プログラミング2　第4回

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！**）

関数の再帰呼び出し

- 再帰（recursion）
  - 何らかの対象をそれ自体で定義すること
  - プログラムでは関数がその関数自体を呼び出すこと

- 例題
  - 次の数列
    - $F_1 = 1$
    - $F_n = n * F_{n-1}$ $(n \geqq 2)$

注意：この手の数列は再帰の頻出例だが、再帰で書く必要は全くなく（むしろ効率が悪い）、ループで書いた方が良い

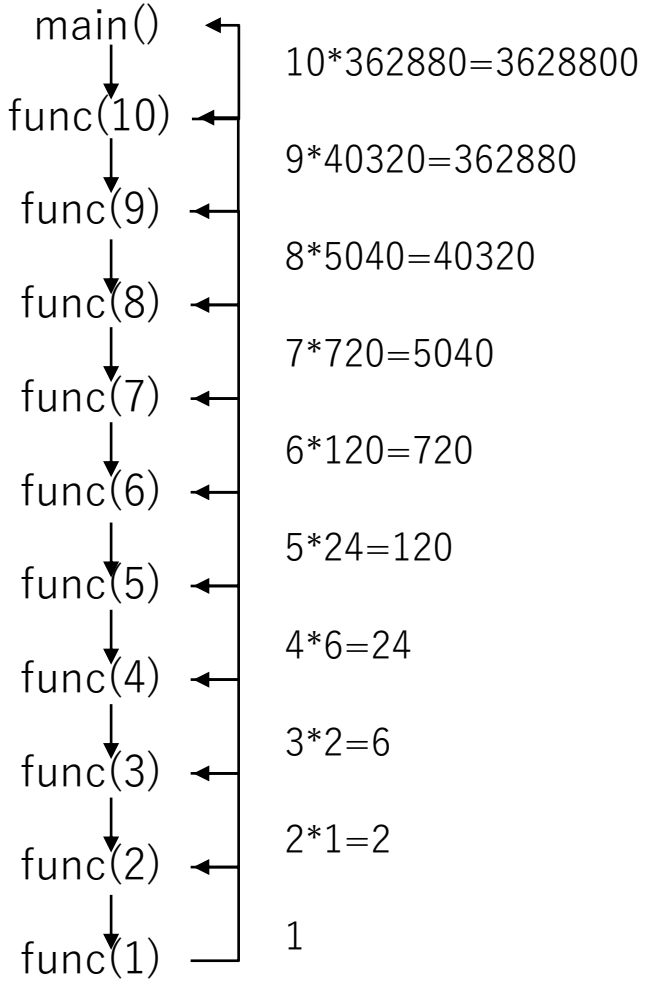
```c
#include <stdio.h>

int fact(int n)
{
  if (n==1) {
    return 1;
  } else {
    return n * fact(n-1);
  }
}

int main(void)
{
  printf("10! = %d¥n", fact(10));
  return 0;
}
```

終了条件（ないと永遠に再帰呼び出ししてしまう）

fact関数自体を呼び出し



フィボナッチ数

- N番目のフィボナッチ数は次の数列で表される
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_{n+2} = F_n + F_{n+1}$　　$(n \geqq 0)$
  - 具体的には
    0, 1, 1, 2, 3 ,5, 8, 13, 21, 34, 55, 89, 144, …

- 再帰プログラムといえばフィボナッチというくらいの頻出例だが、再帰で書く意味は薄い
  - 再帰という考え方を学ぶための例にすぎない
  - 右の速度をn = 45くらいで比べてみると良い
    - 48くらいからunsigned intの表現範囲を超える

再帰の例
簡潔に書ける

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

unsigned int fib(unsigned int n)
{
 if(n <= 1){
  return n;
 }else{
  return fib(n-1) + fib(n-2);
 }
}

int main(int argc, char **argv)
{
 if(argc!=2) {
  printf("usage: %s num\n", argv[0]);
  return 0;
 }
 printf("fib(n) = %u\n", fib(atoi(argv[1])));
 return 0;
}
```

再帰しない例
(main関数は
上と同じ)

