プログラミング2　第2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！**）

第一回課題について

- 論理演算の結果は、int型の変数に保存できる
  - C言語では、0がfalse、0以外（一つ以上1のbitがある）がtrue
    - 前回口頭で1がtrue、1以外がfalseと言った気がするが、間違い

```
int flag1 = (i % 3 == 0);
int flag2 = (i % 8 == 0);
if (flag1)
    …
```

- printf()関数は自動で改行しない

```
for(i = 0; i < 5; i++)
    printf(“%d “,i);
printf(“¥n”);
```

“0 1 2 3 4 ”と出力

¥nで改行のみ出力（¥は半角バックスラッシュ）

ビット演算、シフト演算

- C言語にはビット列を操作するための演算子がある
  - Pythonにもあるが発想がC寄り

演算子	構文	意味	例
&	a & b	AND	0110 & 0011 → 0010　（ビットを比較して両方１なら1、そうでなければ０）
|	a | b	OR	0110 | 0011 → 0111　（ビットを比較してどっちか一つでも1なら1）
^	a ^ b	XOR	0110 ^ 0011 → 0101　（ビットを比較して異なる場合は1、そうでなければ０）
~	~ a	NOT	~0110 → 1001　（ビットを反転）
>>	a >> b	論理右シフト	1010 >> 2 → 0010　（ビットを単にb回右に移動する。型がUnsignedの場合。）
		算術右シフト	1010 >> 2 → 1110　（最上位の符号ビットで上位ビット側を埋める。Signed 0110 >> 2 → 0001　の負数の場合に算術シフトが選ばれることがある）
<<	a << b	左シフト	1010 << 2 → 1000　（ビットを単にb回左に移動する。）

負数の右シフトは環境によって結果が変わることがあるため、
なるべく使わない方がいい。ビット演算するならUnsigned型を使うべき。

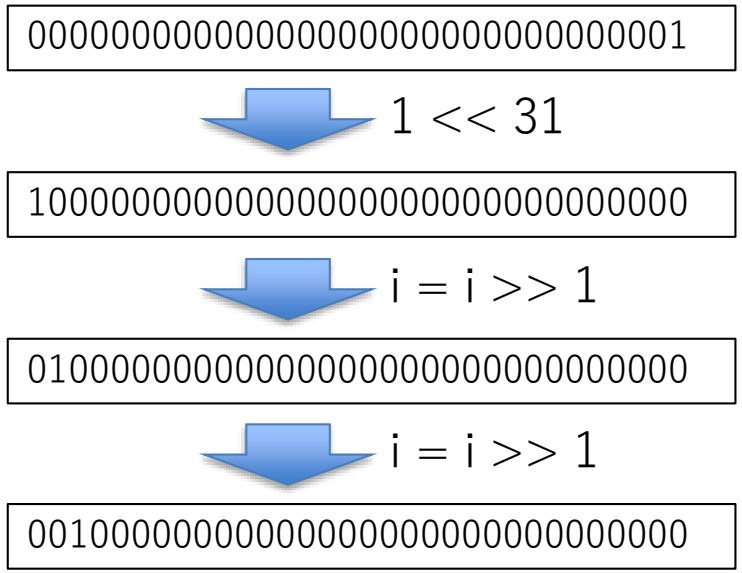
ビット演算、シフト演算の例

EX-1.1の答えではない。
EX-1.1は、見た目が対応する10進数に変数として保存し、printf出力する必要がある。

- ４桁の正整数をビット表現する例

```
#include <stdio.h>

int main(){
  unsigned int gakuseki_bango = 2023;
  unsigned int i;
  for(i = 1 << 31; i > 0; i = i >> 1){
    unsigned int bit = (i & gakuseki_bango) ? 1 : 0;
    printf("%u",bit);
  }
  printf("\n");
  return 0;
}
```



C言語では0以外はtrue

三項演算子　（条件式）？（trueの時の値）：（falseの時の値）
以下のif文とほぼ同じ
if (条件式) (trueの時の実行);
else (falseの時の実行);

ソースファイルへのコメント記述

- コメント（comment）
  - プログラマ（人間）向けにソースファイル中に記述する注釈のこと
  - コンパイラ（機械）はその内容を無視する
- 1行コメントの書き方

```
/* This is a comment */
```

```
// This is a comment
```

// はC99から

- 複数行コメントの書き方

```
/*
 Version 1.0
 Developed by Nagoya Taro
 */
```

```
/*
 * Version 1.0
 * Developed by Nagoya Taro
 */
```

長い複数行コメントは右例のように書くことを推奨

式内部での型変換

- 異なるデータ型の混在
  - Cでは1つの式中に型の異なるデータを混在させることができる
  - 全てのオペランド（被演算子）を一番大きなオペランドに合わせて変換する型の昇格（type promotion）を行って演算する
  - 型変換は演算1つずつに逐次適用される

```
#include <stdio.h>

int main(){
  int a = 4;
  int b = 6;
  float c = 6;

  printf("int/int   %d\n",b/a);
  printf("float/int %f\n",c/a);
  printf("int/int*float/int %f\n",b/a*c/a);
  printf("float/int*int/int %f\n",c/a*b/a);
  return 0;
}
```

int/intなので、b/a = 1。
1 * c は floatに変換され 6.0
6.0 / a は floatに変換され 1.5。

float/intなので、floatに変換され、
c/a = 1.5。
1.5 * b は floatに変換され、9.0。
9.0 / a は floatに変換され、2.25。

