C言語の文法まとめ

C言語基本事項

- **大文字と小文字は区別される**
  - 通常プログラムは小文字で書く
  - 文の終わりには ; を入れる
  - コメントは　/*　　*/ で挟まれた領域または // 以降。コンパイル時には無視される。
- **一文が長く複数行に分ける場合は ;を入れずに改行**

```
a = 1.0 + 2.0 + 3.0 + 4.0 + 5.0
  + 6.0 + 7.0 + 8.0 +
    9.0 + 10.0;
```

C言語プログラムの基本構造

```
#include<ヘッダファイル名.h>

int main(){
    宣言部 (変数宣言など)

    実行部 (計算など)

}
```

プログラムはmain関数として実行される。
ヘッダファイルにはプログラム内でライブラリ関数などを用いる場合に読み込む。
数値計算で必要となるのは

stdio.h : 入出力関数(printf, scanf, fopenなど)
math.h : 数学関数(sin, cos, pow, expなど)
stdlib.h : 整数型絶対値(abs)
complex.h : 複素数型宣言、複素関数(cexp, creal, cimagなど)

変数の型

- 変数(数値を格納する入れ物)には型がある
- プログラムで使うすべての変数の型を明示的に指定する

型	
整数型	**int**
単精度実数型 (物理の数値計算ではあまり使わない)	float
倍精度実数型	**double**
複素数型(物理の数値計算ではあまり使わない)	float complex　(complex.hのインクルードが必要)
倍精度複素数型	double complex (complex.hのインクルードが必要)
論理型	bool　(stdbool.hのインクルードが必要)
文字型	**char**

太字はよく使うもの。

変数名：英数字とアンダーラインが使える。英字から始まる

変数の宣言例

```
int a;                          // 整数型で変数aを宣言
double b, c;                    // 倍精度実数型のbとcを宣言
double d = 0.0;                 // 倍精度実数型のdを宣言して初期値0.0を代入
double complex dt1;             //  倍精度複素数型のdt1を宣言
const int a = 100;              // プログラム中で値を変更しない変数(定数)にはconst修飾子をつける
char a =  “s“;                  // 長さ1文字の文字列aにaを代入
char a[5] = “abcd”              // 長さ4文字の文字列。最後にヌル文字¥0が必要なので確保する配列の大きさは5
```

変数について

- **変数はコンピュータのメモリ上の保持されるため表現できる値に上限がある。**
- 整数型：4バイト (=32ビット)
  (1バイト=8ビット、1ビットは0か1の情報をもつ)
  ⇒2進数で32桁、正負の表現に1ビット使う(これは正確な表現ではなく負の数は補数を使って表す)ので最小値は-2^{31}、最大値は $2^{31}-1$
- 単精度実数：4バイト、10^{-38}～10^{38}程度の範囲を約7桁の精度で表現
  →物理の計算では精度が不十分
- 倍精度実数：8バイト、10^{-308}~10^{308}程度の範囲を約16桁の精度で表現
- 単精度複素数型：8バイト、実部と虚部それぞれが単精度実数型
- 倍精度複素数型：16バイト、実部と虚部それぞれが倍精度実数型
- 論理型：値は真(true.)か偽(false.)
- 文字型：1文字1バイト

→物理の計算では実数と複素数型に基本的に倍精度のものを使う。

倍精度実数のメモリ上での形式(発展)

倍精度実数型：8バイト＝64ビット

$$実数 = (-1)^{符号} \times (1 + b_1 2^{-1} + b_2 2^{-2} + b_3 2^{-3} + \ldots + b_{52} 2^{-52}) \times 2^{(e-1023)}$$

この形式で近似されるため全ての実数値が表現できるわけではない

四則演算と代入文

- 数学の=(イコール)は左辺と右辺が等しいことを表すが
  プログラムの=は**右辺の計算結果を左辺の変数に代入する**、の意味
- 四則演算は +, -, *, /
- 累乗はない(*で何度も掛けるか、数学関数のpowを使う)
- **同じ型同士の演算は同じ型になる。**
- 整数と実数の演算→実数、実数と複素数の演算→複素数

```
a = 1.0;          // 倍精度実数型のaに1.0を代入
i = i + 5;     // 整数型iの値を5増やす (再帰代入)
i += 5;           // i=i+5;をこのように書くこともできる
a = 4*4*4:      // aは4の3乗 (64)
a = pow(4.0,3.0);   // aは4.0の3乗 (64.0)。powは入出力ともに倍精度実数型
a = 5 / 2;          // aは2となる。これは整数型の5を整数型の2で割っているため
/* 同じ型同士の演算となり、割り算の商の整数の2が右辺の計算結果となる。
   2.5にしたい場合は5または2のいずれかを実数にする。
   5と書くと整数、5.0と書くと実数となる */
```

