C言語速習コース
基礎知識

C言語とは

- 手続き型言語
  - 問題解決の手続きを『順次記述』していく
    - 数字を入力してください
    - 入力された数字を3倍にしてください
    - 結果をディスプレイに表示してください

      …
- 主な用途
  - システムソフトウェア（OS、コンパイラなど）
  - 科学技術計算（高速な計算を行わせたいもの）
  - 電化製品などの組み込みシステム (Nintendo DS)

C言語プログラミングの手順

1. テキストエディタでソースコードを書く
   - emacsやviなどのテキストエディタ
   - eclipseなどの開発環境
   - WordやWriterは（普通は）使わない

2. コンパイラで実行可能形式に変換（コンパイル）する
   - gcc ソースコード （-o 出力ファイル名）
     ※出力ファイル名を指定しないとa.outになる
3. 実行する
   - ./a.out

- メモ
  - コンパイラによる変換を行わずにいきなり実行できるプログラムもあります
    - 例：シェルスクリプト、Perl、Python、Ruby、etc.

C言語のフォーマット

おまじない
（外部宣言の読み込み ）

関数（手続き）の宣言
変数（※）の宣言

関数（手続き）の定義
main関数よりプログラムは始動

ファイルの拡張子は .c

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

int test();
int rv;

int main(int argc, char**argv){
        rv = test(3);
        printf("%d¥n", rc);
        return EXIT_SUCCESS;
}
int test(int v){
        return v*3;
}
```

順不同（例外あり）

行列積を計算してみよう(その0)

- $A * B = C$
  - $\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix} \begin{pmatrix} b_{00} & b_{01} & b_{02} \\ b_{10} & b_{11} & b_{12} \\ b_{20} & b_{21} & b_{22} \end{pmatrix} = ?$

- 最終的には
  - ファイルからデータを読み込み計算する
  - Aは 2x3, Bは 3x4 の行列

行列積を計算してみよう(その１)

Hands On !!

- おまじないを書いてください
  - Ｃ言語の多くの場合のテンプレートになる記述
  - 全部半角で！！
- $ emacs matmal.c

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

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

        return EXIT_SUCCESS;

}
```

宣言と定義

- 関数・変数の利用前には宣言もしくは定義が必要
  - 宣言
    - コンパイル時、「〇〇という名前の関数・変数を利用します」と表明する
  - 定義
    - 「〇〇という関数・変数の中身は××です」と指定する
    - 定義は宣言も兼ねる
      - 関数A が関数B を使うとき、Bの定義が関数A の定義より前に行われていればOK

変数の定義

- 定義方法
  - 変数型 変数名 (= 初期化値)

```
int a;
int b = 0;
double c = 0.0;
char d = ‘x’;
```

int型 変数名 a の宣言

int型 変数名 b を初期化値0 で定義

double型 変数名 c を初期化値0.0で定義

char型 変数名 d を初期化値 x で定義

- 基本型
  - 整数型: int, unsigned int, long int, short int ,…
  - 浮動小数点型: float, double
  - 文字型: char, unsigned char
  - 型なし: void

同じ種類の「型」でも入れられる情報の制限（最大値など）が異なる

演算子 (1/3)

- 算術演算子

演算子	種別	例	備考
＋	加算	$x + y$	
－	減算	$x - y$	
＊	乗算	$x * y$	
／	除算	x/y	整数型に使うと切り捨て
％	剰余	$x\%y$	整数型でしか利用できない

- 代入演算子

演算子	種別	例	備考
＝	代入	$x = y$	xにyを代入する
○＝	算術演算 代入	$x \mathrel{+}= y$ $x \mathrel{-}= y$	○は算術演算子 x○yを演算してxに代入

演算子(2/3)

- 比較・論理演算子
  - 比較演算子は偽の場合整数型0、真はそれ以外の整数が入る（ほとんどの場合1）

演算子	種別	例	備考
＜	小なり	$x < y$	
＜＝	以下	$x <= y$	
＞	大なり	$x > y$	
＞＝	以上	$x >= y$	
＝＝	等しい	$x == y$	代入との間違いに注意
！＝	異なる	$x! = y$	
＆＆	AND	$x\&\&y$	((x!=1) && (y==1)) 等とつかう
｜｜	OR	$x\|\|y$	
！	NOT	$!x$	単項演算子 !((x!=1) && (y==1))