上の結果

```
int/int   1
float/int 1.500000
int/int*float/int 1.500000
float/int*int/int 2.250000
```

代入文での型変換

- 代入文の右辺と左辺で型が異なる場合
  - 右辺の型が左辺の型に変換される
  - 左辺の型が右辺の型に比べて
    - 大きい場合、特に問題ない
    - 小さい場合、一部の情報が失われる可能性がある
  - 浮動小数点数から整数への変換では小数部が失われる
  - 変換後の値が型に収まらない場合は意味のない値になる
- 大きいサイズから小さいサイズへの型変換
  - 整数では上位ビットが失われる
  - 浮動小数点数では精度が落ちる
  - ※ float型からdouble型への変換は精度や正確さの向上を伴うものではない

EX-1.1では、
```
long long int upTo4Digits = 1023;
```
としていたが、型変換するので本当は `int` で十分。

型変換

- doubleをfloatに型変換、floatをdoubleに型変換するプログラム

```
#include <stdio.h>

int main(){
  double  d = 1.0123456789;
  float   f = d;
  double d2 = f;

  printf("%12.10lf\n",d);
  printf("%12.10lf\n",f);
  printf("%12.10lf\n",d2);
  return 0;
}
```

- double型の変数 d には小数点以下10桁の数値が保存されている
- float 型の変数 f に代入によって型変換すると、小さい桁の部分の情報が失われる
- double型の変数に float 型の変数を代入する分には問題ない。ただし、代入し直しても失われた小さい桁の部分は戻らない。

- 実行結果

```
$ ./a.out
1.0123456789
1.0123456717
1.0123456717
```

型キャスト

- 型キャスト（type cast）での型変換
  - 型キャストの一般的な形式

(型)値

```
#include <stdio.h>

int main(){
  int a = 4;
  int b = 6;
  float c = 6;

  printf("int/int   %d\n",b/a);
  printf("float/int %f\n",c/a);
  printf("int/int*float/int %f\n",(float)b/a*c/a);
  printf("float/int*int/int %f\n",c/a*b/a);
  return 0;
}
```

floatで型変換してみる

上の結果

```
int/int   1
float/int 1.500000
int/int*float/int 2.250000
float/int*int/int 2.250000
```

ちゃんと2.25になる

ユーザー定義の関数

- 関数（function）
  - ある仕事を行うための文のグループ
  - プログラムの可読性や再利用性が向上する
- ユーザー定義関数を含むソースファイルの書き方（右図参照）
  - ソースコードは上から読んでいく
  - 上で定義されていない関数は使えない
  - 関数プロトタイプ（function prototype）
    - 関数の定義（名前と型だけ）を予め記述しておくもの
  - 戻り値の型
    - return文で返す値の型を記述する
    - return文がない（値を返さない）場合はvoidを指定する

```
#include <stdio.h>

/* 関数プロトタイプを宣言 */

int main(void)
{
/* … */
}

戻り値の型 関数1(仮引数リスト)
{
/* … */
}

戻り値の型 関数2(仮引数リスト)
{
/* … */
}
```

関数プロトタイプの書き方

- 関数プロトタイプ

戻り値の型　関数名(仮引数リスト);

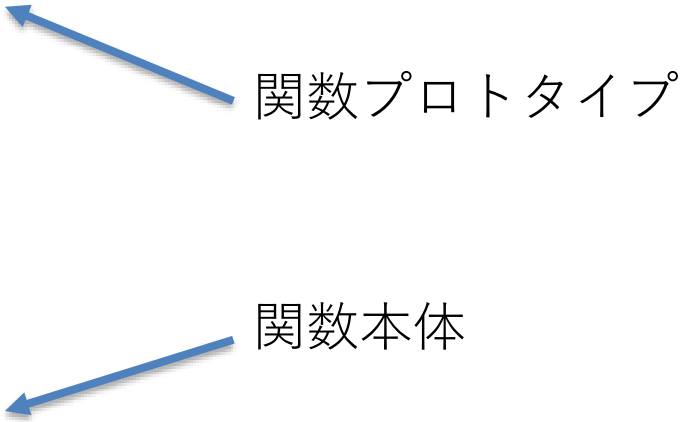
文末のセミコロンを忘れずに!

- 具体例

```
void print_hello(void);

int main(void)
{
  print_hello();
}

void print_hello(void)
{
  printf("Hello, World¥n");
}
```



関数の引数

- 関数の引数
  - 関数は0個以上の引数を取ることができる
  - 関数は引数の値を受け取る変数である仮引数（formal parameter）を宣言する必要がある
  - 仮引数リストには変数の型名と変数名を記述し、カンマ区切りで複数宣言できる
  - 引数が0個の場合は仮引数リストにvoidを記述する（省略可）
  - 仮引数はブロック内で通常の変数と同様に使用できる
  - 関数を呼び出すときに指定する変数は実引数（actual parameter）と呼ばれる
  - 仮引数名と実引数名が一致している必要はない