インクリメント・デクリメント演算

i=i+1;　は　i++; または ++i; とかける。
i=i-1;　は　i--; または --i; とかける

前置形：　++i;　--i;
加算・減算された後に結果が実行される。

```
i=0;
printf(“%d”, a[++i]);
```

iに1が代入された後にprintf関数でa[1]が表示される。

```
i=0;
printf(“%d”, a[i++]);
```

printf関数でa[0]が表示された後にiに1が代入される

実行文

- プログラムの実行文は上から順番に一行ずつ実行される(重要)

```
int a, b, c;

a = 2;
b = 5;
c = a + b;
```

変数aに2を代入
変数bに5を代入
変数cにa + bを代入。この時点でaとbに値が入っているので
2+5が計算されて変数cに7が代入される

```
int a, b, c;

c = a + b;
a = 2;
b = 5;
```

順番を変えると結果が変わる。
cにa+bを代入。aとbに値がまだ入っていない。システム依存になるが
おそらくaにもbにも初期値の0が入っているのでcもおそらく0となる
aに2を代入
bに5を代入
cの値はその後変更していないので0のまま。

数学的には同じに見えるがプログラムでは結果は異なる。
プログラムを書くときは**右辺に使う変数に値が入っているかを確認**しながら書く

標準入出力

- 標準入力(キーボードからの入力)

```
scanf(“%d”, &var):          // 標準入力(キーボード)からの整数型の入力を読み込んで変数varに代入
scanf(“%lf %lf”, &var1, &var2); // 標準入力から倍精度実数型var1,var2に値を代入
```

入出力の型を明示。%d 整数型、%f 実数型　%lf 倍精度実数型(入力で使う)
scanfのときには読み込みたい変数名の前に&をつける(変数のアドレス)

- 標準出力(画面への出力)

```
printf(“%d”, var);                                  // 標準出力(画面)に整数型変数varの値を出力
printf(”var1 = %f,   var2 = %f¥n”, var1,var2);  // 標準出力(画面)にvar1,var2の値を出力
```

¥nは改行
シェル側でリダイレクトを使うことで標準入出力先をファイルなどに変更することができる
計算した結果はprintfまたはfprintfで必ず出力する

forループ

- 実行文は上から一行ずつ実行するのが原則だがよく使う例外は2つだけ
- for文：**for(式1(実行文); 式2(条件文); 式3(実行文))｛繰り返し文｝**
- 式1を実行(初期化)、式2の条件を判定、満たされていれば繰り返し文を実行
- 式3を実行してから式2の条件判定に戻る

```
sum = 0;                    // 変数sumに0を代入
for(i = 1;i<n; i++){        // iに1を代入して繰り返し文を実行、i<nの間iを1ずつ増やして繰り返す
    sum = sum + i;          // 変数sumの値をiだけ増やす
}                           //
                            //  i=nとなりi<nを満たさなくなるとforループから抜けて次の行へ。
```

$$\sum_{i=1}^{n} i$$ を計算していることになっている

forループ

増分値(ストライド)を1以外にする

```
sum = 0;
for( i = 1; i<n; i+=3){   // 増分が3になる。i=1の次はi=4, i =7, ...となる。
    sum = sum + i;
}
```

配列

- 配列とは：変数が複数並んだもの(メモリ上でも並んで配置される)。ベクトルや行列が表現できる。

配列の宣言の基本

```
double a[5], b[4][3];   // aは5つの要素を持つ倍精度実数変数の配列, bは4x3の二次元配列(行列)
```

