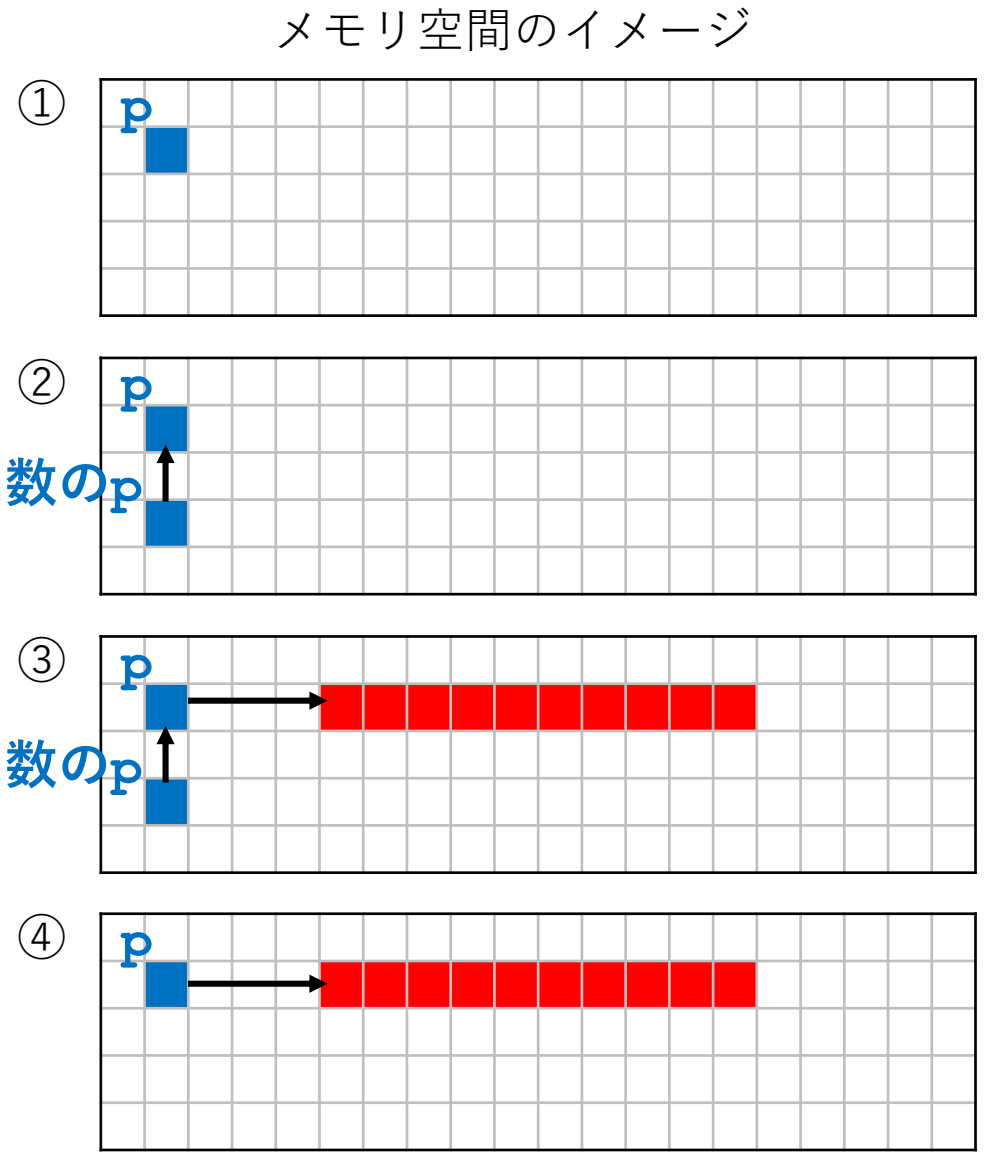
```
#include <stdio.h>
#include <stdlib.h>

void init(int n, int **p) {
③  *p = malloc(sizeof(int)*n);
   int i;
   for(i=0; i<n; i++){
     (*p)[i] = i;
   }
④ }

int main(int argc, char *argv[]) {
   int i;
①  int *p = NULL;
②  init(10, &p);
   for(i=0; i<10; i++){
     printf("%d %d\n", i, p[i]);
   }
   free(p);
   return 0;
}
```