演算子(3/3)

- 増分・減分演算子
  - 主に制御用に加減算と参照を組み合わせたもの

演算子	種別	例	備考
＋＋	後置増分	$x++$	xを返してから1加算
＋＋	前置増分	$++x$	1加算してからxを返す
－－	後置減分	$x--$	xをかえしてから1減算
－－	前置減分	$--x$	1減算してからxを返す

  - 注意！
    - if (a == b++){ … } : a が b に等しければ…実行
    - if (a == ++b){ … } : a が b+1 に等しければ…実行
- 演算子の優先順位
  - （強）単項＞２項算術＞２項比較＞２項代入＞２項論理（弱）
    - ２項代入式を参照すると、実は代入後の値が返ります
    - c=(a=2) とすると、 c=2になる
  - 不安なら括弧をつけて

行列積を計算してみよう(その2)

Hands On !!

- a x b + c = d を出力してみましょう
  - 実数変数の定義
  - 実数変数の代入

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

int main(int argc, char**argv){
        double a, b, c, d;
        a = 3.; // 3.0 の省略形
        b = .5; // 0.5 の省略形
        c = 1.5;
        d = a * b + c;

        printf("%f ¥n", d);

        return EXIT_SUCCESS;
}
```

- $ gcc matmal.c
  $ ./a.out

コメント文とコンパイラ

TIPS!!

- コメント文
  - プログラムにおける要点や次に実装を続ける人へのアドバイスなどを書く
  - コンパイラによって削除される

```
a = 3.; // 3.0 の省略形 ←全角文字もかける
b = .5; /* 0.5 の省略形
この書き方は改行もできます */
```

- コンパイラ
  - テキスト形式のプログラムを実行可能な機械語に変換するアプリケーション
    - $ gcc matmal.c
  - 同じCコンパイラでもさまざまな団体が独自の機能を持ったものを提供・販売している
    - gcc (GNU C) , icc (Intel C), cl (Microsoft Visual C)
    - 実行環境に適したものはプログラムが高速に動作したりする

制御構文

- 条件文
  - if (条件式){ 真・手続き } else { 偽・手続き }
    - else 以降は省略可能
- ループ
  - while (条件式) { ループ手続き }
    - 条件式を評価し真の間ループを続ける
  - do { ループ手続 } while (条件式)
    - ループを行ってから条件式を評価、真の間ループを続ける
  - for (初期化; 条件式; 再初期化) { ループ手続き }
    - 値を変えながらループをする。
    - for (i = 0; i < 10; i++){ print(“%d¥n”, i); }
      1. ループに入る際に初期化部分を実行
      2. 条件式を評価し、真ならループを行う
      3. ループ後、再初期化を行い2に戻る

条件式は真偽を返す式が入る

行列積を計算してみよう(その３)

Hands On !!

- １～１０まで出力してみましょう
  - 演算子
  - For ループ

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

int main(int argc, char**argv){
        int a;
        for(a=1;a<=10;a++){
                printf(“%d¥n”, a);
        }
        return EXIT_SUCCESS;
}
```

- $ gcc matmal.c
  $ ./a.out
- While ループで書くとどうなりますか？？

配列の定義とアクセス(1/2)

- 定義方法

```
int a[3];
int b[3] = {1, 2, 3};
int c[]  = {1, 2, 3};
int d[3] = {1, 2};
```

サイズ3の配列を作成、初期化なし

サイズ3の配列を作成、各要素を初期化

同上

サイズ3の配列を作成
0番、1番要素のみ1, 2 で初期化、後は 0

- アクセス方法
  - 最初の要素の添字は 0
  - i 番目の要素は a[i]
  - 配列のサイズを超えてアクセスしてもエラーは出ない時もある
    - 予期しないデータ改変