- 仮引数リストの書き方

```
戻り値の型　関数名(型名 仮引数名1, 型名 仮引数名2);
```

return文とexit関数

- return文
  - 呼び出した時点でその関数を終了する
  - 引数を1つとることができ、関数の戻り値となる
  - main関数から呼び出すとプログラムが終了する
  - 例 `return var;`
- exit関数
  - 呼び出した時点でプログラム自体を終了する（関数の呼び出し元には戻らない）
  - 引数を1つとることができ、正常終了ならばEXIT_SUCCESS（0でも良い）、失敗終了ならばEXIT_FAILUREを指定する
  - 使用するにはヘッダーファイルstdlib.hのインクルードが必要
  - 例 `exit(EXIT_FAILURE);`
- その他
  - 異常終了させるabort()もあるが、プログラムが正しく終了処理されない可能性があるため注意が必要

break文, continue文

- break文
  - 通常の終了判定式を経ずにループから脱出
- continue文
  - それ以降の残りの文を実行せず、次の繰り返し処理に進む
- 使用例
  - 1～10までの階乗を計算
  - ただし、階乗の値が1万を超えたらそれ以降は計算しない

```
int i, n = 1;
for (i=1; i<10; i++) {
    n *= i;
    printf("%d != %d¥n", i, n);
    if (n > 10000) {
        break;
    }
}
```

ユーザー定義関数の作成

- int型の引数を2つとり
その和を戻り値として
返す関数
add_int()
を含むプログラム

```
#include <stdio.h>

int add_int(int a, int b);

int main(void)
{
  int a = 5;
  int b = 6;

  int n = add_int(a, b);
  printf("a + b = %d¥n", n);
  return 0;
}

int add_int(int a, int b)
{
  return a + b;
}
```

関数プロトタイプ宣言

※プロトタイプ宣言では
仮引数名を省略すること
もできる
int add_int(int, int);

add_int()の呼び出し
戻り値は変数nに代入

関数の本体

グローバル変数とローカル変数

- グローバル変数(global variable)
  - 関数の外で宣言
  - 変数の有効範囲(scope)はプログラム内
  - プログラムが終了するまで存在する
- ローカル変数(local variable)
  - 関数(ブロック)内で宣言
  - 変数の有効範囲は関数(ブロック)内
  - 関数が終了すると変数は消滅する
  - 関数の仮引数もローカル変数
  - グローバル変数と同じ変数名も使える
    - ただし同じ名前のグローバル変数は
      その関数(ブロック)では使えない
    - （ローカル変数が優先されるということ）

```
#include <stdio.h>

void func(void);
int counter = 0;

int main(void)
{
  int n;
  func();
  n = counter;
  printf("%d¥n", n);
}

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

ブロックと変数の有効範囲

- ブロック（blocks）
  - 中括弧で囲まれた1つ以上の文をまとめた単位
  - ブロックは局所的な有効範囲を生成
  - 関数以外にもif文やfor文によってもブロックができている

```
int a, b;
...
if (a > b) {
    int c = a % b;
    if (c == 0) {
        int g = a;
        ...
    }
    ...
} else {
    ...
}
```

ここで変数cを宣言することはANSI Cでも問題ない

このifブロックでは変数cとgの両方が有効

入れ子のifを抜けたここでは変数gは有効範囲外

このブロックでは変数cもgも有効範囲外

ブロックと変数の有効範囲：ループとスコープ

- ループ内で宣言したローカル変数・ローカル配列はループボディに入る度に初期化されてしまう（ループイタレーションをまたいで維持されない）ことに注意

```
int a, b;
...
for (...; ...; ...) {
    int c;
    ...
    c = ...;
}
```

- 前回のループステップでどのような値だったかは関係なく変数cはここで毎回初期化される（未定義の状態から始まる）
- ループイタレーションをまたいで維持したい情報はより広いスコープの変数（配列）に持たせる必要がある

配列

- 配列（array）
  - 全てが同じ型を持ち、共通の名前によってアクセスできる変数のリスト（並び）のこと
  - 配列中のそれぞれの変数は配列要素（array element）と呼ぶ
- 1次元配列を宣言するときの一般的な形式

```
型 変数名[サイズ];
```

- 宣言の例 (int型で20個の要素を持つ配列)

```
int ary[20];
```

- 配列要素へのアクセス
  - 配列名とアクセスしたい要素番号(インデックス、index)を大括弧([ ])に記述
  - 配列aryの前から2番目の要素に値を代入する例

```
ary[1] = 4;
```

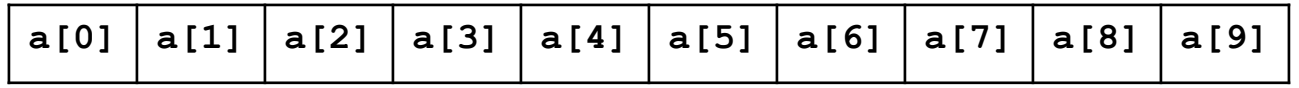
- C言語のインデックスは必ず0から始まる。
- 0から始まらないプログラミング言語もある（Fortran、R、Lua、BASICなど）

Cの配列に関する注意点

- 1次元配列はメモリ上に連続で確保される
  - サイズ変更などは不自由だが、効率が良い

```
int a[10];
```

メモリ上でのイメージ



- 配列添え字の範囲チェックはない
  - 範囲を超えてアクセスするコードを書いてもコンパイルは成功する
  - 動作は未定義である。エラー終了、動いたように見える、他のデータを壊す場合もある。



```
int a[10];
a[10] = 5;
```

- サイズnだと添え字最大値はn-1なので誤ったコード
- コンパイル時にこのエラーチェックはない（コンパイラに依存）

- 配列名だけを使って配列全体への操作を記述するような方法は提供されていない