*p[i]ではないことに注意

メモリ空間のイメージ



関数の引数に対してmallocする時

- アドレスを戻り値にするのもあり

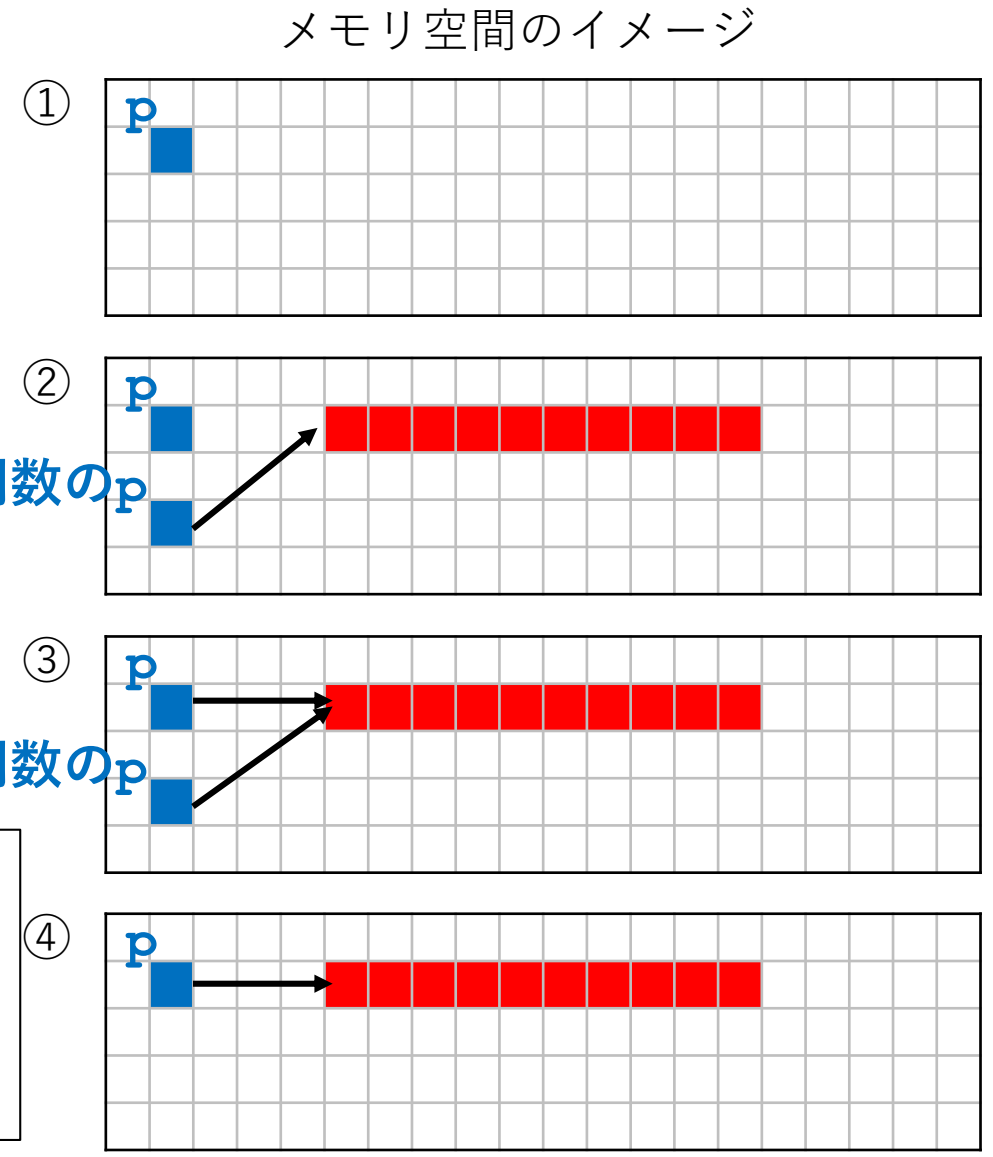
```
#include <stdio.h>
#include <stdlib.h>

int* init(int n) {
②  int *p = malloc(sizeof(int)*n);
   int i;
   for(i=0; i<n; i++){
     p[i] = i;
   }
③  return p;
④ }

int main(int argc, char *argv[]) {
   int i;
①  int *p = NULL;
   p = init(10);
   for(i=0; i<10; i++){
     printf("%d %d\n", i, p[i]);
   }
   free(p);
   return 0;
}
```