配列の定義とアクセス(2/2)

```
#include <stdio.h>
int main() {
  int a[3];
  int b[3] = {1, 2, 3};
  int c[]  = {1, 2, 3};
  int d[3] = {1, 2};
  int i;
  for (i = 0; i < 3; i++) {
    printf("a[%d]=%d, ", i, a[i]);
    printf("b[%d]=%d, ", i, b[i]);
    printf("c[%d]=%d, ", i, c[i]);
    printf("d[%d]=%d ¥n"  , i, d[i]);
  }
  printf("a[%d]=%d¥n", 3, a[3]);
  return EXIT_SUCCESS;
}
```

出力

```
a[0]=1627408016, b[0]=1, c[0]=1, d[0]=1
a[1]=-1,         b[1]=2, c[1]=2, d[1]=2
a[2]=1,          b[2]=3, c[2]=3, d[2]=0
a[3]=4198562
```

初期化されてない
何が出るか不定状態

定義(確保)されていない領域
出力の前に異常終了することもある
その際“Segmentation Fault”と出る

printf: 画面に出力する関数
後ほど説明

※配列全体を0で初期化

- Int a[3] = {0}

文字列

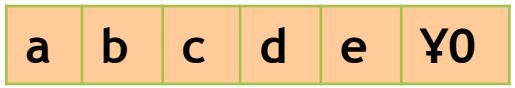
※この方法では日本語は使えないので注意
（C言語における日本語の扱いは
とても難しいので今回はやりません）

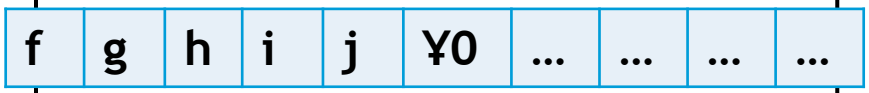
- Cの文字列は Char型の配列で実装されている
  - 文字列＝文字の集まり

```
#include <stdio.h>
#include <string.h>
int main() {
  char *c = “abcde”;
  char d[16];

  strncpy(d, “fghij”, 15);


  d[1] = ‘c’;
   return EXIT_SUCCESS;
}
```





← リテラル文字列
書き換え禁止メモリ上にとられる

← 15文字入る箱を宣言
1個は制御文字¥0が入る

← dに終端が来るまで、もしくは15文字の小さい方分”fghij”をコピー

← Dの2文字目をcに変更
つまり “fchij”になる

多次元配列

```
#include <stdio.h>
int main() {
  int a[3][4];
  int b[3][4] = {{1, 2, 3, 4},
                 {5, 6, 7, 8},
                 {9, 10, 11, 12}};
  int c[2][2][2] = {{{ 1, 2},{ 3, 4}},
                    {{ 5, 6},{ 7, 8}}};

  printf("%d¥n", b[0][0]);
  return EXIT_SUCCESS;
}
```

← サイズ3x4の配列を作成

← サイズ3x4の配列を作成して定数で初期化
(b[0][0]=1, b[0][1]=2, …)

← サイズ2x2x2の配列を作成して定数で初期化

← (0,0)の値にアクセス

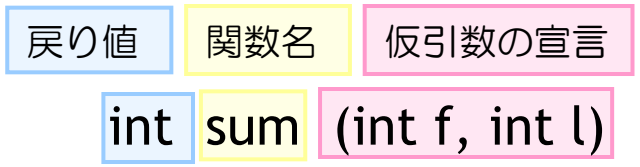
連続アクセスする場合は、右の数字が連続するように読む方が速い
詳しくは次回の講義で。

a[0][0], a[0][1], a[0][2], a[1][3], a[1][0], … , a[2][3] の順番

関数の定義(1/2)

```
#include <stdio.h>
int sum(int f, int l)
{
  int sum = 0;
  int i;
  for (i = f; i <= l; i++) {
    sum += i;
  }
  return sum; //返値の指定
  //関数定義と同じ型(例外:void型)
}
int main()
{
  int s;
  s = sum(1, 10);
  printf("sum=%d¥n", s);
  return EXIT_SUCCESS;
}
```

- int sum (int f, int l)
  - fからl (f <=l)の総和を求める関数



関数の定義(2/2)

- 注意：関数は使用する前に定義
  - しかしプロトタイプ宣言を行えばOK
    (※最近のコンパイラだとOKな場合もある)

