C 言語の文法まとめ

ver. 2019/05/10

C 言語の基本事項

大文字と小文字は区別される。通常小文字で書く。

文の終わりには;を入れる

/*　　*/で挟まれた領域や//で始まる文はコメント文としてコンパイル時には無視される。

一行が長くなって複数行にまたがる場合は;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;
```

○プログラムの基本構造

```
#include<ヘッダファイル名.h>
int main(){
        宣言文（変数宣言など）
        実行文
}
```

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)

型

変数型	記述方法
整数型	`int`

単精度実数型	float	
倍精度実数型	double	
複素数型	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	

変数名：英数字とアンダーラインが使える。英字かアンダーラインから始まる名前。

変数の宣言例

```
int a;
double b,c;
double d = 0.0;  /* 初期値入力 */
double complex dt1;
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 */
```

文字型変数の宣言例

```
char a = "a";  /* 長さ 1 文字の文字列 */
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 */
```

変数型について

整数型：4 バイト(32 ビット)，2 進数でメモリに格納。最大値 2^{31}-1 まで
単精度実数型：4 バイト、10^{-38}~10^{38}，約 7 桁の精度(符号、指数と仮数部に格納)
倍精度実数型：8 バイト、10^{-308}~10^{308} 約 16 桁の精度
単精度複素数型：8 バイト、実部と虚部が単精度実数型
倍精度複素数型：16 バイト、実部と虚部が倍精度実数型
論理型： true または false

物理の計算では実数と複素数型では基本的に倍精度のものを使う。

○四則演算

(+, -,*, /)、累乗はない。(*で何度もかけるか、数学関数の pow を使う)

同じ型の演算は同じ型となる。(整数型/整数型は整数型)

```
a = 5 / 2;   /* a は 2 となる。 2.5 にしたい場合は 5.0/2.0 とする */
```

違う形の演算は型変換が行われる(整数と実数の演算は実数に、実数と複素数の演算は複素数に)

```
a = 4*4*4;        /* 4 の三乗 */
```

```
a = pow(4.0,3.0);  /* 4 の三乗、入力も出力も倍精度実数型 */
```

○代入文

変数に値を代入するには=を使う(数学の等号ではない)

```
a = 5 / 2.0;    /* 倍精度実数型の a に 5/2.0 の計算結果である 2.5 を代入 */
i = i + 5;      /* i を 5 増やす。(再帰代入) */
i += 5;         /* i=i+5 をこのように省略してかける */
```

○インクリメント・デクリメント演算

i=i+1; は i++; あるいは ++i; とかける。
i=i-1; は i--; あるいは --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。

基本例：
```
i=0;
printf("%d", a[++i]);
```
とすると i に 1 が代入され、a[1]が表示される。

後置形; i++; i--;

基本例：
```
i=0;
printf("%d", a[i++]);
```
とすると a[0]が表示された後 i に 1 が代入される

○for 文の使い方

for(式 1; 式 2; 式 3) { 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

繰り返し実行文を実行、式 3 を実行してから式 2 の条件判定に戻る

基本例１（sum（初期値 0）に N 回 1 を足す）

```
sum = 0;
for( i=0; i<N; i++){
   sum = sum + 1;
}
```

基本例２（sum（初期値 0）に i が 1～N まで増える間、3 回ごとに i を足す）

```
sum = 0;
for(i=0;i<N;i+=3){
   sum = sum + i;
}
```

補足説明： インクリメント

for 文のカウンタはこの場合は 3 つずつ増えて行く。つまり、i=1, 4, 7, 10, …の様に増えて行く事に注意。

○標準入出力(scanf, printf 文の使い方)

基本例 1（コマンドラインから hoge を読んで、hoge をディスプレーに出力）

```
scanf("%d", &hoge);
printf("%d", hoge);
```

補足説明： 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 のときは読み込みたい変数名の前に&をつける(変数のアドレス)

printf 文で改行は¥n

基本例 2(コマンドラインから倍精度実数 hoge1 と hoge2 を読んで両方を出力

```
scanf("%lf %lf", &hoge1, &hoge2);
printf("hoge1 = %f    hoge2 = %f\n", hoge1, hoge2);
```

○配列（ベクトル）の扱い方

型宣言

基本例

```
double a[5], b[4][3]; /* a は 5 つの要素を持つ配列、b は 4x3 の行列 */
```

配列のラベルは 0 から始まる。a[5]として定義されている場合、5 つの要素は a[0], a[1], a[2], a[3], a[4]となる。

配列を定義

初期値設定

```
double l[5]={0.,0.,0.,0.,0.}; /* 配列 l の成分がすべて 0 */
int l2[5] = {1,2,3,4,5}; /* 配列 l2 の各要素の初期値を設定 */
```

基本例（n 次元配列 a_iの全要素に 0 を代入）

```
for(i=0; i<n; i++){
        a[i] = 0.0;
}
```

補足説明：当然、0 以外の要素の代入・演算も可能

また、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<n; i++){
        a[i] = b[i+3];
}
```