Init関数のpに保存されたアドレスをmain関数のpに代入している

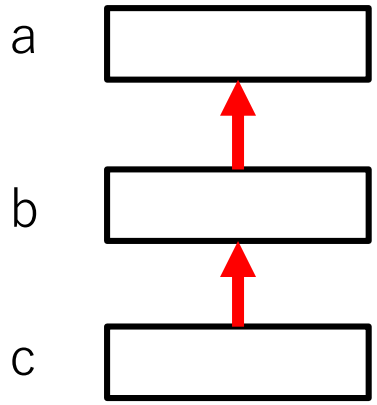
メモリ空間のイメージ



多重間接参照

- ポインタのポインタ
- 例: 2重間接参照

```
int a, *b, **c;
b = &a;
c = &b;
```



  - 3重以上も使える

- 用途
  - リンク構造の際に有用
  - 次回講義の構造体で必要性がでてくる

多重間接参照

- 間接演算子を付けて参照先の情報を出力すれば、「参照先」の値やアドレスが見える

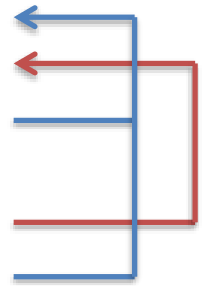
```c
#include <stdio.h>

int main(void)
{
  int a, *b, **c;
  a = 2023;
  b = &a;
  c = &b;

  printf("a    = %8d %p¥n", a, &a);
  printf("b    = %8d %p¥n", b, &b);
  printf("*b   = %8d %p¥n", *b, &*b);
  printf("c    = %8d %p¥n", c, &c);
  printf("*c   = %8d %p¥n", *c, &*c);
  printf("**c  = %8d %p¥n", **c, &**c);

  return 0;
}
```

```
$ ./a.out
a    =    2023 0xffffcc2c
b    =  -13268 0xffffcc20
*b   =    2023 0xffffcc2c
c    =  -13280 0xffffcc18
*c   =  -13268 0xffffcc20
**c  =    2023 0xffffcc2c
```



ポインタ配列

- ポインタも他のデータ型と同様に配列にすることができる
- 例: 20個の要素を持つint型ポインタ変数の配列

```
int *pa[20];
```

  - 9番目の要素にint型変数xのアドレスを代入する場合

```
pa[8] = &x;
```

  - ポインタ配列に代入できるのはアドレスだけであり、int型の値などは代入できない

```
pa[2] = 100;  /* これは間違い */
```

```
int x;
pa[2] = &x;
*pa[2] = 100;  /* これは正しい */
```

ちなみに
int *pa[20];　と書こうが
int* pa[20];　と書こうが
int * pa[20];　と書こうが
差はない。

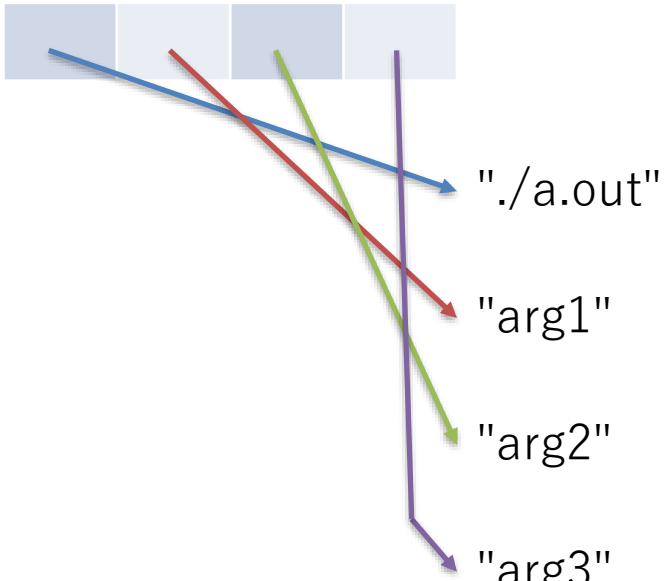
main関数の第二引数もポインタ配列

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
  int i;
  for(i=0; i<argc; i++){
    printf("[%d] %s¥n", i, argv[i]);
  }
  return 0;
}
```

ポインタ配列

```
$ ./a.out arg1 arg2 arg3
```




```c
int main(int argc, char *argv[])
```
と書いても、
```c
int main(int argc, char **argv)
```
と書いても、どちらでも良い

多次元配列の動的なメモリ割り当て

- M行N列2次元配列の静的なメモリ割り当ての例

```
double x[M][N];
```

- ポインタ配列に1次元配列のアドレスを代入して2次元配列として使う
  - 1次元配列の先頭を指すポインタをまとめたポインタ配列

```
double *px[M];
for(i=0; i<M; i++) px[i] = malloc(N*sizeof(double));
```

- ポインタ配列も動的にメモリ割り当て
  - アスタリスク2つを付けて宣言するポインタのポインタを使う

```
double **px;
px = malloc(M*sizeof(double *));
for(i=0; i<M; i++) px[i] = malloc(N*sizeof(double));
```

動的に割り当てた多次元配列の解放

- malloc()した回数分だけメモリ解放が必要

```
double **px;
px = malloc(M*sizeof(double *));
for(i=0; i<M; i++){
  px[i] = malloc(N*sizeof(double));
}
```