プロトタイプ宣言

OK

```
#include <stdio.h>

int sum(int f, int l)
{
        :
  return sum;
}

int main()
{
        :
  s = sum(1, 10);
        :
}
```

NG

```
#include <stdio.h>

int main()
{
        :
  s = sum(1, 10);
        :
}

int sum(int f, int l)
{
        :
  return sum;
}
```

OK

```
#include <stdio.h>
int sum(int, int);

int main()
{
        :
  s = sum(1, 10);
        :
}

int sum(int f, int l)
{
        :
  return sum;
}
```

配列を引数にとる関数

```
int a[3];
int b[2][3];
int c[1][2][3];

void func(int a[]);
void func(int *a);
void func(int a[3]);

void func(int b[][3]);
void func(int (*b)[3]);
void func(int b[2][3]);

void func(int c[][2][3]);
void func(int (*c)[2][3]);
void func(int c[1][2][3]);
```

- 1次元配列の引数は以下で書ける
  - (型) 変数名[サイズ]
  - (型) 変数名[]
  - (型) *変数名
- 変数名[] と (*変数名) は等価
  - 一番左の次元のサイズ以外は可変長にできない
  - データのメモリ配置上の問題だが、ここでは割愛する

以上より多次元配列含めた例は左のようになる

一次元配列を用いた同次元可変長配列を受け入れる関数

TIPS!!

- (型) 関数名((型) 変数名[], int サイズ)
  - 一次元配列はどんな長さでも受け入れる
  - 配列の長さがわからないので、必要に応じてサイズを渡すことになる
- 一次元配列で多次元配列は表現可能
  - 次元ごとのサイズが必要になることが多い
  - (参考)この書き方をみるとどうしてもyに当たる値が必要だとわかる→省略不能

```
int a[12]; // x=3,y=4の時 3*4

void func(int *a, int x, int y){
  int i, j;
  for (i = 0; i < x; i++){
    for (j = 0; j < y; j++){
                printf("%d¥n", a[i*x+j]);
    }
  }
}
```

書き換わる引数

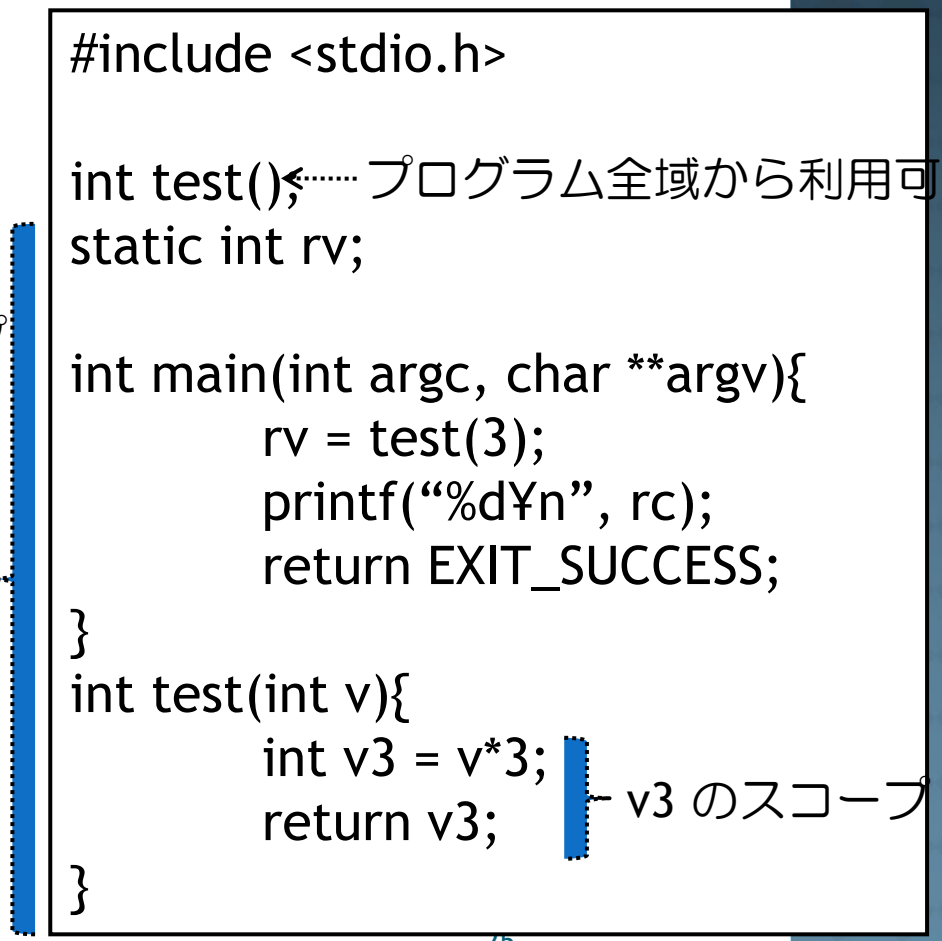
- 引数に配列をとったときのみ関数の中で書き換えると、呼び出した関数側でも値が変わってしまう
  - 本当は間違った理解だが、詳しい理由は次の講義で。