←C言語では配列のラベルは0から始まる。
メモリ上にこのように5つの倍精度実数変数の領域が連続して確保される

配列の初期値設定

```
double a[5] = {0.,0.,0.,0.,0.};   // 配列aのすべての要素に初期値0.0を代入
int k[5] = {1, 2, 3, 4, 5};       // 配列kの各要素に初期値(順に1,2,3,4,5)を設定
```

実行文として設定してもよい(こちらのほうが自由度が高い)。**forループを使う**

```
for (i=0;i<5;i++){
    a[i] = 0.0;                  // 5次元配列のすべての要素(a[0]~a[4])に0.0を代入
}
```

異なる配列の違う添字を持つ要素の代入・演算も可能

```
for(i=0;i<5;i++){
    a[i] = b[i+3];    // a[0]=b[3], a[1]=b[4] ... のように代入される
}
```

1次元配列の計算例

ベクトル和

ベクトルの内積

```
double a[5], b[5], c[5];                    // a, b, cを5要素の配列(5次元ベクトル)として宣言
double adotb;                               // adotbとして倍精度実数型変数を宣言

// ここでaとbには値を代入しておく(省略)

for( i = 0; i<5; i++){
    c[i] = a[i] + b[i];                     // ベクトルcにベクトルaとbの和を代入
}
for( i = 0;i<5;i++){
    printf("c(%d)=%f¥n", i,  c[i]);         // ベクトルcの値を要素ごとに出力
     /* 1つ目の変数(i)は%dの場所に整数型として、2つ目の変数(c[i])は%fの場所に実数型として
        出力されるため c(0) = 1.00... のようになる */
}
adotb = 0.0;                                // 和を取るときは変数の初期化をする
for( i = 0; i<5; i++){
    adotb += a[i]*b[i];                     // adotbの値をa(i)*b(i)だけ増やす(内積)
}
printf("a dot b = %f¥n", adotb);            // adotbの値を出力
```

行列

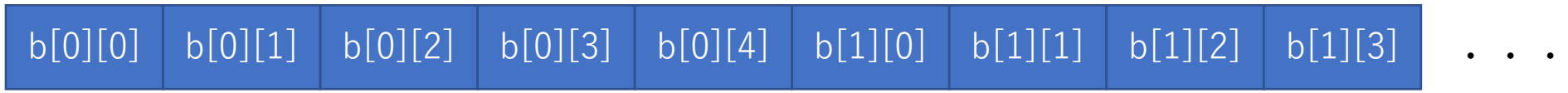
- 2次元配列で行列が表現できる

```
double b[5][5];

for(i=0;i<5;i++){
    for(j=0;j<5;j++){
        b[i][j] = 0.0;
    }
}
```

行列の初期化
二重ループ。この例では計算の順番は
i=0, j=0 → i=0, j=1 → i=0, j=2 → i=0, j=3 → i=0, j=4 →
i=1, j=0 → i=1, j=1 →　・・・
二重forループでは内側のループが先に変化する。

C言語での多次元配列は



の順でメモリ上に配置されるため(Fortranでは行・列が逆になる)forループは2つ目の
インデックスを先に回すほうがメモリ上で連続的にアクセスできてよい(キャッシュミスが少なくなる)

行列の和・値の出力

```
double a[5][5], b[5][5], c[5][5];     // 5x5の2次元配列(行列)としてa,b,cを宣言

/* a, bに値をここで代入(省略) */

for(i = 0; i<5; i++){
    for(j = 0; j<5; j++){
        c[i][j] = a[i][j] + b[i][j];        // cの各要素にa+bの値を代入
    }
}

for (i = 0; i<5; i++){
    for(j = 0; j<5; j++){
        printf(" %f ", c[i][j]);            // c[i][j]の値を出力。
    }
    printf("¥n");                           // 改行を出力
}
```

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

行列の積

一つの要素を計算するのにforループが必要

```
double amat[5][5], bmat[5][5], cmat[5][5];    // 5x5の行列を宣言

/* amatとbmatの値をここで代入 */

for(i = 0; i<5; i++){
    for(j = 0; j<5; j++){
        cmat[i][j] = 0.0;                              // cの値を初期化する
        for (k = 0; k<5; k++){
            cmat[i][j] += amat[i][k] * bmat[k][j];  // amatとbmatの積をcmatに代入。
        }
    }
}
```