```
for(i=0; i<M; i++){
  free(px[i]);
}
free(px);
```

pxを先に解放しないように注意
（pxを先に解放するとpx[i]の場所が分からなくなってしまい解放できない）

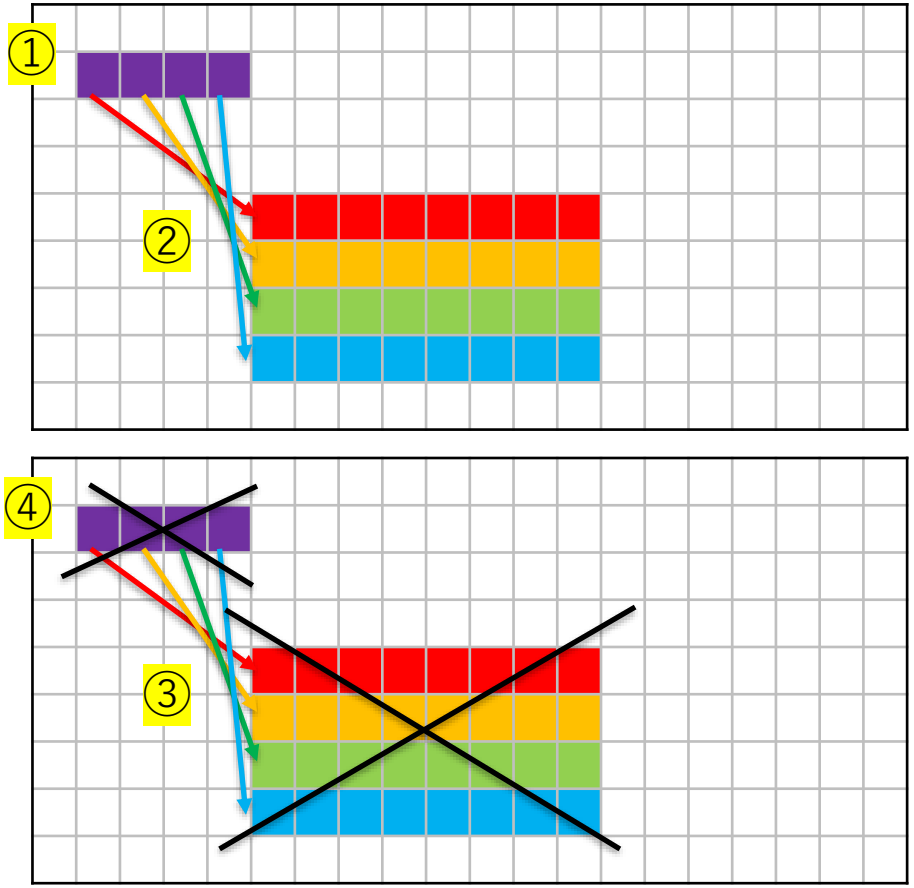
二次元配列の動的な割り当て・解放のイメージ

割り当て

```
      double **px;
①     px = malloc(M*sizeof(double *));
②     for(i=0; i<M; i++){
        px[i] = malloc(N*sizeof(double));
      }
```

解放

```
③     for(i=0; i<M; i++){
        free(px[i]);
      }
④     free(px);
```



確保されたアドレスの確認

- 2段階のmallocで確保したアドレスを表示して確認する

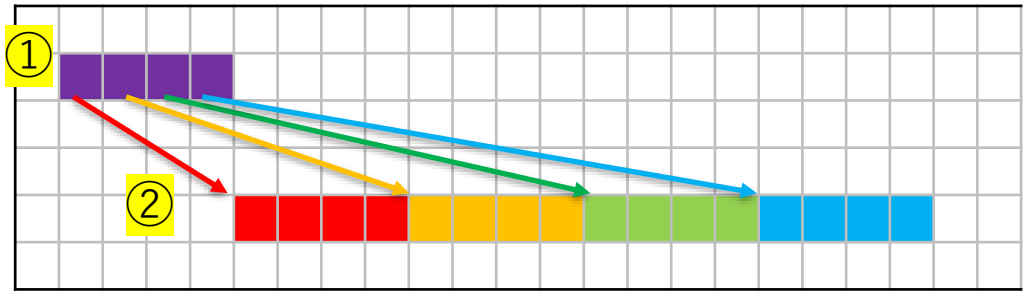
```
#define M 4
#define N 3

int main(int argc, char *argv[])
{
  int i;
  double **px;
  px = malloc(M*sizeof(double *));
  printf("px    %p¥n", px);
  for(i=0; i<M; i++){
    px[i] = malloc(N*sizeof(double));
    printf("px[%d] %p¥n", i, px[i]);
  }
  return 0;
}
```

```
$ ./a.out
px    0x8000003e0
px[0] 0x800000460
px[1] 0x800000480
px[2] 0x8000004a0
px[3] 0x8000004c0
```

ちょうど8バイト×4＝32バイト差

この実行例では連続領域に確保されたが、別々のmalloc()で確保した領域が連続することは一切仮定できない。



静的に割り当てられた2次元配列を関数へ渡す方法

- 静的確保の場合：可変長配列（C99の機能）を使った仮引数の指定方法の例