```
unsigned int fib(unsigned int n)
{
 if(n < 2) return n;
 unsigned int fib_n_m2 = 0;
 unsigned int fib_n_m1 = 1;
 unsigned int fib_n;
 int i;
 for(i = 2; i <= n; i++){
  fib_n = fib_n_m2 + fib_n_m1;
  fib_n_m2 = fib_n_m1;
  fib_n_m1 = fib_n;
 }
 return fib_n;
}
```

クイックソート

```c
#include <stdio.h>


void quick_sort (int *array, int left, int right);


int main (void) {
  int array[10] = { 2, 1, 8, 5, 4, 7, 9, 0, 6, 3 };
  int i;

  printf("Before: ");
  for (i = 0; i < 10; i++) { printf("%d ", array[i]); }
  printf("\n");

  quick_sort(array, 0, 9);

  printf("After : ");
  for (i = 0; i < 10; i++) { printf("%d ", array[i]); }
  printf("\n");

  return 0;
}
```

```c
void quick_sort (int *array, int left, int right) {
  int i,j,pivot;

  if (left < right) {
    pivot = left;
    i = left;
    j = right+1;
    do {
      do {i++;} while(array[i] < array[pivot]);
      do {j--;} while(array[pivot] < array[j]);
      if (i < j) {
        int tmp = array[i];
        array[i] = array[j];
        array[j] = tmp;
      }
    } while(i < j);
    int tmp = array[pivot];
    array[pivot] = array[j];
    array[j] = tmp;
    pivot = j;

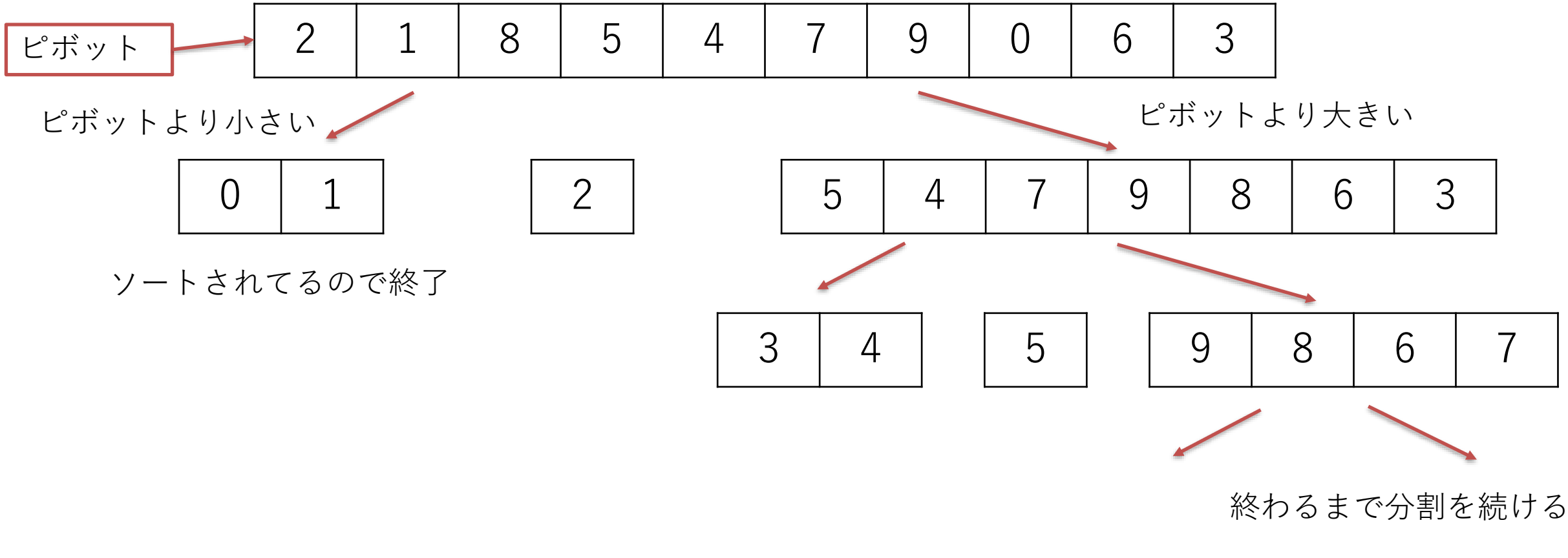
    quick_sort(array, left, pivot-1);
    quick_sort(array, pivot+1, right);
  }
}
```

- クイックソートも再帰呼び出しの有名な例である

ピボットの右と左に分けて再帰呼び出し

クイックソートのアルゴリズム

- ピボットとして適当な値（例えば左端）を定め、それより大きな値と小さな値に分ける
- 分けたものに対して同様の処理を行う
- 繰り返し処理を行い全てソート済になったら終了



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
  - 基本的には1 byte = 8 bit 単位でアドレスが割り振られている

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

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

