情報工学実験第Ｉ
Ｃプログラミング（2005 年度後期）

［時間］（後期）木曜１～２時限　　［担当］甲斐

Ｉ．オリエンテーション

１．目的

システム開発のためのＣプログラミング手法の習得を目的とする。

２．授業形式

これまでに、Ｃ＋＋言語を習得してきたが、システム開発にはＣ言語のポインター、構造体の知識があった方が、より複雑な処理が可能となる。本授業では、情報処理センターを使用し、実習を兼ねて行う。

講義内容スケジュール

日　　程	講義内容予定
９月２２日	オリエンテーション（７－３０１）
	入出力について
２９日	一次元配列とポインタ変数
	二次元配列とポインタ変数
１０月　６日	関数の復習（Call by value），再帰関数その１
	関数（Call by reference）,再帰関数その２
１３日	コマンドラインの取得とファイル入出力関数
	構造体の基礎
２０日	構造体配列
	リスト構造
２７日	２進木と構造体（または総復習）

Ⅱ．Ｃプログラミング

１．Ｃ言語の特徴

(1)英小文字ベースである

(2)プログラムは、LISP や APL と同じく、関数（モジュール）の集まりとして構成する。
　　関数の再帰呼び出しはもちろん可能である。関数は独立にコンパイルして良い。

(3)データとしてアドレス・ポインタが使える。

(4)演算子の数が非常に多い。
　剰余（%），インクリメント（++），デクリメント（−），ビット毎の演算（&，|，・），
　論理演算（!，&&，||），シフト（>>，<<），　　自己演算（+=，−=，*=，/=他）など

(5)文字列処理のための特別な機能はないが、
　文字列（"string"のように書く）と１文字（'c'）の区別、
　文字列や配列の終わり（¥0）
　復帰改行記号（¥n）
　ファイル終端（EOF）記号（0）
　エラー記号（−1）
　　などがプログラムで識別できる。

(6)式や関数が値を持つことから、次のような表現ができる。
　　a=b=c=d=0;
　　while((c=getchar( )) != EOF) {……}

(7)言語仕様上は入出力の機能はなく、入出力はすべて関数で行なう。

(8) 近代的な流れ制御機構がある。
　if( ) ～ else ; for( ) ～;
　do ～ while( ); switch ～ case ～ default;　など

(9) プリプロセッサがあり、次のようなマクロ機構が使える。
　#define SIZE 100
　#include "filename"

(10)表現が簡潔で、タイピングの量が少なくてすむ。Pascal との比較の例を次に示す。

Ｃ言語の表記法	ＰＡＳＣＡＬの表記法

{……}
a = 123;
count++; または　++count;
price *= 1.03;
x ? a : b;
int a[ ], *p, f( );
	begin …… end
	a := 123;
	count := count + 1;
	price := price * 1.03;
	if x then a else b;
	var a:array [……] of integer;
	　　p: ↑ integer;
	　　function f( ): integer;

(11)Ｃプログラムに引数が渡せる。
　/home/usr9/ut994999%　example　13　15

２．Ｃプログラムの形式

プログラム例：　２数を入力し、その加減乗除の演算結果を表示する。

```
#include <stdio.h>        /* C++の<iostrean>と同等 */
                          /*  コメントは //ではなく、このようにする */
int main( )
{
    int i, j, wa, sa, seki, syo;

    scanf("%d%d", &i, &j);
    wa = i + j; sa = i - j;
    seki = i * j; syo = i / j;
    printf("i + j = %d¥n", wa);
    printf("i - j = %d¥n", sa);
    printf("i * j = %d¥n", seki);
    printf("i / j = %d¥n", syo);
}
```

３．Ｃ言語の規約

3.1 文字セット

・ＡＳＣＩＩ文字セットに従う。

3.2 識別子（変数名、関数名など）

・識別子として使えるのは、

英小文字（ａ～ｚ），英大文字（Ａ～Ｚ），数字（０～９），アンダースコア（＿）の任意の組合せ。ただし先頭の文字は数字であってはならない。

・識別子の長さは通常８文字まで認識する。（オプションで３９文字まで）

3.3 空白

・空白とは、スペース，タブ，改行，およびコメントをいう。

・コメントは、／＊ではじまり、＊／で終わる。

・空白はどこにどれだけあっても無視されるので、プログラムが読み易くなるように適当に使用できる。

3.4 予約語

・以下の語は予約語であるので、ユーザー定義の識別子として使用してはならない。

データ型	char double enum float int long short struct union unsigned void
記憶クラス	auto extern register static typedef
制御文	break case continue default do else entry for goto if return switch while
演算子	sizeof
キーワード	const far near pascal