この時、b の配列の大きさは n+3 より大きくないと不可

宣言時に配列の大きさに変数を使いたい場合は const 修飾子を使う

```
const int n=100;
double a[n], b[n], c[n];
```

コンパイル時に配列の大きさが決まらない場合(動的確保)

```
int n;
double *a;

scanf("%d", &n); /* 例えば実行時に配列のサイズとなる整数型を読み込む
 */
a = malloc(sizeof(double)*n); /* a にサイズ n の実数型配列を確保する */
/* ここで計算をする */
free(a); /*  a の配列をメモリから開放する */
```

多次元配列の場合

```
int n,m;
double *a;
scanf ("%d %d", &n, &m);

a = malloc(sizeof(double) * n * m); /* 一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] = .....;  /* 添え字を 1 次元化する。[i][j]成分は
[i*m+j]として参照する */
        }
}
free(a);
```

○行列の扱い方

基本例（n×n 行列 b_{ij}の全要素に 0 を代入）

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

```
}
```

補足説明： 2 重ループの構造上、内側のループが優先的に変化する。

C 言語では多次元配列は

b[0][0], b[0][1], b[0][2], ... b[0,n-1], b[1][0], b[1][1], b[1][2], ...

の順番にメモリに格納される (Fortran では逆)。

この順に演算をしたほうがキャッシュミスが少なくなる。

○型変換

cast 演算子を使う[(型名) 式]。整数の実数化、実数の整数化

基本例（i の値を ai と実数化、a の値を ia と整数化）

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

複素数の実部、虚部取り出し (creal, cimag)

基本例

```
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 */
```

２つの実数を実部と虚部とする複素数を作る

```
z = 2.0+I*1.0;   /*複素数に特定の数値を代入( z に 2.0+i を代入) */
```

虚数単位は I として complex.h で定義されている

○組み込み関数(倍精度実数用)

数学関数

math.h	名称	数学的	math.h	名称	数学的
log(x)	自然対数	$\log_e(x)$	log10(x)	常用対数	$\log_{10}(x)$
exp(x)	指数	e^x	sqrt(x)	平方根	$\sqrt{x}$
sin(x)	正弦	$\sin(x)$	cos(x)	余弦	$\cos(x)$
tan(x)	正接	$\tan(x)$	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	$\sinh(x)$
cosh(x)	双曲線余弦	$\cosh(x)$	tanh(x)	双曲線正接	$\tanh(x)$

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	$\|x\|$

複素数関数

FORTRAN式	名称	数学式	FORTRAN式	名称	数学式
creal(z)	実部	$\mathrm{Re}\, z$	cimag(z)	虚部	$\mathrm{Im}\, z$
conj(z)	複素共役	z^*	carg(z)	偏角	$\mathrm{Arg}\, z$
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	$\arctan(z)$

補足説明：角度の単位はラジアン単位なので注意（180°=πラジアン）

○if 文の使い方

基本例１（i の値が 0 の時は 0 を、１以上の時は 1 を、それ以外（つまり負の値）の時は-1 を加える）

```
if (i==0){
   i = i;
}else if (i >= 1){
   i++;
}else{
   i--;
}
```

補足説明：　関係演算子

A==B　　:A が B と等しい時（A=B）に真
A!=B　　:A が B と等くない時（A≠B）に真
A>=B　　:A が B 以上の時（A≧B）に真
A>B　　:A が B より大きい時（A>B）に真
A<=B　　:A が B 以下の時（A≦B）に真
A<B　　:A が B より小さい時（A<B）に真

○if 文による for 文ループの制御

基本例 1　for 文で終了条件を書かない (break でループを抜ける)

```
for(x=0.1; ; x+=0.1){
        y = x*x;
        printf("%f %f\n", x, y);
        if ( x > 1.0)
                break;  /* x>1 なら for ループから出る */
}
```

条件が満たされないとプログラムは終了しないので端末から Ctrl-C で強制終了する。

基本例 2 特定の条件のときはそれより下の for 文内を実行しない(continue で for ループの頭に戻る)

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

        if ( i == 7)
                continue; /*  i が 7 のときは戻って i=8 の処理に移る。*/
        /* ここに i が 7 以外のときの計算を書く */

}
```

○論理演算

if 文の中で、比較演算子や論理型の変数と共に用いられる。

論理演算子	意味 & 意味	
!	以外	! a==b
&&	かつ	a==b && b==c
||	もしくは	a==b || b==c

例：

```
if(a==B && b>=c)                    ：a と b が等しく、b が c より大きい時真
if(!(a==b))                 ：a と b が等し無い時真(a!=b と同じ)
if((a==b) == (b>=c))        ：a と b が等しく、b が c より大きい時真、また
                              a と b が等しく無く、b が c 以下の時真
```

○関数

関数を定義することで組み込み関数と同じように使うことができる。

関数の値は return 文で返す。

基本例 1： $f(x) = 4x^3 - 5x$ を関数 func として定義して main 関数から呼び出す

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);
}

int main(){
  double x;
  for(;;){
    printf("%f %f\n", x, func(x));
    x +=0.1;
    if(x>5.0) break;
  }
}
```