```
int a = 48;
```

- 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

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

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;
```

int型のポインタ変数pを宣言

pに変数aのアドレスを代入

メモリ上の特定アドレスを指し示す矢印のように使われるのでポインタと呼ばれる

	アドレス	データ	
	…	…	
	1000	8752	
	1004	-12898	
a	**1008**	48	int型のデータ
	…		
	100004	65536	
p	100008	**1008**	int*型のデータ

ポインタの使用

- ポインタが示すアドレスのデータを間接的に使用する

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

int型のポインタ変数pを宣言

pに変数aのアドレスを代入

pに保存されたアドレスのデータを間接的に変更

	アドレス	データ	
	…	…	
	1000	8752	
	1004	-12898	
a	**1008**	**9999**	int型のデータ
	…		
	100004	65536	
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

ポインタ演算子

- ポインタを通じて変数にアクセスする
- アドレス演算子（ampersand, &）
  - 変数のアドレスを取り出す
- 間接演算子（asterisk, *）
  - ポインタの示すアドレスにある値にアクセスする
  - 乗算の*とは異なる（乗算は2項演算子、こちらは単項演算子）
- ポインタ変数の宣言の一般的な形式

```
型　*変数名;
```

ポインタの使用例１

```c
#include <stdio.h>

int main(void)
{
  int *p;
  int q;

  q = 199;
  p = &q;
  printf("%d¥n", *p);

  return 0;
}
```

ポインタ変数pにアドレス演算子&を
使って変数qのアドレスを代入

ポインタ変数pから間接演算子*を使って
pが示すアドレスの値を取り出し

- 実行例

```
$ ./a.out
199
```

ポインタの使用例２

```c
#include <stdio.h>

int main(void)
{
  int *p;
  int q;
  q = 199;
  p = &q;
  *p = 200;
  printf("%d¥n", q);

  return 0;
}
```

ポインタ変数pにアドレス演算子&を使って変数qのアドレスを代入

間接演算子*を使ってpが示すアドレスにint型データの200を代入

変数qの値を出力

- 実行例

```
$ ./a.out
200
```

例２の流れ

```
1.    int *p;
2.    int q;
3.    q = 199;
4.    p = &q;
5.    *p = 200;
6.    printf("%d¥n", q);
```

1,2

	アドレス例	データ
p	1000	不定
q	1004	不定

3

p	1000	不定
q	1004	**199　（数値）**

4

p	1000	**1004　（番地）**
q	1004	**199　（数値）**

5

p	1000	**1004　（番地）**
q	1004	**200　（数値）**

- 3行目で変数qに199が入る
- 4行目でポインタ変数pに変数qのアドレスが入る
- 5行目でpが指すアドレス（つまり変数q）に200が入る
- 6行目でqの値を出力すると200になっている