3.5 基本データ型

型	サイズ	値の範囲
char	8 ビット	-128 ～ 127
int	16 ビット	-32768 ～ 32767
short	16 ビット	-32768 ～ 32767
long	32 ビット	-2147483648 ～ 2147483647
float	32 ビット	約10^{-37}～10^{38}
double	64 ビット	約10^{-307}～10^{308}
unsigned char	8 ビット	0 ～ 255
unsigned int	16 ビット	0 ～ 65535
unsigned short	16 ビット	0 ～ 65535
unsigned long	32 ビット	0 ～ 4294967295

４．コンパイル方法

```
gcc  aaa.c  -o  aaa
```

Ⅲ．入出力

scanf()と printf()

Ｃ言語では、(cin、cout に対する）>>, << の代わりに scanf( ) 、printf( ) を使用する。

出力：printf 関数 printf(書式 , 変数 , 変数 , ・・・)　　iomanip は使えません。

例：　　出力

```
int   sum=50, a=10, b=20, c=30;
double   average=38.62;
printf("abcdef");                          abcdef
printf("総和は  %d  です。", sum);          総和は  50  です。
printf("平均は  %lf  です。", average);     平均は  38.62  です。
printf("%d  +  %d  =  %d¥n", a, b, c);     10  +  20  =  30        改行が入る。
```

（資料では¥ですが、UNIX 上では＼で出ます。）

型	書式	値の範囲
char	%c　%s	-128 ～ 127
int	%d	-32768 ～ 32767
short	%d	-32768 ～ 32767
long	%ld	-2147483648 ～ 2147483647
float	%f	約１０$^{-37}$～１０^{38}
double	%lf	約１０$^{-307}$～１０^{308}

フィールドを指定して出力する。

```
printf("総和は  %8d  平均は  %10.2lf  です。", sum, average);
```

出力：総和は　　50　平均は　　38.62　です。　　桁数の方が少ない場合は数値優先

8 桁　　10 桁、小数２桁　右詰めになります。(左詰めの場合はマイナスを付ける)

８進数での出力（%o）１６進での出力（%x）も用意されている。

残りをゼロで埋める。printf("%05d",data);　データが 123 なら、00123 と出力される。

入力：scanf 関数　scanf(書式 , 変数 , 変数 , ・・・)

例：

```
int   data;
scanf("%d",&data);          /* 注意！　変数名の前に&を付ける。 */
                            /* 書式は、間を空けたり、別の文字を入れないこと */
```

・EOF の処理は、以下のようにします。(.eof( ) は使用できません。)

```
while( scanf("%d", &data)!=EOF){  ・・・  }
```

EOF は通常－１の値をとります。従って、EOF の値を保存しておくには int にする。

```
int   eofval;
eofval = scanf("%d", &data)!=EOF
```

ヘッダーファイルとしてＣ＋＋では、#include <iostream>でしたが、Ｃでは、#include <stdio.h>とすること。

本授業のディレクトリーとして「cprogram」を作成し、その中で全ての作業をすること。

ディレクトリーの作成：mkdir　cprogram

ディレクトリーへの移動：cd cprogram

［演習１］

キーボードから、学籍番号の数字部、年齢、身長、体重を読込み、出力するプログラムを作成せよ。

なお、入力、出力に際しては、その体裁を整えること。

［演習２］

複数の正の整数値を**整数**変数に読み込み、最大値、最小値、平均を出力するプログラムを作成せよ。
なお、データの終了には－１を使用すること。（－１が処理に入らないように注意すること）
（終了は EOF ではありません！）