右の出力は
-1 0 0, 0
になる

```
#include <stdio.h>
void func(int *f, int l)
{
  f[0] = -1;
  l = -1;
}
int main()
{
  int f[] = {0, 0, 0}, l = 0;
  func(f, l);
  printf("%d %d %d, %d¥n", f[0],
                          f[1], f[2], l);
  return EXIT_SUCCESS;
}
```

スコープ

- 変数・関数の名前が参照される範囲
  - Global scope
    - プログラムの全ての場所から利用可能
  - File scope (static)
    - 宣言されたファイル内でのみ利用可能
  - Local scope
    - 特定のブロック内でのみ利用可能
    - 関数, {}で区切られた範囲 for ループ 等
- Global Scope を使いすぎるのはバグの元
  - どこで書き換えられるか不明に



PRINTFの話 (1/3)

- 指定されたフォーマットにそってコンソールに出力する関数
  - printf(“フォーマット指定文字列”, 変数, …, 変数);
  - printf(“今日は%d日です¥n”, day);

- フォーマット指定文字列（一例）
  - %<フラグ><フィールド幅><精度><変換指定文字>
    - フラグ:左詰め(-), 正符号付け(+) フィールド幅:数値出力幅 “%2d” = 整数2文字で出力
      “%02d” = 空きは0で埋める整数2文字
    - 精度:小数点以下の桁数 “%2.1f” = 整数部分2文字
      ＋小数1桁まで

PRINTFの話 (2/3)

- 変換指定文字
  - 後ろにつなげる変数の数と一致するように記述する必要がある。

指定文字	意味	データ型
%c	1文字として出力する	文字型
%d	10進数で出力する	整数型
%x	16進数で出力する	
%o	8進数で出力する	
%f	[-]dddd.dddddの形式で出力する	浮動小数点型
%e	指数形式で出力する	
%s	文字列として出力する	文字列

PRINTFの話 (3/3)

- エスケープシーケンス
  - フォーマットとして特殊な意味がある文字列
    - ¥ で始まる

エスケープシーケンス	意味
¥n	改行
¥a	警告音（音が出る）
¥t	タブ
¥b	バックスペース
¥¥	¥
¥’	‘
¥”	“
¥0	文字列の終端

行列積を計算してみよう(その４)

Hands On !!

- 特定の行列の積を計算してみよう
  - 配列
  - 関数の定義
  - 行列積のフローって？
    - A,B行列を設定する
      - makemat()
      - とりあえず全要素2を代入してください
      - 固定長なので配列のサイズはいらない
    - 行列掛け合わせ
      - matmal()
    - 結果を出力する
      - printmat()

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

int main(int argc, char**argv){
        double a[2][3], b[3][4];
        double c[2][4];
        makemat(a, b);
        matmal(c, a, b);
        printmat(c);
        return EXIT_SUCCESS;
}
```

局所変数の薦め

TIPS!!

- 変数はなるべく局所変数を使うように
  - 関数の中でブラックボックス化できる
  - 関数の中で参照する変数は、引数に書いてあるもの以外は、他所に影響を与えない

メンテしやすい＋改造しやすい＋読みやすい