```
int a[10], b[10];
/* ... */
a = b;
```

- このように書いても配列bの全要素の値が配列aに代入されるという動作にはならない
- このコードはコンパイルエラーになる

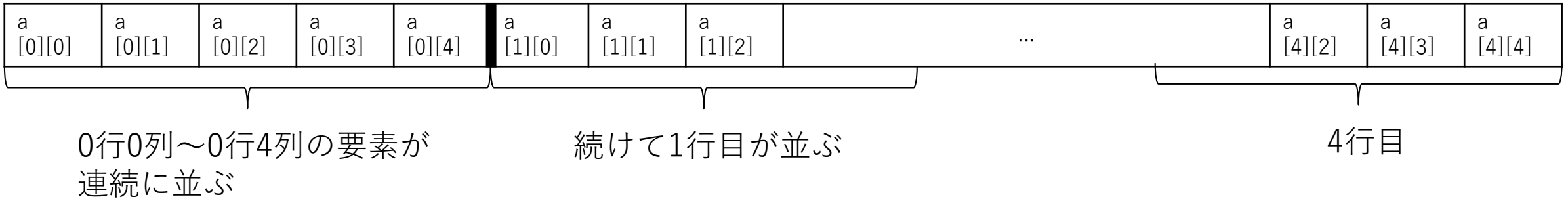
多次元配列

- 2次元以上の配列は大括弧（角括弧）の数を増やす

```
int a[5][5];
double x[100][10][3];
```

5×5のint型2次元配列 a
100×10×3のdouble型3次元配列 x

- 2次元配列のメモリ上における配置
  - Cは行優先(Row-major)スタイル



文字列の取り扱い

- C言語における文字列
  - 文字列のためのデータ型は存在しない
  - 文字（char）の配列で文字列を取り扱う
  - 文字列の終端として最後に必ずNULL文字（¥0 バックスラッシュゼロ）を入れる
- 具体例

```
char name[80];
name[0] = 'O';
name[1] = 'g';
name[2] = 'i';
name[3] = 'n';
name[4] = 'o';
name[5] = '¥0';
printf("%s¥n", name);
```

配列の初期化

- 配列の各要素にも宣言時に初期値を設定できる
- 1次元配列の初期化の一般的な形式

```
型 配列名[サイズ] = {値のリスト};
```

- 例

```
int today[3] = {2017, 12, 11};
```
中括弧の中にカンマ区切りで値のリストを書く

```
char a[2] = {'N', 'U'};
```

```
char pref[10] = "Aichi";
```
文字列は中括弧をつけずに文字列定数とすればNULL文字も追加される

```
int pwr[] = {1, 2, 4, 8, 16};
```
サイズ指定しない場合は初期値に合わせてコンパイル時に調整される

```
char univ[] = "Nagoya University";
```
サイズ指定なしは文字列に便利

```
int data[5] = {0};
```
値リスト数がサイズより少ない場合、残りは0になる
上は {0, 0, 0, 0, 0} が入る

```
int data[5] = {1};
```
下は {1, 0, 0, 0, 0} が入る

```
int x[3][3] = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```
2次元の例

配列の初期化：注意点

- 空の配列を宣言することはできない
  - OK
    ```
    int pwr[] = {1, 2, 4, 8, 16};
    ```
    サイズ指定しない場合は、初期値に合わせてコンパイル時にサイズが調整される
  - NG
    ```
    int pwr[];
    ```
    コンパイルエラー
  - 名前だけを確保しておくことはできる
    ```
    int *pwr;
    ```
    プログラム実行開始後に長さを決めたい場合はこの方法（詳細は後の講義で）
  - 右辺側を空っぽ（？）にすることはできる
    ```
    int data[5] = {};
    ```
    長さ5の配列、初期値は0

プログラム外部からの文字列等の入力方法

- scanf()関数
  - 書き方の例 `scanf("%d", &n);`
  - scanf()関数に到達すると入力待ち状態になり、キーボードから入力できる
  - 第1引数である2重引用符(" ")の中はフォーマット指定子
    - フォーマット指定子はprintf()と同じ
    - 上のように%dなら、整数の入力を受け付ける
  - 第2引数以降は入力値を代入する変数のアドレス（後述）
- main関数の引数（後述）
  - 本講義で多用
  - 実行時のコマンド引数として入力
    - 全部文字列として扱われるので、適切な型に変換が必要
  - 自動化の際に便利

scanf()関数の利用例

- キーボードから整数と実数の値を入力し、それぞれint型とdouble型の変数に格納するプログラム

```
#include <stdio.h>

int main(void)
{
  int n;
  double x;

  printf("Input an integer number: ");
  scanf("%d", &n);

  printf("Input a real number: ");
  scanf("%lf", &x);

  printf("n= %d, x= %lf¥n", n, x);
  return 0;
}
```

int型変数の入力

double型変数の入力

Pythonで
`print("...")`
`n = input()`
と書いているのと同じだが、
・静的型付けなのでフォーマット指定子が必要
・変数名の前に &（アドレス演算子）が必要

“&”って何？

- &はアドレス演算子
  - 変数のメモリアドレスを返す
- メモリアドレス
  - メモリは多数の0,1の並びなので、管理のための番地が振ってある
  - ロッカーに例えるなら、ロッカー番号がアドレスに相当する
- 変数とメモリアドレス
  - 変数はロッカーの名札みたいなもの。変数a＝99、変数aが016番を使っているなら、
    - &a は 016
    - a は 99