```
void matvec(int m, int n, double a[][n], double x[], double y[])
{
  ...
}

int main(void)
{
  double a[M][N], x[N], y[M];
  matvec(M, N, a, x, y);
  ...
}
```

2次元配列の形状が分かるように
受け取る必要がある

動的に割り当てられた2次元配列を関数へ渡す方法

- 動的確保の場合

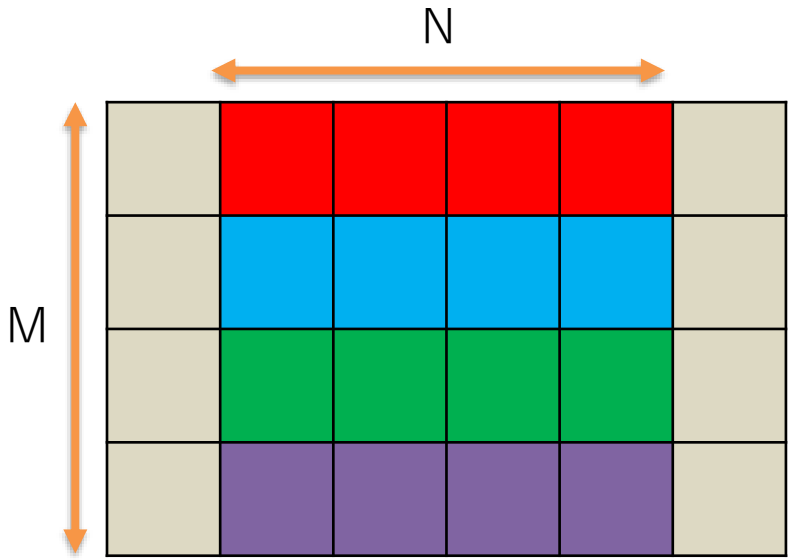
```
void matvec(int m, int n, double **a, double *x, double *y)
{
  ...
}


int main(void)
{
  double **a, *x, *y;
  a = malloc(M*sizeof(double *));
  for (i=0; i<M; i++) a[i] = (double *) malloc(N*sizeof(double));
  x = malloc(N*sizeof(double));
  y = malloc(M*sizeof(double));
  matvec(M, N, a, x, y);
  ...
}
```

- アスタリスク2つのポインタ変数で受け取ることができる
- 受け取った側で形状がわかるようにサイズ情報も渡すのが一般的（サイズ情報も渡さないと使いようがない）

一次元配列を二次元配列として利用する

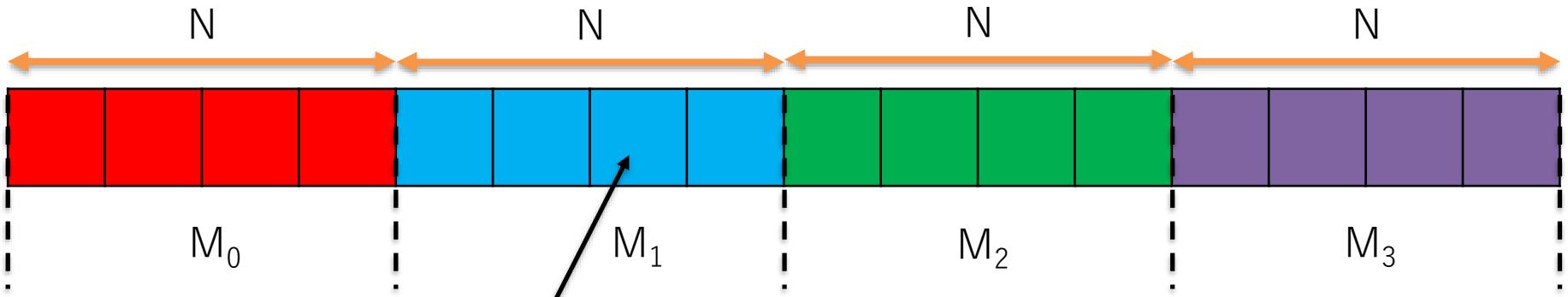
- 行列をmalloc()を用いてメモリに動的に割り当てる場合、ポインタのポインタを用いるとプログラミングの手間が多い。しかも行同士のメモリが連続するとは限らない。（計算内容によっては性能的に損。）



メモリは連続にアクセスした方が性能が出るが、各行が隙間なくつながっている保証がない。

一次元配列を二次元配列として利用する

- 行列をmalloc()を用いてメモリに動的に割り当てる場合、ポインタのポインタを用いるとプログラミングの手間が多い。しかも行同士のメモリが連続するとは限らない。（計算内容によっては性能的に損。）
- 一次元配列を確保して二次元配列として使う
  - 対応するインデックスを計算すれば良いという発想



配列の対応の例：
二次元配列上のa[1][2]と
一次元配列上のa[6]が対応（a[1*N+2]と考える）

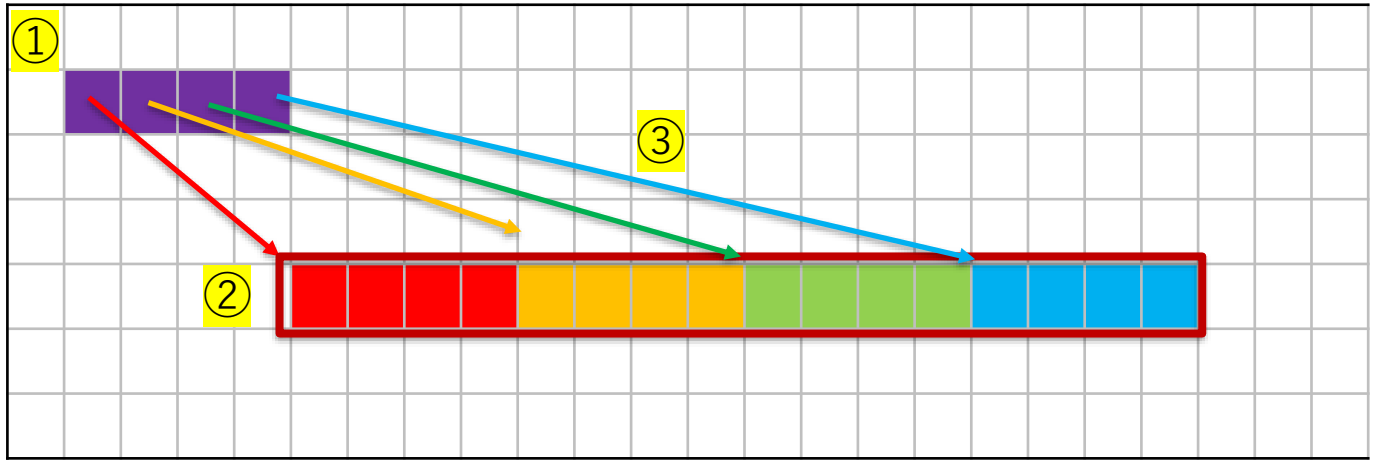
メモリの連続性が保証された動的二次元配列