```
void write_array(int a[], int n){
  int i; // a, n, i しか使わない
  for (i=0; i<n; i++){
     a[i]=i;
  }
}
int main(int argc, char **argv){
     int a[10];
     write_array(a, 10); // aをいじるんだな
     return EXIT_SUCCESS;
}
```

```
int a[10], n=10;
void write_array(){
  int i;
  for (i=0; i<n; i++){
     a[i]=i;
  }
}
int main(int argc, char **argv){

     write_array(); //何をするんだろう？
     return EXIT_SUCCESS;
}
```

SSCANFの話 (1/2)

- 文字列をフォーマット文字列に従って変数に格納する関数
  - sscanf(変数, “フォーマット指定文字列”, 変数, …, 変数);
  - sscanf(buf, “今日は%d日です”, &day);
  - sscanf(buf, “今日の曜日は%3sです”, dofweek);
    - 文字列の場合は & がいらない（配列だから）
    - “今日の曜日はSunですね” も引っかかります(前方一致)

- フォーマット指定文字列（一例）
  - %<フィールド幅><変換指定文字>
  - フィールド幅は変換する文字数か文字列の短い方
    - sscanf(“1234”, “%2d”, &a); なら a=12になる
    - sscanf(“1234”, “%8d”, &a); なら a=1234になる
    - 文字列を受けるときは、代入する変数のサイズ-1を超えないよう必ず指定すること(-1 分は ¥0)
  - 返値：何個代入したか

SSCANFの話 (2/2)

- 書式はprintfとほとんど同じだが・・・
  - doubleとfloat の扱いについて
    - printf: 両方 %f で受ける
    - sscanf: double は %lf, float は %f で受ける

printf, sscanf とも、もっと便利なフォーマット指定文字があるので、必要なら調べてみると良い

テキストファイル入力(1/3)

- プログラムを実行した場所にあるテキストファイルの開き方(fopen)
  - 第一引数
    - ファイル名
  - 第二引数
    - モード
      - r: 読み専用
      - w: 書き専用（上書き）
      - a: 追記（最後にかく）
      - r+: 読み書き
      - w+: 書き読み（上書き）
      - a+: 読み追記

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char**argv){
  FILE *fp;//ファイルポインタという
  fp = fopen("filename", "r");
  if(fp == NULL){
    perror("fopen:");
    exit(EXIT_FAILURE);
  } /* ここに処理 */
  fclose(fp);
}
```

上書きでは、開いた瞬間にファイルが空になってしまう。
w, a, w+, a+ではファイルがないと自動的に作成される。
使い終わったらfclose(fp)で閉じる

テキストファイル入力(2/3)

- ファイルの行毎読み込み (fgets)
- 第一引数
  - 読み込み先
- 第二引数
  - 読み込み先のサイズ
    - サイズ-1か改行があるまで読み込む。
      読めなかった分は次回読む
    - -1は¥0を自動付加するため。
    - 改行も読むので実質行のサイズより2つは大きくとる
- 第三引数
  - ファイルポインタ
- ファイルを全部読み終わるかエラーすると返値がNULLになる

```
char s[16]; //バッファという
while(fgets(s, 16, fp) != NULL){
    sscanf(s, “format”, &value);
}
```

行内の数字の数がわかっていれば sscanf を使って変数に代入

- sscanf(s, “%lf %lf %lf”, &(a[0][0]), &(a[0][1]), &(a[0][2]));
  - a は配列だが a[][]は配列じゃないので & が必要

テキストファイル入力(2/3)

- 文字列の分断(strtok)
  - 第一引数
    - 分断元文字列
  - 第二引数
    - 何の文字で分断するか
  - tpには前から数えて最初に分断した文字列が入る
    - 第一引数をNULLにしてもう一度呼び出すと、次に分断された文字列が入る。分断する文字は呼び出しのたびに変更可能
    - 何故 *tp なのかはここでは説明しないのでおまじないのようにつかう
    - 分断元文字列がなくなるとNULLを返す
    - なお、分断元文字列はこの関数のあと書き換えられているので中身を他の用途に再利用できないことに注意

```
#include <string.h>
char *tp;
tp = strtok(s, " ");
while(tp != NULL){
    sscanf(tp, "format", &value);
    tp = strtok(NULL, " ");
}
```