ロッカー

001 ■	002 ■	003 ■	004 ■
005 ■	006 ■	007 ■	008 ■
009 ■	010 ■	011 ■	012 ■
013 ■	014 ■	015 ■	016 ■

なぜscanf()はアドレスを引数に取るのか

- 前提としてC言語の関数は、値渡し方式である
  - 参照渡しという方式の言語もある
  - 値渡しでは、**変数の値のコピー**を関数に渡す
    - コピーはメモリ上の別のところに作られる
  - よって、関数の中で変数の値を書き換えても、それはコピーなので関数呼び出し元の変数には影響がない

```
#include <stdio.h>

int add1(int a){
  a = a + 1;
}

int main(){
  int a = 1;
  add1(a);
  printf("%d\n",a);
}
```

add1()のa

main()のa

1を足すのはadd1のa

001 ■	002 ■	003 ■	004 ■ 1+1
005 ■	006 ■	007 ■	008 ■
009 ■ 1	010 ■	011 ■	012 ■
013 ■	014 ■	015 ■	016 ■

このprintf()が参照するのは009なので、出力は1

なぜscanf()はアドレスを引数に取るのか

- 変数の中身を書き換えるために変数のアドレスを渡す
  - 値渡しには変わりないので、アドレスの値のコピーが関数に渡り、コピーがメモリ上の別のところに作られる
  - アドレスを保存する変数を**ポインタ**という
    - 変数の前のアスタリスクを間接演算子という
    - 右のプログラムを a = a + 1 とすると、004番の中身である009に1が足されてしまい、main()のaは変更されない
  - 指定されたメモリアドレスを直接書き換えることで、結果的に呼び出し元の変数の値も変わる

```
#include <stdio.h>

int add1(int *a){
  *a = *a + 1;
}

int main(){
  int a = 1;
  add1(&a);
  printf("%d\n",a);
}
```

*aにはmain()のaのアドレス009が保存されている

009番に1を足す

main()のa

001 ■	002 ■	003 ■	004 ■ **009**
005 ■	006 ■	007 ■	008 ■
009 ■ **1+1**	010 ■	011 ■	012 ■
013 ■	014 ■	015 ■	016 ■

このprintf()の出力は2

main関数の宣言と実行時引数の利用（1/3）

- 引数を取る場合、main関数はこのように書く

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

- 第一引数はmain関数実行時の引数の数
  - ただし、実行されているプログラムそのものの情報を含むため、最低でも1
- 第二引数以降は実際の引数
  - argv[0], argv[1], … にそれぞれ文字列として入ってくる
    - 引数として数字を与えてもプログラム中では文字列として見える
    - atoi関数を使えば数字として扱うことができる
  - argv[0]は実行したファイルのパス

main関数の宣言と実行時引数の利用（2/3）

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

int main(int argc, char **argv) {
  printf("%d args¥n", argc);
  printf("%d %s¥n", 0, argv[0]);
  if(argc>1){
    printf("%d %s¥n", 1, argv[1]);
  }
  if(argc>2){
    printf("%d %s¥n", 2, argv[2]);
  }
  return 0;
}
```



```
$ ./a.out
1 args
0 ./a.out
$ `pwd`/a.out      ※`pwd`は現在のディレクトリ
1 args
0 /mdhome/home4/nr2635614/work/a.out
$ `pwd`/a.out nagoya
2 args
0 /mdhome/home4/nr2635614/work/a.out
1 nagoya
$ `pwd`/a.out nagoya university
3 args
0 /mdhome/home4/nr2635614/work/a.out
1 nagoya
2 university
```

main関数の宣言と実行時引数の利用（3/3）

引数を数字として扱いたいときは
n = atoi(argv[1]);
などとして変換する必要がある。

```
int main(int argc, char **argv) {
  int n;
  printf("%d args¥n", argc);
  printf("%d %s¥n", 0, argv[0]);
  if(argc>1){
    n = atoi(argv[1]);
    printf("%d %d¥n", 1, n);
  }
  if(argc>2){
    n = atoi(argv[2]);
    printf("%d %d¥n", 2, n);
  }
  return 0;
}
```