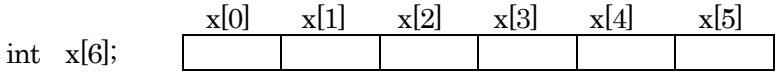
［実行例］チェックデータ（必ずこのデータを使用して提出すること）

データを入力して下さい。データの終了は－１
1
2
4
5
6
-1
最大値は、6 です。
最小値は、1 です。
平均値は、3.600000 です。

注意！平均値が 3.0 にならないように。

Ⅳ．配列

配列は、同じ属性を持つ連続した領域のことです。



注意：宣言では、[　]の中は個数ですが、実行部で使用する時は、0 から始まる。従って、最後は個数－１

配列の宣言は、次のようにします。

```
int     arr1[100], arr2[100], sum1 = sum2 = 0, kosu, i, arr3[ ] = {1,2,3};
double  ave[2];

printf("データ数を入力せよ");
scanf("%d", &kosu);       /*  個数の読込み */

printf("データを１組（２個）づつ入力せよ");
for( i=0; i<kosu; i++ )   scanf("%d%d", &arr1[i], &arr2[i]);   /*  それぞれのデータの読込み */
for( i=0; i<kosu; i++ ){                                       /*  ループは i=0  から始める方がいいでしょう */
    sum1 += arr1[i];      /*  データ１の総和 */
    sum2 += arr2[i];      /*  データ２の総和 */
}
ave[1] = (double)sum1 / kosu;       /*  データ１の平均 */
ave[2] = (double)sum2 / kosu;       /*  データ２の平均 */

for( i=0; i<2 ; i++)   printf("%d 個目の平均は%lf    ",ave[i]);
printf("¥n");
```

arr3 の配列は、初期値を入れています。通常は[ ]の中には個数を入れません。
（自動的に３個となる。　従って、使用する時は、arr[0]～arr[2]までで、arr3[3]以降は使えない）
大きく配列を宣言したい時は、arr3[10] = {1,2,3}; とすること。

注意：配列は、宣言した大きさ以上の部分は使えません。例えば arr[200]=1;はダメですが、コンパイルエラーとしては出てこないので、注意が必要です。特に、変数に値を入れて使用する場合には気を付けて下さい。
上のプログラムでは、kosu として 101 以上の数値を入れてしまうとアウトです。
（うまく行く場合もありますが、通常は、実行時にコアファイルができるか、ハングアップしてしまいます。）

文字の取り扱い

文字は、char を使用する。
１文字読込みは、getchar( )関数を使用する。１文字出力は、putchar(char)関数を使用する。
改行（¥n）やタブ（¥t）なども１文字として取り扱う。

なお、getchar( )を受ける変数は char でもよさそうなのですが、一般的には int で受ける。
（EOF が内部表現上、－１であることが多いため）（配列使用時には注意）

```
int   ch1;
char  ch2='a';      /* ch2 は初期値として文字 a を設定している。１文字はシングルクォーテーションで囲む */
ch1 = getchar( );          /*  文字変数 ch1 に１文字読込む */
putchar(ch1);  /*  文字変数 ch1 から標準出力に書き出す */
```

また、scanf( )でも１文字が読める。
（%c 使用、ただし、他の書式（%d など）と混合して使用する時は、かなりの注意が必要）

```
char  chx;
scanf("%c", &chx);
```

EOF をチェックする場合は、以下のようになる。

```
int  ch1;
while( (ch1=getchar( )) != EOF )  putchar(ch1);      /* 読込んだ文字をそのまま画面出力する（改行も含めて） */
```

または、

```
while(1){
    ch1 = getchar( );
    if( ch1 == EOF) break;          /* この場合は、int ch1; なければなりません。char ではできない */
    putchar(ch1);
}
```

文字列の取り扱い

C++と異なり、string（クラス）が使用できない。従って、.eof( )も使えません。
文字列の取り扱いは、文字の１次元配列として使用します。

文字列は最後に必ず ‘¥0’ を付けることになっています。
例えば、char cha[ ]=”abcde”;は、

	cha[0]	cha [1]	cha [2]	cha [3]	cha [4]	cha [5]
char cha[ ];	‘a’	‘b’	‘c’	‘d’	‘e’	‘¥0’

となる。

文字がシングルクォーテーションで囲むのに対し、文字列はダブルクォーテーションで囲む。
‘a’ と ”a” では異なる。”a”は文字列なので、配列表現となる。（”a” は、{’a’, ‘¥0”} と２個の配列）

文字列の読込み

文字列の読込みには以下の２通りがある。

①scanf を使用する方法

```
char  chx[100];
printf(“文字列を入力して下さい : ”);
scanf(“%s”, chx);          /* 注意 : 文字列の読込みには&が付かないので気を付ける */
printf(“読込んだ文字列は ¥”%s¥” です。¥n”, chx);
```