配列のテクニック

- **宣言時に配列の大きさを変数に使いたい**
  →配列より手前で const修飾子をつけて配列の大きさに使う整数型を宣言。

```
const int  n = 100;   // const修飾子をつけるとプログラム中で値が変更できなくなる
double a[n], b[n], c[2*n][2*n];
```

- **配列の割付**：コンパイル時に配列の大きさを決めない(プログラム実行時に決める動的確保)

```
int n;
double *a;

scanf(”%d”, &n);                    // 例えば実行時に標準入力からnの値を読み込む
a = malloc(sizeof(double)*n);  // aを要素数nの倍精度実数型配列としてメモリに割り当てる
/* ここで計算をする(a[n]が使用可能) */
free(a);                            // aの配列をメモリから解放する
```

配列のテクニック

- **多次元配列の割付**

```
int n,m;
double *a;

scanf(”%d %d”, &n, &m);              // 標準入力からnとmの値を読み込む

a = malloc(sizeof(double)*n*m);  //  1次元配列として確保

for(i=0; i<n; i++){
    for( j=0; j<n; j++){
        a[i*m+j] =  ..... ;              // 添字を1次元化する。[i][j]成分は[i*m+j]として参照する
    }
}
free(a);                                    // aの配列をメモリから解放する
```

条件文

- 実行文は上から一行ずつ実行するのが原則だがよく使う例外は2つだけ
- if文：条件を論理式で指定し条件を満たすかどうかで処理内容を変える

```
int  i;                          // 整数型変数iを定義

scanf(”%d”, &i);          // iの値を標準入力から読み込む

if (i<0)
    i=-i;
                                 // iの値が負の場合はiに-iの値を代入

printf(“i=%d¥n”,i);     // iの値を標準出力に出力(入力した値の絶対値)
```

条件文

```
if (論理式) 実行文;

if (論理式){
   実行文1;
   実行文2;
}

if (論理式){
   実行文1;
}else{
   実行文2;
}

if (論理式1){
    実行文1;
}else if (論理式2){
    実行文2;
}else{
    実行文3;
}
```

論理式が真のときのみ実行文1と実行文2が実行される。(実行文が複数の時は{}で囲む)

論理式が真の場合は実行文1が実行され、偽の場合は実行文2が実行される

論理式1が真のときは実行文1を実行
論理式1が偽かつ論理式2が真の時は実行文2を実行
論理式1も論理式2も偽の場合は 実行文3を実行

else ifは複数あってもよい。

論理式

関係演算子	
a == b	aがbと等しい時に真
a != b	aがbと等しくないとき真
a >= b	aがb以上の時に真
a > b	aがbより大きい時に真
a <= b	aがb以下の時に真
a < b	aがbより小さい時に真

等しい時は=一つではなく2つなので注意

論理演算子	意味	使用例
!	(右)以外	! a==b
&&	かつ	a==b && b==c
||	もしくは	a==b || b==c
.eqv.	論理値が等しい	a<0 .eqv b<0 (aとbの符号が同じ)
.neqv.	論理値が異なる	a<0 .neqv. b<0 (aとbの符号が違う)

プログラムの終了

- プログラム(関数)は最終行まで到達すれば終了
- 条件文を使うと途中で終了させることも可能(return関数)
- 正常終了の場合は0を返す(mainがint型の場合)

```
#include<stdio.h>
#include<math.h>
int main(){
    double x;
    scanf("%lf", &x);                                   // xを標準出力から読み込む
    if( x<0.0){
        printf("x is negative¥n");
        return(-1);
    }                                                   // xが負であれば-1を返してプログラムを終了
                                                        // xがゼロか正であれば以下の処理が行われる
    printf("sqrt(x) = %f¥n", sqrt(x));
    return(0);                                          // mainプログラム終了
}
```