```
double **a;

① a = malloc(m*sizeof(void*));
② a[0] = malloc(m*n*sizeof(double));
③ for(i=1; i<m; i++){
    a[i] = a[i-1] + n;
  }

  for(i=0; i<m; i++) {
    y[i] = 0.0;
    for(j=0; j<n; j++){
      y[i] += a[i][j] * x[j];
    }
  }

  free(a[0]);
  free(a);
```



計算部分に行列サイズを書かなくて良いのは便利ではある

第５回 演習

EX-5.1

- 実行時引数として入力された文字列を辞書順に並び替えるプログラムを作成する
  - 具体的には、dictionary_sort関数を作成する
    - 戻り値、引数は自由設定
  - 文字列の比較にはstrcmp関数を使う
    - strcmp(s1,s2)と呼び出したときの戻り値は、文字列s1が文字列s2よりアルファベット順で後ならば正の値（並び替え対象）、同じならば0となる。
  - ポインタのポインタ **sorted が指すデータを次ページのように出力した時、ソートされた文字列が出力されるように実装すること
    - ソートされた文字列を格納するための配列を新たに確保しても、しなくてもどちらでも良い
  - main関数の引数であるargcには引数の個数が保存されているので適宜使用する
    - 実行時引数の一つ目は、実行ファイル名（./a.outなど）なので注意

ひな形と実行例

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

int main(int argc, char **argv){

 if(argc < 2){
  printf("error: Enter the strings to be sorted as runtime arguments.\n");
  exit(1);
 }
 char **sorted;

 /*戻り値自由*/ dictionary_sort(/*引数自由*/);

 int i;
 printf("input:\n");
 for(i = 1; i < argc; i++){
  printf("%s ", argv[i]);
 }
 printf("\n");

 printf("output:\n");
 for(i = 0; i < argc-1; i++){
  printf("%s ", sorted[i]);
 }
 printf("\n");

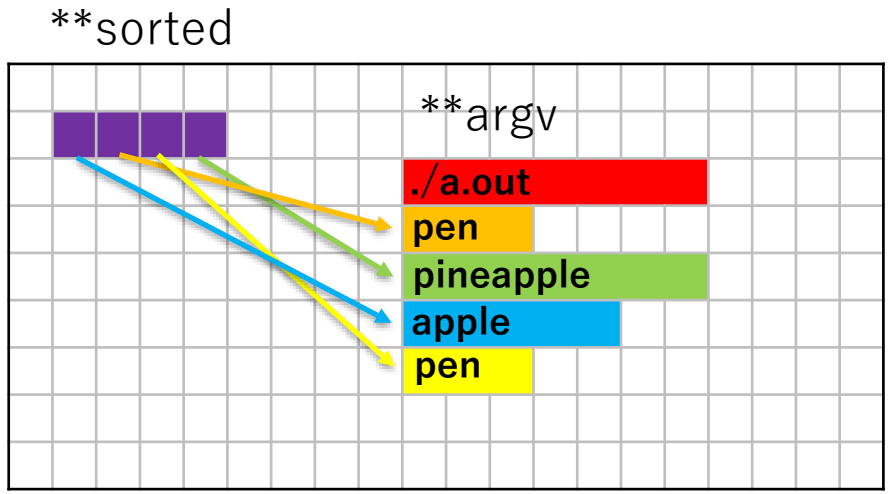
 return 0;
}
```