②getchar を使用する方法

```
char  chx[100];
int  i, ch;
printf(“文字列を入力して下さい（最後はスペースを入力）: ”);
for(i=0; (ch = getchar( )) !=’ ‘; i++) chx[i] = ch;  // ‘ ‘ はスペースが間に入っている。読込の最後にスペースを入れる
chx[i] = ‘¥0’;
printf(“読込んだ文字列は ¥”%s¥” です。¥n”, chx);
```

なお、文字列は　最後に’¥0’を１文字使用するので、実際に使用できる文字列は配列の大きさより１つ減る。

注意 : ②のプログラムで次のようにするとダメです。理由は、char が８ビットで int が 16 ビット（または 32）だから。
なぜだか分かりますか？

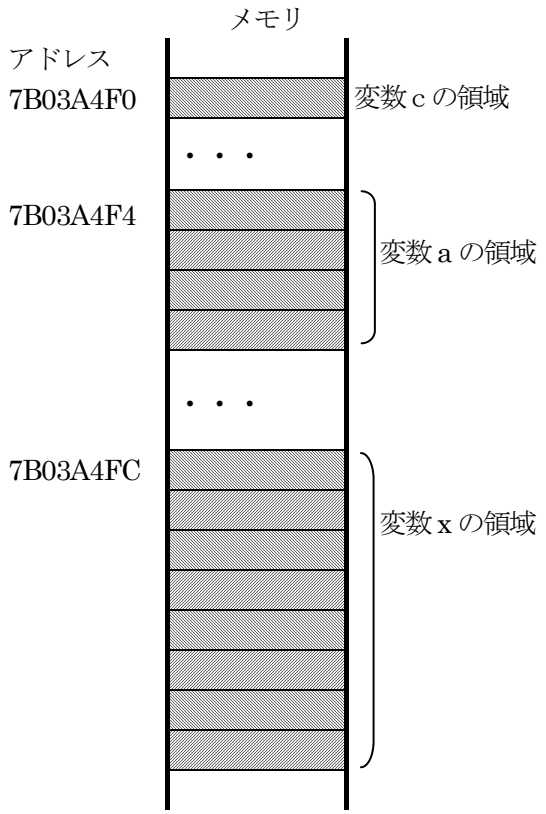
```
int  chx[100];
int  i, ch;
printf(“文字列を入力して下さい（最後はスペースを入力）: ”);
for(i=0; (ch = getchar( )) !=’ ‘; i++) chx[i] = ch;
chx[i] = ‘¥0’;
printf(“読込んだ文字列は ¥”%s¥” です。¥n”, chx);
```

Ⅴアドレスとポインタ

◎アドレスとは・・・・・

下のように変数を宣言すると、その変数に合わせて必要な領域がメモリ中に右図のように確保される。

```
char    c;
int     a;
double  x;
```



変数が配置されるメモリ中のアドレスは、システムにより自動的に割り当てられる。通常、変数がどのアドレスに配置されたかを知る必要はあまりないが、アドレス演算子「&」を用いるとそれを知ることができる。以下の簡単なプログラムがそれを示している。

```
#include <stdio.h>
int main( )
{
    int x;
    x = 100;
    printf("Content in   %p   is   %d.¥n", &x, x);
}
```

実行結果の例
Content in　7b03a4f4　is　100.

【メモ】出力書式 %p について
%p の書式は、ポインタ変換であり、void へのポインタ（あらゆる型へのポインタ）の値（つまりアドレス）を表示させるためのもので、変換形式は処理系に依存するが、一般に %x に近い形となる。また処理系によっては%P と指定すると、%X と同様の表示をするものもある。

このアドレス演算子はすでに以下のような所で使われていた。

```
int x, y;
scanf("%d%d", &x, &y);
```

すなわち、scanf 関数は読み込む変数のアドレスを指定して、そこへ値を格納する関数であるということに注意せよ。
次に以下の例を考える。

```
char s[10];

scanf("%s", s);
```