条件文の例：クロネッカーのデルタ

```
int delta, i, j;

scanf(“%d %d”,  &i, &j);

if (i == j){
    delta = 1;
}else{
    delta = 0;
}

printf(“delta=%d¥n”, delta);
```

```
int delta, i, j;

scanf(“%d %d”,  &i, &j);

delta = 0;
if(i==j) delta = 1;

printf(“delta = %d¥n”, delta);
```

forループとif文の組み合わせ

無限ループ, break

```
for(x=0.1; ; x+=0.1){  // 終了条件を指定しないと無限ループとなる
   // ここでいろいろ計算
   a = ....;
    if ( a<0 ) break;       // ある条件が満たされたらループから抜ける
}
```

break文でforループから抜ける。
条件が満たされない場合は計算が終わらないため実行してみて終わらない場合は強制終了(C-c)する

continue: forループの先頭まで戻り、カウンタを次の値に進める

```
for( i = 0;i<n;i++){

   a[i] = ....; // ここで何かを計算する

    if (a[i]>0) continue; //a[i]>0なら戻ってカウンタを一つ回す

    /* a[i]が0か負の場合だけこの領域で処理が行われる */

}
```

```
for( i = 0; i<n; i++){

    if( i == 7) continue; // iが7のときだけ何もしない

     // iが7以外のときは以下の計算をする。
     // ....

}
```

型変換

cast演算子を使う[(型名) 式]

整数型↔実数型

```
ai = (double) i;    // 整数型変数iを倍精度実数型に変換
ia = (int) a;       // 倍精度実数型変数aを整数型に変換
```

整数・実数型↔文字型

```
c = (char) i;       // 整数型変数iを文字型変数cに変換
i =  (int) c;       // 文字型変数cを整数型変数iに変換
```

複素数

複素数を使う時はcomplex.hをインクルードする。

値の代入

```
z = 2.0+I*3.0; // 複素数に特定の数値を代入。虚数単位はIとしてcomplex.hで定義されている
```

実部・虚部の取り出し

```
rez = creal(z)   // 複素数zの実部を倍精度実数型rezに代入
imz = cimag(z) // 複素数zの虚部を倍精度実数型imzに代入
```

組み込み関数

数学関数

関数	名称	数学式	関数	名称	数学式
log(x)	自然対数		log10(x)	常用対数	$\log_{10}(x)$
exp(x)	指数		sqrt(x)	平方根	$\sqrt{x}$
sin(x)	正弦		cos(x)	余弦	
tan(x)	正接		asin(x)	逆正弦	arcsin(x)
acos(x)	逆余弦	arccos(x)	atan(x)	逆正接	arctan(x)
atan2(y,x)	逆正接2	arctan(y/x)	sinh(x)	双曲線正弦	
cosh(x)	双曲線余弦		tanh(x)	双曲線正接	
fmin(x,y)	最小値	min(x,y)	fmax(x,y)	最大値	max(x,y)
pow(x,y)	累乗	x^y	fabs(x)	絶対値	$\|x\|$

角度の単位はラジアン (180° ＝ π rad)
数学関数はヘッダファイル math.hに含まれている

組み込み関数

複素数関数

関数	名称	数学式	関数	名称	数学式
creal(z)	実部	Re z	cimag(z)	虚部	Im z
conj(z)	複素共役	z*	carg(z)	偏角	Arg z
cexp(z)	指数	e^z	cabs(z)	絶対値	|z|
clog(z)	自然対数	log z	csqrt(z)	平方根	$\sqrt{z}$
csin(z)	正弦	sin(z)	ccos(z)	余弦	cos(z)
ctan(z)	正接	tan(z)	casin(z)	逆正弦	arcsin(z)
cacos(z)	逆余弦	arccos(z)	catan(z)	逆正接	arctan(z)

角度の単位はラジアン (180° ＝ π rad)
数学関数はヘッダファイル math.hに含まれている

関数

- 関数：与えられた入力変数に対して操作をして値を返す一連の手続き
- mainも関数。int型であれば整数値をreturnで返す。
- 例えば $4x^3-5x$を何度も様々なxに対して計算したいとする。
  プログラムのあちこちに$4x^3-5x$と繰り返して書くとミスのもとになる。
  また後から関数形に変更が必要になったときにすべて探し出して変更する必要がある。
- 関数として$f(x)=4x^3-5x$を計算する部分を独立させる。