```
$ ./a.out 2022 11 22
4 args
0 ./a.out
1 2022
2 11
$ ./a.out 2022 nagoya
3 args
0 ./a.out
1 2022
2 0
```

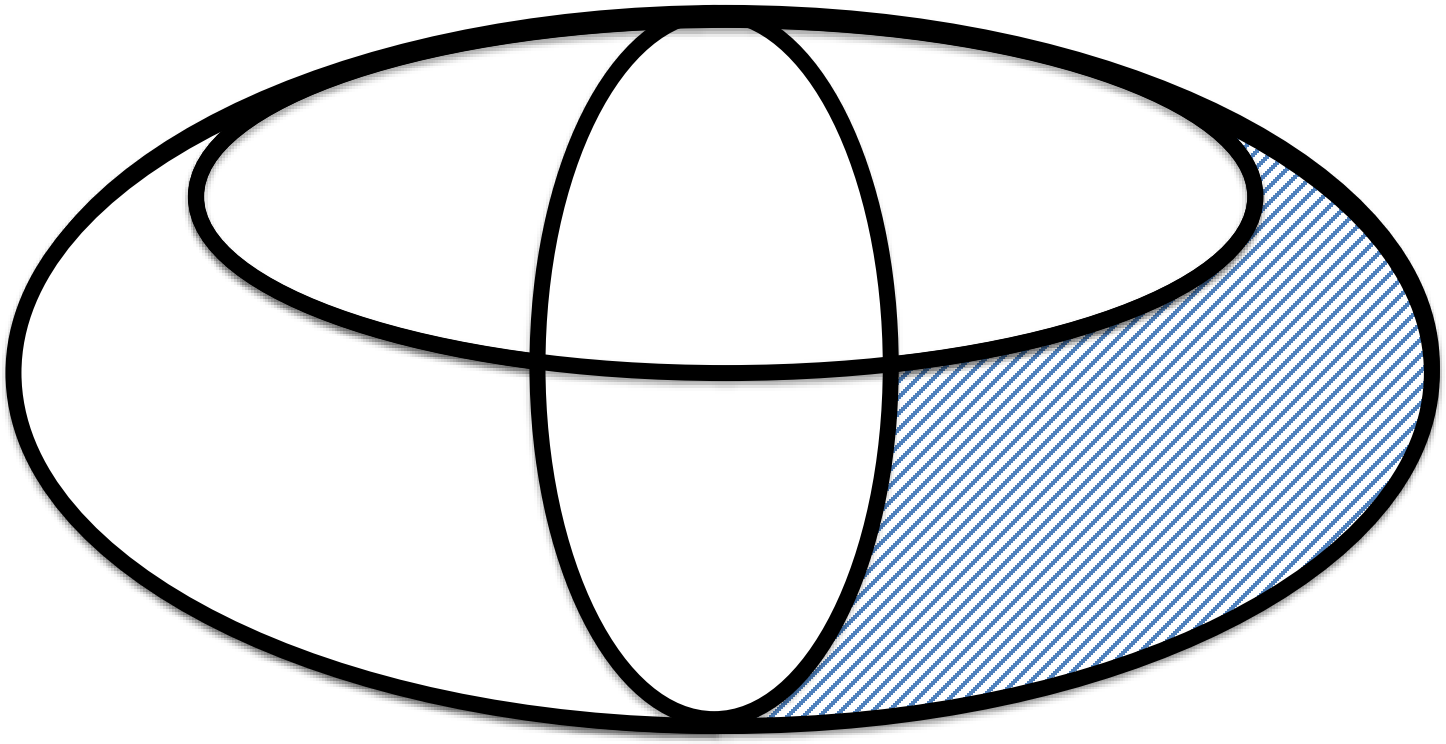
うまく変換できなかった（当然）

- intではなくlongに変換するatolや、doubleに変換するatofなどもある
- これで「引数として受け取った値を用いて計算を行う」ようなプログラムがある程度自由に作れるはずである

第２回 演習

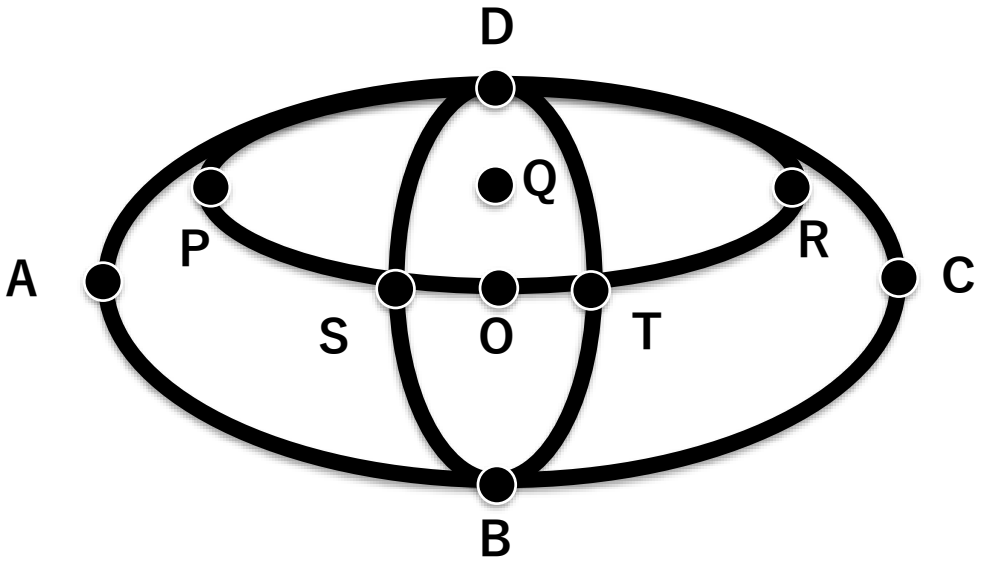
EX-2.1

- 下図のように３つの楕円が重なった図形の斜線部分の面積（の近似値）を求めよ



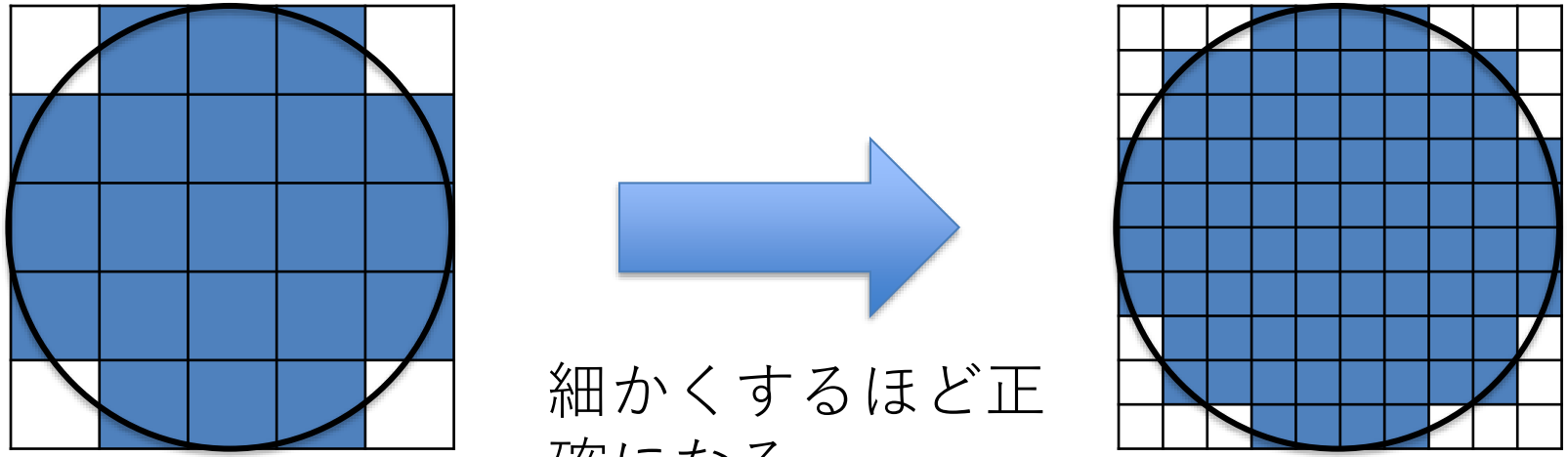
EX-2.1　図形の寸法

- 一番大きい楕円
  - ABCDを通り中心がO
  - 楕円の一番長い場所 AC = 1
  - 楕円の一番短い場所 BD = 0.5
- 二番目に大きい楕円
  - PORDを通り中心がQ
  - 楕円の一番長い場所 PR = 0.75
  - 楕円の一番短い場所 OD = 0.25
- 一番小さい楕円
  - SBTDを通り中心がO
  - 楕円の一番短い場所 ST = 0.25
  - 楕円の一番長い場所 BD = 0.5



EX-2.1　実装の方針

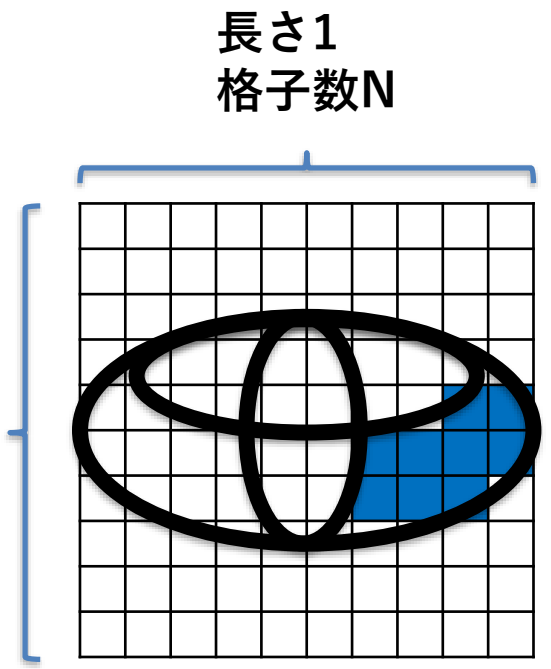
- 領域を細かく区切って近似する
  - 格子の中心が図形に入っているかどうかを判定
  - （図形に入っている格子数）/（全格子数）＝ 求める面積の近似値



EX-2.1　実装の条件

- 入力された座標が楕円に入っているかどうかを判定する関数is_in_ellipseを実装する