関数の引数とアドレス (1)

- 値を渡す

```
void func(int x) {
  x = x + 1;
}

int main(void) {
  int n = 5;
  func(n);
  printf("%d¥n", n);
  return 0;
}
```

	アドレス	データ
n	1000	5
	…	

- アドレスを渡す

```
void func(int *x) {
  *x = *x + 1;
}

int main(void) {
  int n = 5;
  func(&n);
  printf("%d¥n", n);
  return 0;
}
```

	アドレス	データ
n	1000	5
	…	

関数の引数とアドレス (2)

- 値を渡す

```
void func(int x) {
  x = x + 1;
}

int main(void) {
  int n = 5;
  func(n);
  printf("%d¥n", n);
  return 0;
}
```

	アドレス	データ
n	1000	5
	…	
x	2000	5

- アドレスを渡す

```
void func(int *x) {
  *x = *x + 1;
}

int main(void) {
  int n = 5;
  func(&n);
  printf("%d¥n", n);
  return 0;
}
```

	アドレス	データ
n	1000	5
	…	
x	2000	1000

変数xはメモリ上の別の場所に作られる。データは渡されたものの複製。

関数の引数とアドレス (3)

- 値を渡す

```
void func(int x) {
  x = x + 1;
}

int main(void) {
  int n = 5;
  func(n);
  printf("%d¥n", n);
  return 0;
}
```

変更されるのはあくまで変数xのデータ。nには影響なし。

	アドレス	データ
n	1000	5
	…	
x	2000	5 → 6

- アドレスを渡す

```
void func(int *x) {
  *x = *x + 1;
}

int main(void) {
  int n = 5;
  func(&n);
  printf("%d¥n", n);
  return 0;
}
```

	アドレス	データ
n	1000	5 → 6
	…	
x	2000	1000

間接演算子を使って、渡ってきたアドレスのデータにアクセスできる。

関数の引数とアドレス (4)

このようにポインタと間接演算子を使うことで、参照渡し**っぽい**ことができる。（Cはあくまで値渡し。間違ってるサイトも結構ある。それを糾弾するサイトもある。
https://qiita.com/raccy/items/d4c5e7995a8fc90109ee）

- 値を渡す

```
void func(int x) {
  x = x + 1;
}

int main(void) {
  int n = 5;
  func(n);
  printf("%d¥n", n);
  return 0;
}
```

出力は5

	アドレス	データ
n	1000	5
	…	
x	2000	6

- アドレスを渡す

```
void func(int *x) {
  *x = *x + 1;
}

int main(void) {
  int n = 5;
  func(&n);
  printf("%d¥n", n);
  return 0;
}
```

出力は6

	アドレス	データ
n	1000	6
	…	
x	2000	1000

ポインタ式

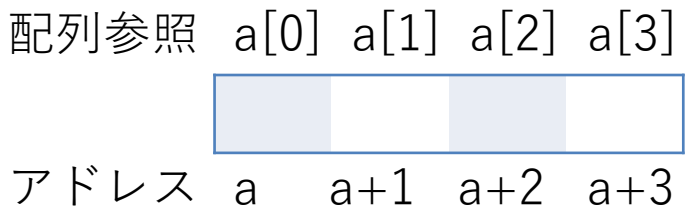
- ポインタ変数
  - 整数の加算・減算、インクリメント・デクリメントができる
  - ポインタに1を加えると、データ1個分だけ進んだアドレスを指すようになる
  - ポインタから1を引くと、データ1個分だけ戻ったアドレスを指すようになる
  - 1個分はデータ型に依存する
    - intは4バイト、doubleは8バイトなど
  - ポインタ演算が意味を持つのは、原則として同じ配列の内部と、その末尾の一つ先まで