- 関数を置く場所
  1. main関数の手前
  2. main関数の後ろ

- return文は関数の処理を中断してその段階の値を返す。

関数

例１：関数funcがmain関数より前に読み込まれる場合

```
#include<stdio.h>

double func(double x){
    double y;
    y = 4.0*x*x*x – 5.0*x;
    return(y);                    // 倍精度実数型変数yに入っている値を返す
}

int main(){
    double x;

    for(x=0.0;;){
    printf(“%f %f¥n”, x, func(x));  // ここで関数にxの値を渡して呼び出す
    x += 0.1;
    if( x > 5.0) break;
    }
}
```

関数

例2：関数funcがmain関数の後に読み込まれる場合

```
#include<stdio.h>

int main(){
    double x;
    double func(double);  // 関数プロトタイプ宣言。mainより手前でもよい。
    for(x=0.0;;){
    printf("%f %f¥n", x, func(x));  // ここで関数にxの値を渡して呼び出す
    x += 0.1;
    if( x > 5.0) break;
    }
}

double func(double x){
    double y;
    y = 4.0*x*x*x – 5.0*x;
    return(y);                    // 倍精度実数型変数yに入っている値を返す
}
```

再帰関数

- 再帰(recursion)とは：関数で自分自身を引用すること

フィボナッチ数列

```
int fibonacci(int n){
    int fibo;
    if( n<0 ) {
        fibo = -1;
        return(fibo);
    }
    if( n==0 ){
        fibo = 0;
    }else if(n == 1) {
        fibo = 1;
    } else {
        fibo = fibonacci(n-1) + fibonacci(n-2);  // 自分自身を呼び出す
    }
    return(fibo);
}
```

関数での複数の値・配列の受け渡し

関数の戻り値は一変数なので複数変数、あるいは配列の値を戻り値としてほしいときは
引数に**変数や配列のアドレス**を渡し、関数の中でその変数の値を更新する。

例：外積を計算

```
#include<stdio.h>
int main(){
    double a[3], b[3], c[3];
    void vectorproduct(double [3], double [3], double [3]); // voidは戻り値がないタイプの関数
    a[0] = 6.0; a[1] = 3.0; a[2] = 4.0;
    b[0] = 3.0; b[1] = -2.0; b[2] = -4.0;
    vectorproduct( a, b, c );                    // 配列のアドレスを渡す。(配列名が配列のアドレス)
    printf("%f %f %f ¥n", c[0], c[1], c[2]);
    return 0;
}
void vectorproduct(double a[3], double b[3], double c[3]){  // double *a, double *b, double *cでもよい
        c[0] = a[1]*b[2]-a[2]*b[1];
        c[1] = a[2]*b[0]-a[0]*b[2];
        c[2] = a[0]*b[1]-a[1]*b[0];
        return;                                  // 関数に戻り値はない
}
```

変数の共有

- 基本は引数にすべて書く。引数に書かれていないものは参照できない

```
#include<stdio.h>
double f(double x, double a, double b, double c){    // 2次関数を計算する関数
    return(a*x*x + b*x + c);
}

int main(){
  double  x0;
  x0 = 1.0;
  a = 1.0; b = 2.0; c = 3.0;
  printf("%f %f¥n", x0, f(x0,a,b,c));  // 関数にx0,a,b,cを渡す
  a = 2.0;
  printf("%f %f¥n", x0, f(x0,a,b,c));  // aの値が変わったのでf(x0,a,b,c)の値が変わる
}
```

関数fの独立性が高い。計算に必要な変数がすべて引数に入っているので
この関数だけを他のコードで使うことができる。ただし引数が煩雑になる。

変数の共有(グローバル変数)

- グローバル変数をmain関数とその他の関数で共有する。
- 関数はグローバル変数に依存するため独立性が低くなる
- 基本は引数として渡すのが間違いが少ない

```
#include<stdio.h>
double a, b, c;                    // グローバル変数。mainの外側で宣言する

double f(double x){                // 2次関数を計算する関数副プログラム
    return(a*x*x + b*x + c);       // a, b, cはグローバル変数の値を参照する
}

int main(){
  double  x0;
  x0 = 1.0;
  a = 1.0; b = 2.0; c = 3.0;
  printf("%f %f¥n", x0, f(x0));  // 関数のxとしてx0を渡す
  a = 2.0;
  printf("%f %f¥n", x0, f(x0));  // aの値が変わったのでf(x0)の値が変わる
}
```