  - int is_in_ellipse(double x, double y, double cx, double cy, double a, double b);
    - 戻り値：入っていたら0以外の整数（基本は1）、入ってなかったら0。
    - x, y：判定する点のx, y座標
    - cx, cy：楕円の中心座標
    - a, b：楕円の長半径と短半径。a, bどっちが長(短)半径か決める必要はない。

- 長さ1 × 1の領域をN × Nの格子に区切り、格子の**中心が**入っているかどうかを判定する
  - 格子の中心座標を計算する必要がある
  - 一辺の格子数Nはmain関数の引数として、実行時に与える



関数の中身を実装

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

int is_in_ellipse(double x, double y, double cx, double cy, double a, double b);

int main(int argc, char **argv){
  if(argc != 2){
    printf("how to use: %s N\n",argv[0]);
    return 0;
  }
  int n = atoi(argv[1]);
  double s = 0.0;

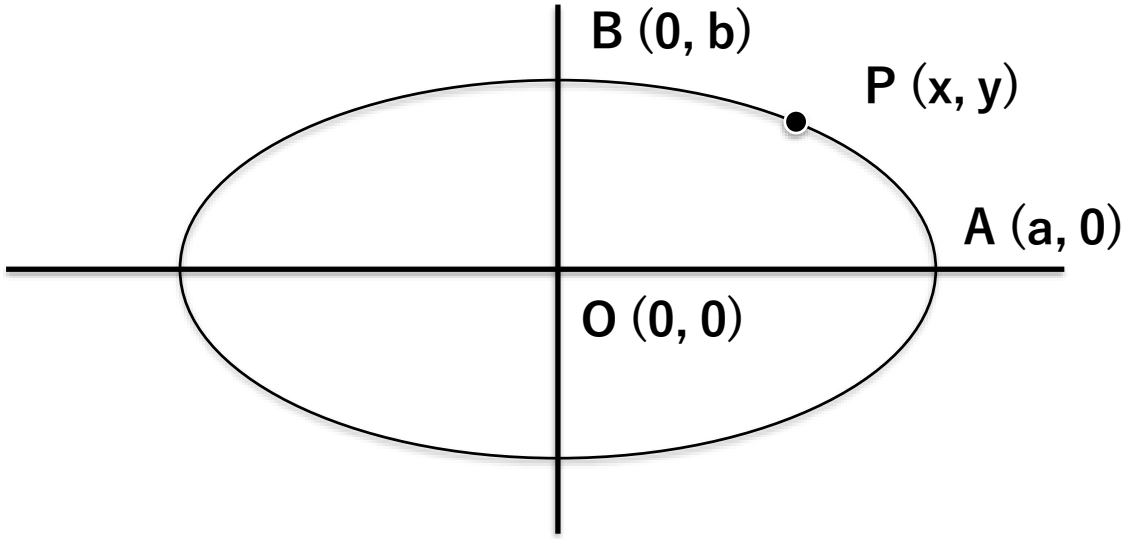
  /* この辺を実装 */

  printf("S = %f\n", s);
  return 0;
}

int is_in_ellipse(double x, double y, double cx, double cy, double a, double b){
  /* この辺を実装 */
}
```