実行例

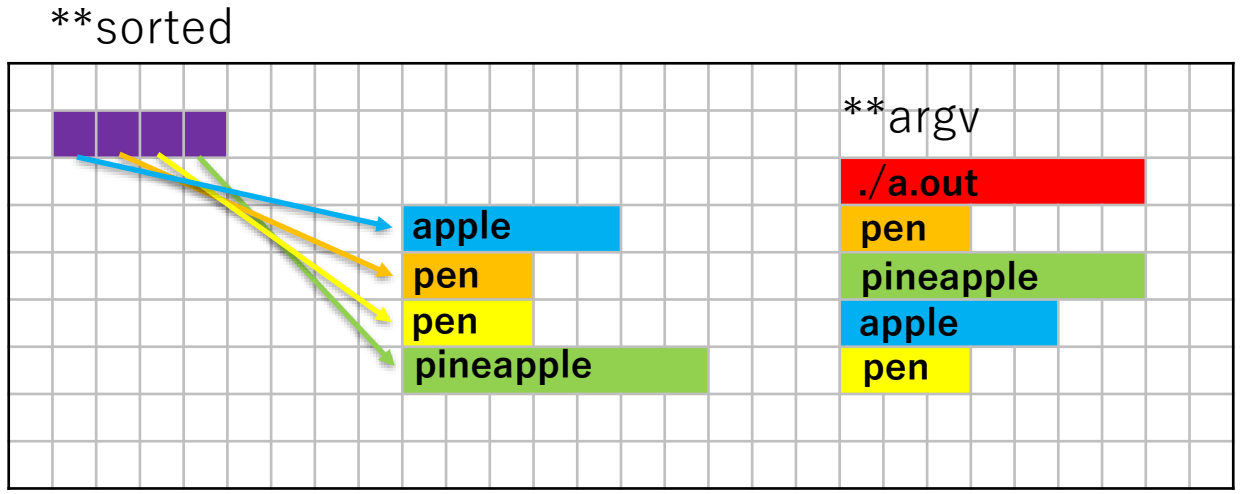
```
$ ./a.out pen pineapple apple pen
input
pen pineapple apple pen
output
apple pen pen pineapple
```

EX-5.1 ヒント

実装方針1



実装方針2



EX-5.1 関数定義及び呼び出しの例

```
void dictionary_sort(int n, char ***sorted, char **src);
```

ソートする文字列の数

**sortedのポインタ

ソート対象の文字列の配列

呼び出し元

```
dictionary_sort(argc-1, &sorted, &argv[1]);
```

argvの2要素目のアドレスを渡していて、
関数の中では2要素目が配列の先頭であるかのように
使うことができる

EX-5.2

- マインスイーパーマップ生成
  - マインスイーパーは、盤面内の爆弾の場所を推理するゲーム
  - この課題でやるのはマップ生成のみ（雛形はtact）
- main関数の引数
  1. m：盤面の縦の長さ
  2. n：盤面の横の長さ
  3. b：爆弾の数（b <= m*n）
  4. seed：rand()のためのseed値
     - rand()は乱数列を返す関数
     - seed値に従って、予め設定された乱数列を返す
     - 爆弾をランダムに配置するために使用