ファイル入出力

標準入力： scanf(“%d”, &var); または fscanf(stdin, “%d”, &var);
標準出力： printf(“%d”, var); または fprintf(stdout, “%d”, var);
標準エラー出力： fprintf(stderr, “%d”, var);
stdin, stdout, stderrはstdio.hで標準入出力とエラー出力のストリームとして定義されている

シェル側でファイルに入出力

シェルでリダイレクトによって標準入出力をファイルに変更
Linuxのリダイレクトの部分を復習してください

```
./a.out < inputfile > outputfile     # inputfileから標準入力を読み込み、標準出力をoutputfileに出力
```

C言語でファイルを指定

```
FILE *fp1, *fp2;   /* FILE型ストリーム。stdio.hで定義されている */
fp1=fopen(‘inputfile.txt’, ‘r’);     // inputfile.txtを読み込み専用として開く
if( fp1==NULL){ // ファイルが開けなかった場合のエラー処理
    printf(“fp1 file open error¥n”);
    return(-1);
}
fp2= fopen(‘outputfile.txt’, ‘w’); // outputfile.txtを書き込み専用として開く。
fscanf(fp1, “%lf”, &var);  // fp1ストリームから倍精度実数varを読み込む
fprintf(fp2, “%f”, var);    // fp2ストリームに倍精度実数varを書き込む
fclose(fp1);  // ファイルを閉じる
fclose(fp2);  // ファイルを閉じる
```

ファイル入出力

fopen関数の読み書きのモード

r: 読み込み
w: 書き込み
a: 追加書き込み (ファイルが既に存在する場合はファイルの末尾に書き込む)

書式指定

- printf, fprintfで出力の書式指定ができる。

	サイズ指定なし	サイズ指定
整数	%d	%5d (5桁の整数)
実数	%f	%15.10f 全桁数15で小数点以下が10桁
実数(指数表示)	%e	%15.10e
文字	%c	
文字列	%s	%10s（10桁の文字列)

- scanf, fscanfでは単精度・倍精度実数も明示的に指定する

	サイズ指定なし
整数	%d
倍精度整数	%ld
実数	%f
倍精度実数	%lf

gccのコンパイルオプション

- -o ファイル名　：作成される実行ファイルの名前を指定
- -O0, -O1, -O2, -O3　：最適化オプション
  -O3が最も最適化されるが計算結果がおかしくなる可能性もある
- -lm : 数学関数を使う時 (数学ライブラリlibm.aを読み込む)
- -Wall ：全てのコンパイル時の警告メッセージを出力
- -Wuninitialized ：初期化されずに使われた変数を検出
- -pedantic ：標準外の機能利用を警告
- -c ：実行ファイルを作らずその手前のオブジェクトファイル(.o)のみ作成

デバッグ

- プログラムは一回で完全なものは書けない。間違い(バグ)が多くの場合ある。
- 数値計算を行う研究もかなりの時間はバグ取り(デバッグ)に費やされる
- コンパイル時にエラーが出て実行ファイルが作成されない
  - 警告が出ても実行ファイルが作成される場合もある(-Wallをつけている場合など)のでls –lで実行ファイルと更新日時をチェック
  - エラーメッセージを見てバグを順番に取り除く
- コンパイルは出来たが実行すると結果が明らかにおかしい
  - こちらは見つけるのが難しい
- **バグが見つけられない場合はどんどん質問してください。**

コンパイルエラーの例

エラーがどのファイルの何行目で出たかが出るので場所を確認する

test.c:8:3: error: use of undeclared identifier 'x0'

- test.cの8行目(3文字目)でエラー。
- ”x0”が使われているが型が宣言されていないエラー
- 変数の宣言し忘れ、変数名の打ち間違いはないか？

test.c:11:3: warning: array index 3 is past the end of the array (which contains 3 elements) [-Warray-bounds]
array[3] = 1.0;

WarningであってErrorではないので実行ファイルはできるがこの配列の次元1のインデックスは3要素(array[3] で宣言)しかないのに4要素目であるarray[3]を参照しようとしているという警告。このまま実行すると配列の領域外参照となる。