読み込んだらsscanf を使って変数に代入
ファイルを全部読み終わるかエラーするとNULLになる

テキストファイル出力

- ファイルの書き込み (fprintf)
- 第一引数
  - ファイルポインタ
- 第二引数以降
  - printf と同じ

```
fprintf(fp, “format”, value…);
```

このほかにも、低水準関数の fwrite, fread などがありますが、気になる人だけ調べてください

定数値の名付け

- 意味のある定数をプログラムに埋め込むのはメンテナンス性が下がって良くない
  - fgets での読み込み先変数（バッファ）のサイズなど
    - 後にサイズを変えたいと思ったときに様々な場所を変える必要がある
    - 定数そのままでは何を指した定数かわからず、変更漏れがでる
- マクロの利用

```
#define BUFFER_SIZE (16+2) /* 16文字/行 +2は終端と改行 (¥n¥0)
            演算子を含む場合は括弧を忘れない（テキストが置換されるため）
            ない場合、たとえば BUFFER_SIZE * 4 = 16+2*4 になってしまう  */
char s[BUFFER_SIZE];
while(fgets(s, BUFFER_SIZE, fp) != NULL){
            sscanf(s, “format”, &value);
}
```

PRINTF, SSCANFの仲間

TIPS!!

- たくさんある便利な printf, sscanf の亜種。
  - printf/fprintf/sprintf/vsprintf …
  - scanf/fscanf/sscanf/vsscanf …

- scanf 系に注意（特にscanf）
  - 読み込み切れなかったときに、読み込み途中のデータが残らないようにする書式が難しい
  - 読み込み限界サイズをフォーマッタに指定することを忘れがち
    - コンパイル時にエラーを出してくれない
  - 何故fgets/sscanf を使うの？
    - fgets ならサイズ指定を忘れた場合、引数が足りないというエラーが出る
    - fgets の段階でsscanfでとる値の文字数をチェックできるので、sscanf での限界サイズを指定し忘れても安心

使ってはいけないというわけではない、正しく使えばOK

MAIN関数の役割

- main関数
  - main関数は必ず最初に実行される関数（スタートポイント）
  - main関数が終わるとプログラムが終わる
  - 返値はプログラムが正常に動いたかどうかを返す
    - return EXIT_SUCCESS;
    - return EXIT_FAILURE;

MAIN関数とプログラムの引数

- int main(int argc, char **argv)
  - 引数はプログラム実行時に与えられる文字列の個数と、与えられた文字列（配列）
  - argv[0]には実行したファイル名が入るため、argcは最低でも1
    - $ ./a.out str1 str2 str3
      $ a.out str1 str2 str3 <- 出力

```
int main(int argc, char** argv){
      if(argc != 3){ return EXIT_FAILURE; }
                              //引数の数のチェック
      printf("%s, %s, %s, %s¥n", argv[0], argv[1],
                                        argv[2], argv[3]);
      return EXIT_SUCCESS
 }
```

行列積を計算してみよう(その5)

Hands On !!

- 行列積の計算を完成させよう
  - ファイル入力 2つ
  - A(2x3の行列) matmal_04.dat
    - 一行目 配列Aの一行目 （以下全てスペース区切り）
    - 二行目 配列Aの二行目
    - 三行目 配列Aの三行目
    - 四行目 配列Bの一行目
    - …
    - 七行目 配列Bの四行目
  - 出力はコンソールに (printf)
  - ファイル名をプログラム実行時に渡す
    - Main関数の引数
    - $ ./a.out filenameA filenameB

インターフェース・モジュール

TIPS!!

- 今回 matmal を関数化した理由
  - 何度も使う計算は世界の天才が効率の良いものを作っている
  - 関数を置き換えるだけで使える
    - もし呼び出すための引数や返値も同じなら、さらに便利

→インターフェース

よく使う関数の引数、返値を同じ宣言にしておき、様々な実装を使い分ける。

→モジュール

インターフェースが決められ、プログラムとして切り分けられる部品のこと。

※モジュールには他にいろんな意味もあるけど・・・
基本的には差し替え可能なように切り分けた部品群を示す