ヒント

- 計算の優先順位に注意。不安なら()をつけること
- 型の変換に注意。
- 楕円上の点 P(x, y)は次を満たす
  - $-\frac{x^2}{a^2}+\frac{y^2}{b^2}=1$
  - つまり楕円に入っている場合は？
- 楕円の面積は$S = \pi ab$



EX-2.2

- 3*3の○×ゲームを実装せよ
  - ○が先手、×が後手とする
  - ○はO（大文字のオー）、×はX（大文字のエックス）で代用する
  - scanf()関数で、○×の置き場所を入力する
    - scanf()関数の入力書式は“%d %d”で、i, jの順番とする
  - テーブルは2次元のグローバルな配列とする
  - 先手・後手の手番を交互に繰り返し、盤面が埋まったら、あるいは途中でどちらかが勝ったらそこで終了する
- 具体的には以下の関数を実装する
  - int put_ox(char ox, int i, int j)
  - void print_table()
  - int judge(char ox)
- 次次回でNPCを作ってもらう予定

0,0	0,1	0,2
1,0	1,1	1,2
2,0	2,1	2,2

テーブルの座標(i,j)

EX-2.2

- int put_ox(char ox, int i, int j)
  - テーブルのi, jにO,Xを入力する
  - ○×の手番を示すchar型変数oxと、scanf()関数で入力されたテーブル座標i, jを引数とする
  - 置けた場合は0を、既に○×があったりテーブルの範囲外を指定され置けなかった場合は1を返す
  - 1が帰ってきた場合は、呼び出し元で再入力を促す
- void print_table()
  - テーブルの内容を出力する
  - 何も入っていない場所は-（ハイフンとスペース）を出力
  - テーブル更新後に毎回出力
- int judge(char ox)
  - ゲームが終局したかどうかを判定する
  - 現在の手番の人が勝ったなら1を返す
  - 引き分けの判定までしなくていいです。tableが埋まったら引き分けで終わってください。

関数の中身を実装

```c
#include <stdio.h>

char table[3][3] = {0};
void print_table();
int put_ox(char ox, int i, int j);
int judge(char ox);

int main(){
  int turn;
  for(turn = 0; turn < 9; turn++){
    char ox = (turn%2 == 0) ? 'O': 'X';
    printf("\n%c's turn\n\n",ox);
    int i, j;
    scanf("%d %d",&i,&j);
    while(put_ox(ox,i,j)){
      printf("Try again.\n");
      scanf("%d %d",&i,&j);
    }
    print_table();
    if(judge(ox)){
      printf("%c win!\n",ox);
      return 0;
    }
  }
  printf("Draw\n");
  return 0;
}

void print_table(){
}
int put_ox(char ox, int i, int j){
}
int judge(char ox){
}
```

出力例

毎ターンテーブル出力

```
$ ./a.out

O's turn

2 2
- - -
- - -
- - O

X's turn

2 2
Try again.
1 1
- - -
- X -
- - O

O's turn
```

```
2 0
- - -
- X -
O - O

X's turn

1 2
- - -
- X X
O - O

O's turn

2 1
- - -
- X X
O O O
O win!
```

自動採点しないので、
出力は厳密に同じでなくて良い

勝敗が決まったら終わる

ヒント

- 記述量が多いので、一つ一つデバッグしながら実装してください
- 量が多いだけで、内容はそれほど難しくないはず

提出

- 演習課題EX-2.1, EX-2.2
- TACTでソースファイルと実行結果（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-2-{課題番号}.c　とすること
    - 例：252301111-2-1.c
  - 結果ファイルの名前は　学籍番号-2-{課題番号}.txt　とすること
    - 例：252301111-2-1.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（12/19）講義開始前まで**