コンパイルエラーの例

```
test.c: In function ‘main’:
test.c:13:3: error: expected declaration or statement at end of input
   printf("%f %f¥n", x0, f(x0));
   ^
```

printfの行でエラーが出ているが実際のエラーは
main関数の最後の}忘れ。
エラーが常に示された行にあるとは限らない。

emacsではTabキーを押すと字下げをしてくれるので
{}の書き忘れや;の書き忘れを見つけることができる。

```
test.c: In function ‘main’:
test.c:10:3: error: expected ‘;’ before ‘a’
   a = 1.0; b = 2.0; c = 3.0;
   ^
```

a=1.0; の行でエラーが出ているが
(一つ手前の実行文での;抜け)

コンパイルエラーの例

```
error.c: In function ‘main’:
error.c:8:26: warning: implicit declaration of function ‘sqrt’ [-Wimplicit-function-declaration]
   printf("%f %f¥n", b/a, sqrt(-b));
                          ^
error.c:8:26: warning: incompatible implicit declaration of built-in function ‘sqrt’
error.c:8:26: note: include ‘<math.h>’ or provide a declaration of ‘sqrt’
/tmp/ccYKrlEq.o: 関数 `main' 内:
error.c:(.text+0x30): `sqrt' に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```

数学関数sqrtを使っているがmath.hをインクルードを忘れた場合

```
/tmp/ccttDieZ.o: 関数 `main' 内:
error.c:(.text+0x30): `sqrt' に対する定義されていない参照です
collect2: error: ld returned 1 exit status
```

math.hはインクルードしたがコンパイルオプションで –lm をつけ忘れた場合

数学関数を使う場合はヘッダファイルmath.hをインクルードして –lmをつけてコンパイルする

コンパイルエラーの例

```
test.c: In function ‘main’:
test.c:13:2: error: stray ‘¥343’ in program
    printf("%f %f¥n", x0, f(x0));
  ^
test.c:13:2: error: stray ‘¥200’ in program
```

printfの手前に全角スペースを混ぜた場合
有効ではない空白の文字が混ざっているとエラーがでる
(これは見つけにくい)

```
test.c:14:7: error: subscripted value is not an array, pointer, or vector
 d[1][2]= 0.0;
 ~~~~^~
```

１次元配列dを2次元配列のように２つインデックスを指定した場合
配列のランクが合わないとエラーがでる。

実行しながらのデバッグ

- コンパイルは通ったが結果がおかしい場合、特にどこかで出力が nan(not a number)またはinf(無限大)と表示される
- 負の数の平方根やゼロによる割り算などがおきていないか。
- ゼロで割っているつもりはなくても変数に値を入れ忘れると変数の値がゼロになっている場合がある。
- **printf関数で割り算が行われる変数などの値を割り算の直前で出力して直に確かめる。**

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

int main(){
  double a, b;
  a = 0.0;
  b = 10.0;
  printf(“%f %f¥n”, b/a, sqrt(-b));  // この結果は infおよび-nanとなる
}
```

実行しながらのデバッグ

- コンパイルは通ってnanやinfは出ないが計算結果がおかしい場合
- そのプログラムで答えが既知の問題は解けないか？
  - 使っている関数を簡単なものにしたときに正しい答えが出せるか？
  - 手計算で答えが求められる場合に正しい答えを出すか？
  - どの場合に結果が正しくてどの場合は結果がおかしくなるのかを調べてバグの場所を特定する。

実行しながらのデバッグ

- 書き直す時は削除せずに // でコメントアウトし、色々試すとよい。
- 宣言された配列の領域外に値を代入すると他の変数の値を破壊する可能性がある。
- 関数の引数の順番、数、型などが合っているか？整合性のチェックが入らないので実行時に結果がおかしくなる。
- 色々な所にprintf関数を入れてプログラムの動きをモニターする。
  - 特にforループ, if文などが想定したとおりに動いているかをチェックする。forループではカウンタの値を毎回出力して想定通りに動いているか？if文では分岐それぞれにprintf関数を入れて想定通りの分岐に入っているか？
  - scanf,scanf文で変数・配列に読み込んだデータは正しく読み込めているか？printfでチェックする。
  - 関数に渡す前の引数の値、関数に渡った直後の引数の値、関数実行後の引数の値など想定通りになっているか？
- コンパイルオプションを増やすと情報が得られる可能性がある。