配列とポインタ

- 添え字指定のない配列名だけを使うと、配列の先頭を指すポインタと同様に使える

```
int a[4];
```

  - 添え字指定のない配列名は　a
  - a[1]のアドレスはa+1と書けるため、*(a+1)でも値を取り出す（要素にアクセスする）ことが可能



  - 逆に言うと、間接演算子の代わりに配列の添え字を使うことができる
    - *p = 100; と p[0] = 100; は意味合いとして同じ

ポインタ式の使用例

```
1.  #include <stdio.h>
2.
3.  int main(void)
4.  {
5.    int ary[] = {1, 20, 300, 4000};
6.    int *p;
7.
8.    p = ary;
9.    printf("%d¥n", *p);
10.
11.   p++;
12.   printf("%d¥n", *p);
13.
14.   (*p)++;
15.   printf("%d¥n", *p);
16.
17.   return 0;
18. }
```

ポインタ変数pにint型配列の先頭アドレスを代入

pが指すアドレスをint型1個分だけ進める

pが指すアドレスの値に1を加える

- 実行例

```
$ ./a.out
1
20
21
```

ポインタ式を無理に使う必要はない。配列表現で代用可能。

配列表現で代用した例

ポインタ変数pにint型配列の先頭アドレスを代入

indexを1個分だけ進める

p[index]に1を加える

```
1.  #include <stdio.h>
2.
3.  int main(void)
4.  {
5.    int ary[] = {1, 20, 300, 4000};
6.    int *p;
7.    int index = 0;
8.    p = ary;
9.    printf("%d¥n", p[index]);
10.
11.   index++; //p++;
12.   printf("%d¥n", p[index]);
13.
14.   p[index]++; //(*p)++;
15.   printf("%d¥n", p[index]);
16.
17.   return 0;
18. }
```

- 実行例

```
$ ./a.out
1
20
21
```

前のページと出力は同じ。

アドレスの出力

- printf()で%pを使うと16進数でアドレスを出力できる

- 実行例

```
$ ./a.out
ip: 0x7ffce311737c  dp: 0x7ffce3117370
ip: 0x7ffce3117380  dp: 0x7ffce3117378
```

  - int型はアドレスが4増加（16進数なのでc → d → e → f → 0）
  - double型はアドレスが8増加

```
1.  #include <stdio.h>
2.
3.  int main(void)
4.  {
5.    int *ip, i;
6.    double *dp, d;
7.
8.    ip = &i;
9.    dp = &d;
10.   printf("ip: %p  dp: %p¥n", ip, dp);
11.
12.   ip++;
13.   dp++;
14.   printf("ip: %p  dp: %p¥n", ip, dp);
15.
16.   return 0;
17. }
```

int型とdouble型のポインタ変数にそれぞれint型変数とdouble型変数のアドレスを代入

アドレスを1個分だけ進める

第4回 演習

EX-4.1

- ○×ゲームのNPCを作る
  - 具体的には、NPCが置く場所i, jを返す関数npcと、npcの中で呼ばれる再帰関数npc_recを実装する
- 仕様
  - `void npc(char ox, int turn, int *i, int *j);`
    - 仮引数i, jをポインタ変数とする。これを使ってNPCが置く場所を返す。
    - よって実引数にはアドレスを受け取る（次ページのmain関数例を参考にする）
  - npc_rec関数の戻り値、引数は自由に設定していい。再帰で実装すること。
  - 実行時引数として、OまたはXを受け取り（次ページのmain関数例を参照）、それがプレイヤーの手番になる。（Oを入力したらプレイヤーがO、NPCがX）
- 採点
  1. NPCの勝ち確定の時に、ちゃんと勝ち確定の手を打てる。
  2. プレイヤーがどんな手を打っても負けない。
  3. （相手が最善手を打つ場合）どこに打っても引き分けのケースで、その中でも勝率が高そうな手を自動で選べる。（全探索すると勝ちパターンが多いとか）