*	2	1	0
2	*	2	1
1	2	*	1
0	1	1	1

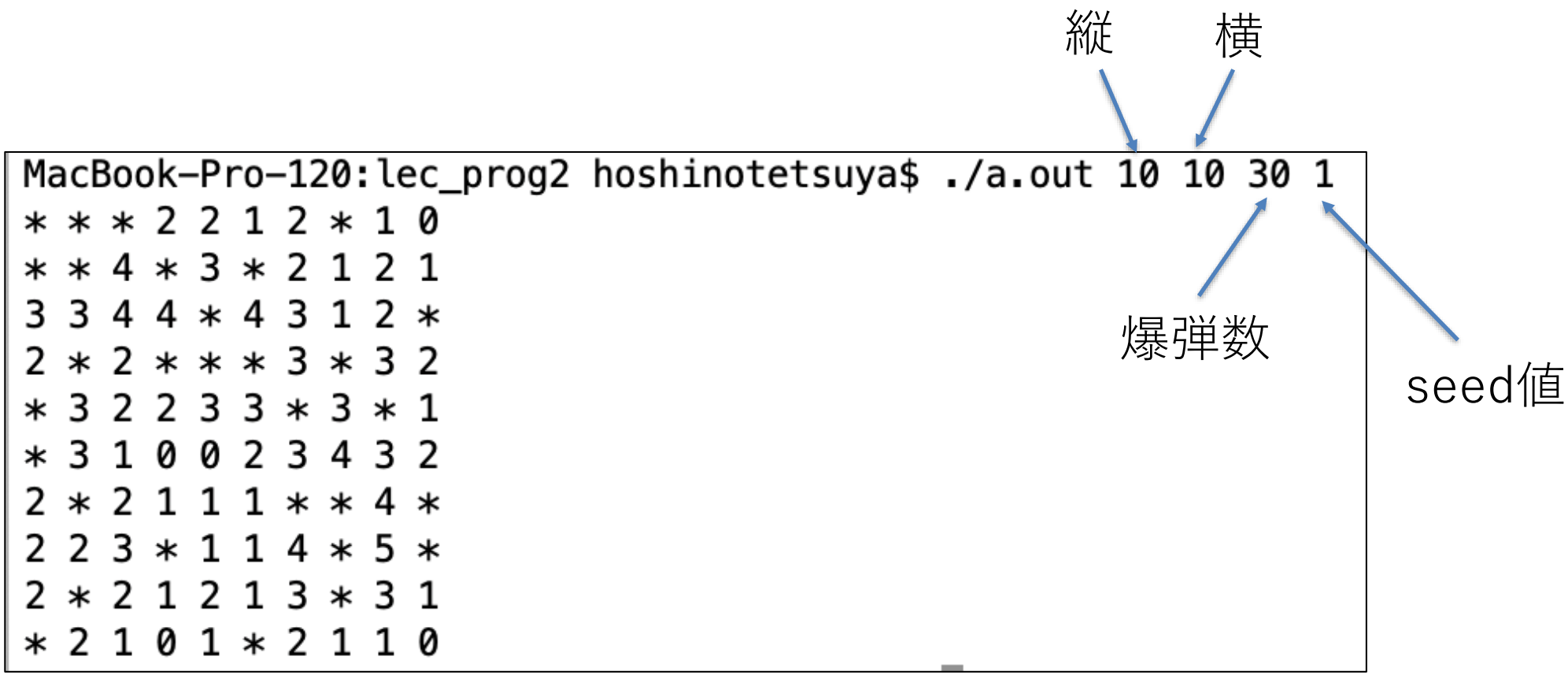
＊：爆弾
0 ～ 8：周囲8マスに存在する爆弾の数

EX-5.2

- 具体的には、make_count関数及びprint_field関数を作成する
- void make_count(int m, int n, char *bomb, int **count_addr);
  - 引数として渡されたcount配列に、爆弾の数を入力する関数
  - count配列の実体はmake_count関数内部でmallocし、main関数でfreeできるようにすること
  - bombには、爆弾の位置情報が予め入っている（fill_bombs関数参照）
  - count配列のbombが存在する場所は、-1か何か適当な数値で埋めるとよい

- void print_field(int m, int n, char *bomb, int *count);
  - 爆弾がある場所は * 、ない場所は周囲の爆弾の数を画面出力する
  - 縦の長さ m、横の長さ n で出力する
  - スペースの数などは厳密には決めないが、二次元でそれっぽく表示する

実行例




```
MacBook-Pro-120:lec_prog2 hoshinotetsuya$ ./a.out 10 10 30 1
* * * 2 2 1 2 * 1 0
* * 4 * 3 * 2 1 2 1
3 3 4 4 * 4 3 1 2 *
2 * 2 * * * 3 * 3 2
* 3 2 2 3 3 * 3 * 1
* 3 1 0 0 2 3 4 3 2
2 * 2 1 1 1 * * 4 *
2 2 3 * 1 1 4 * 5 *
2 * 2 1 2 1 3 * 3 1
* 2 1 0 1 * 2 1 1 0
```

提出

- 演習課題EX-5.1, EX-5.2
- TACTでソースファイルと実行結果（入力+画面に出力されたものをコピー＆ペーストしたテキストファイル）を提出
  - ソースファイルの名前は　学籍番号-5-{課題番号}.c　とすること
    - 例：252301111-5-1.c
  - 結果ファイルの名前は　学籍番号-5-{課題番号}.txt　とすること
    - 例：252301111-5-1.txt
  - 実行可能ファイル（gccにより生成されたファイル）の提出は不要
  - 2課題分計4ファイル提出
- 提出期限
  - **次々回（1/23）講義開始前まで**