基本例 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(;;){
    printf("%f %f\n", x, func(x));
    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);
}
```

○再帰

関数で自分自身を引用することを再帰という。漸化式の計算などに使える。

基本例(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);
        }
        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。

基本例：外積を計算

```
#include<stdio.h>

int main(){
  double a[3], b[3], c[3];
  void outerproduct(double [3], double [3], double [3]);
  a[0]= 6.0; a[1] = 3.0; a[2] = 4.0;
  b[0]= 3.0; b[1] =-2.0; b[2] =-4.0;
  outerproduct(a, b, c);                /*配列のアドレスを渡す */
  printf("%f %f %f\n", c[0], c[1], c[2]);

  return 0;
}

void outerproduct(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];
}
```

○ファイル入出力

標準入力：端末上でのキーボードからの入力

scanf などで標準入力を変数へ読み込める。

標準出力：ディスプレイ上の端末への出力

printf で変数を標準出力に表示できる。

標準エラー出力：ディスプレイ上への端末へ出力されるが標準出力とは区別される。

`fprintf(stderr, ...)`で変数を標準エラー出力に表示できる。

UNIX 上で標準入出力をファイル入出力に変更する：リダイレクト

> 標準出力への出力結果をファイルに保存。a.out が実行ファイルとすると

```
./a.out > outputfile
```

outputfile に./a.out のプログラム実行による標準出力への出力結果を上書き保存

>> すでに存在するファイルに上書きではなく追記したい場合

```
./a.out >> outputfile
```

outputfile の末尾に./a.out のプログラム実行による標準出力への出力結果を追記保存

2> 標準エラー出力への出力結果をファイルに上書き保存。

```
./a.out 2> errorfile
```

標準出力とエラー出力を異なるファイルに上書き保存する場合は

```
./a.out > outputfile 2> errorfile
```

同じファイルに保存する場合は (bash の場合)

`./a.out >& outputfile` または `./a.out &> outputfile` など

ファイルの値を標準入力にわたす場合

```
./a.out < inputfile
```

inputfile の内容が標準入力に渡される。

C で入出力ファイルを指定する

基本例（"ファイル名 1 を読み込み専用で開き、ファイル名 2 を書き込み専用で開き、それぞれのファイルにデータを入出力する）

```
FILE *fp1, *fp2;  /* FILE 型ストリーム、stdio.h で定義されている */
fp1=fopen("ファイル名 1", "r");  /* ファイルを開く */
if (fp1==NULL){  /* ファイルが開けなかった場合エラー処理 */
        printf("fp1 file open error\n");
        return(-1);
}
fp2=fopen("ファイル名 2", "w");  /* ファイルを開く */
if (fp2==NULL){  /* ファイルが開けなかった場合エラー処理 */
        printf("fp2 file open error\n");
        return(-1);
}

fscanf(fp1, "%lf", &hoge);  /* fp1 ストリームから倍精度実数 hoge を読み込む*/
fprintf(fp2, "%f\n", hoge);  /* fp2 ストリームに倍精度実数 hoge を書き込む */

fclose(fp1); /*ファイルを閉じる */
fclose(fp2);
```

補足説明：

fopen 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(ファイルが存在する場合ファイルの末尾に書き込む)

○書式 (printf, fprintf)

基本例

	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 10 桁小数点以下)

文字	%c	
文字列	%s	%10s (10 桁の文字列)

scanf, fscanf で読み込むときは

整数	%d
倍精度整数	%ld
単精度実数	%f
倍精度実数	%lf

○ポインタ

準備中

○コンパイル

```
$ gcc プログラムソースファイル.c  -o 実行ファイル名
```

コンパイルオプション

-O0, -O1, -O2, -O3 : 最適化オプション

-lm 数学関数を使うとき

-Wall : 全てのコンパイル時の警告メッセージを出力

-Wuninitialized : 初期化されていない変数を検出

-pedantic : 標準外の機能利用を警告

○デバッグ

文法以前のチェック事項

① 変数の打ち間違いは無いか？ ；の打ち間違えで：や ，を使っていないか？

② scanf を使うときは読み込む変数の前に&をつけているか？書式指定は正しいか？

③ 配列の大きさが、定義された範囲を超えていないか？

④ for 文や if 文などがきちんと閉じているか？

⑤ emacs では正しくコーディングされているとタブを押したときに各行が整列する。

実践的なデバッグの方法

① 色々な所に printf 文を挿入し、途中結果、for 文の変数の増え方等をモニターする。

② /*　*/ でコメント化することでソースコードの保存や実行を飛ばすことも可能。

③ for ループの変数がちゃんと受け渡されているかどうかチェックする。特に、配列の大きさを超える変数が受け渡されると、他のデータを破壊してしまうので要注意。

④ 配列にきちんとデータが読み込まれているかチェック。

⑤ 検索コマンドで、同じ変数がちゃんと検索されるかどうかチェック。

⑥ コンパイル時にデバッグオプションを追加