1, 2ができてればOK

3ができると満点

EX-4.1について

- main関数の実装例
  - この通りである必要はない

```
#include <stdio.h>

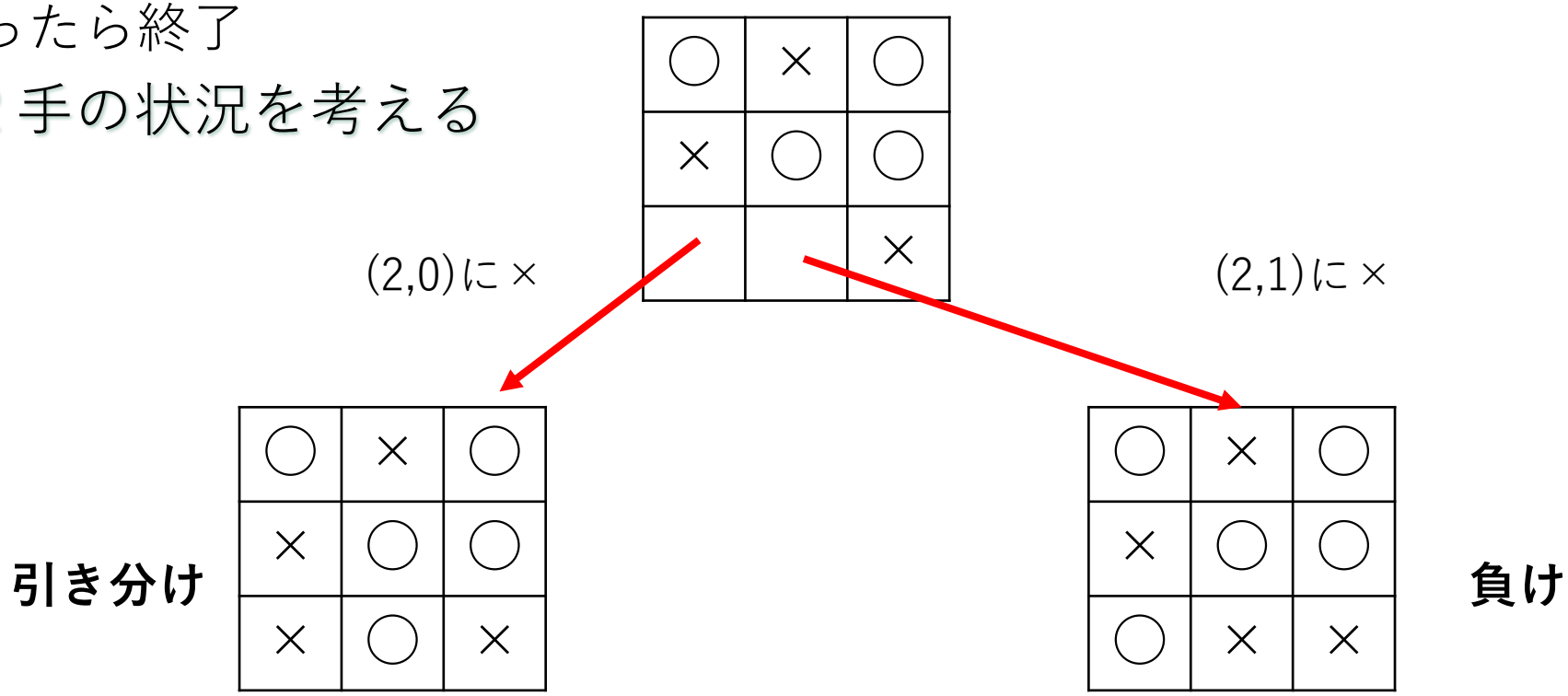
char table[3][3] = {0};
void print_table();
int put_ox(char ox, int i, int j);
int judge(char ox);
void npc(char ox, int turn, int *i, int *j);
/*戻り値は自由*/ npc_rec(/*引数は自由*/);

int main(int argc, char **argv){
 int i;
 int ii,jj;
 if(argc != 2){
  printf("usage: %s O_or_X\n", argv[0]);
  return 1;
 }else if(argv[1][0] != 'O' && argv[1][0] != 'X'){
  printf("error: Enter either O or X as a runtime argument. That will be your mark.\n");
  return 1;
 }
 char you = argv[1][0];
 for(i = 0; i < 9; i++){
  char ox = (i%2 == 0) ? 'O': 'X';
  printf("\n%c's turn\n\n",ox);
  if(ox == you){
   scanf("%d %d",&ii,&jj);
  }else{
   npc(ox,i,&ii,&jj);
  }
  while(put_ox(ox, ii,jj)){
   printf(“Try again.\n");
   scanf("%d %d",&ii,&jj);
  }
  print_table();
  if(judge(ox)){
   printf("%c win!\n",ox);
   return 0;
  }
 }
 printf("Draw\n");

 return 0;
}
```

ヒント

- 再帰関数の終了条件から考える
  - 盤面が埋まったら（９手目なら）終了
  - 勝敗が決まったら終了
- 次に、残り２手の状況を考える



つまりこの時、(2,0)に×を選べば引き分けに持ち込めるのだから、この残り2手の盤面は×にとって**引き分けの盤面**である

ヒント

- 残りN手の状況を考える
  - 残りN-1手が一つでも勝ちの盤面ならば、それを選べば良いので勝ち、この残りN手の盤面は勝ちの盤面
  - 残りN-1手の盤面が全て負けの盤面ならば、どれを選んでも負けなので、この残りN手の盤面も負けの盤面
  - そうでなければ引き分けの盤面
- 相手の勝ちは自分の負けであることに注意
- グローバル変数のテーブルを、シミュレーションのために一時的に操作した場合、シミュレーションを行う前の状態に戻さなくてはならないことに注意
  - テーブルをローカル変数にするのも可。
- 関数の戻り値を２つ設定することはできない。２つの戻り値が欲しい場合は、npc関数のように引数にアドレスを渡して実装するのが良い

ヒント

- npc_recの実装方針例
  - 例えば、int npc_rec(char ox, int turn, int i, int j); と定義する
  - npc_recを、turn手順目で、テーブルのi, j番目にoxを置いた時に、置いた結果が勝ち盤面なのか、引き分け盤面なのか、負け盤面なのかを返す関数と考える
    - 例えば、勝ちなら1, 分けなら0, 負けなら-1を返す
  - i,j番目に置いてみて（table[i][j] = ox; としてみて）、judge関数でoxの勝ちかどうか判定する。勝ちなら1を返す。返す際、i,j番目に置いてみたoxを元に戻す（table[i][j] = 0;にする）。
  - judge関数で勝ち負けは決まらず、かつ最終ターンだった場合、引き分けの0を返す。上と同様に置いてみたoxを元に戻す。
  - 勝ち負けつかず、最終ターンでもない場合、相手の手番にして（oxを反転して）、turn+1手順目のnpc_rec関数を呼ぶ。
  - tableの空白のところ全てに置いてみる（空白の数分npc_recを呼ぶ）。
    - 相手が勝つなら1が返ってくる。（つまり自分は負ける）
  - turn+1手順のnpc_recで返ってきた全ての結果を見て、現在のturn手順目の盤面の勝ち・引き分け・負けを判定する
  - 判定結果を返す。返す際、置いてみたoxを元に戻す。
  - この通りの実装でなくても全く構わない

提出

- 演習課題EX-4.1
- 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-4-{課題番号}.c　とすること
    - 例：252301111-4-1.c
  - 結果ファイルの名前は　学籍番号-4-{課題番号}.txt　とすること
    - 例：252301111-4-1.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
  - 1課題分計2ファイル提出
- 提出期限
  - **次々回（1/9）講義開始前まで**