さて、ここで配列変数 s に文字列を読み込むのに、なぜ s となっていて &s ではないのだろうか？（配列とポインタの関係の所で明確になる。）

◎ポインタ変数

ポインタ変数は、アドレスを格納するための変数である。
単にアドレスが与えられても、どんな型のデータ（int 型，char 型，float 型など）が入っているアドレスなのかが分からないので、格納されるデータ型に合わせた宣言を以下のように行う。

【ポインタ変数を用いた簡単なプログラム】

```
#include <stdio.h>
int main( )
{
    int x, y, *p;

    x = 10;
    p = &x;                 /* ←x のアドレスをポインタ変数 p に入れる。*/
    y = *p + 100;           /* ←*p で p に格納されている内容、すなわち x の値 */
    p = y - 50;             /* p = y － 50 */
    x = x / 2;

    printf("x=%d,  y=%d¥n", x, y);
    printf("Adr. of x:%p,  p=%p¥n", &x, p);
}
```

実行結果の例
x=30, y=110
Adr. of x:1ec0, p=1ec0

ポインタ変数とアドレス及び格納値の関係

格納値	アドレス
x	＆x
＊p	p

［演習３］（７ページ４２行目）
文字列の読込みの部分の注意：について考えて、考察せよ。

［演習４］以下のような宣言、

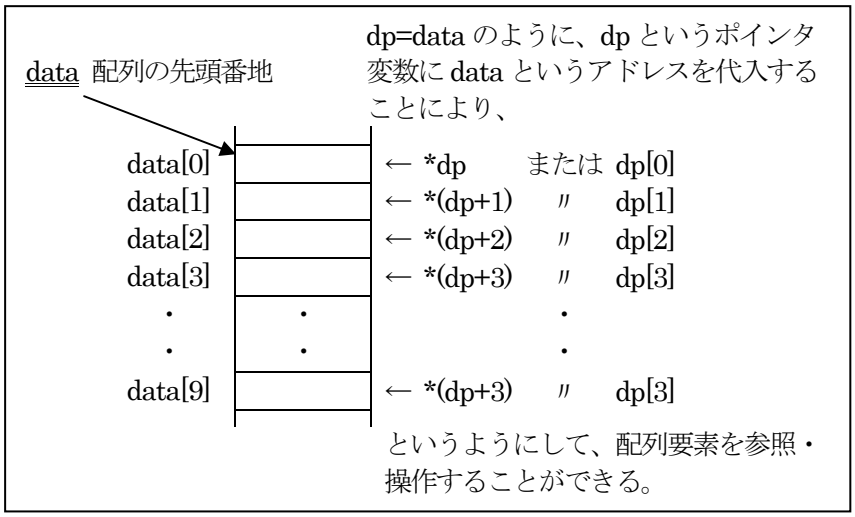
```
char   c1;
int    i1;
char   c2,  c3;
int    i2;
double d1;
int    i3;
```

で、ｃ１，ｉ１，ｃ２，ｃ３，ｉ２，ｄ１，ｉ３のアドレスを出力し、宣言によって変数がメモリにどのように配置されたのか観察した上で考察せよ。

Ⅵポインタと配列について

◎　配列の宣言とポインタ変数の宣言は以下の意味で等価である。

```
int   data[10],*dp;
dp = data;      /* data という配列名は配列の始まるアドレスを指すアドレス定数である。 */
                /* dp = &data[0]; と同等 */
```



ポインタ変数を用いて配列変数にデータを読み込む方法の一例を以下に示す。

【ポインタ変数を使う場合】

```
int data[10], *dp;

for(dp=data; dp<data+10; dp++)
scanf("%d", dp);
```

【ポインタ変数を使わない場合】

```
int data[10],i;

for(i=0; i<10; i++)
scanf("%d", &data[i]);
```

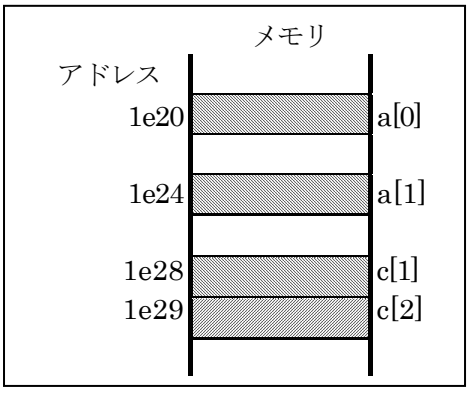
int 型のポインタ変数 dp にとって、dp++のように「dp を１増やす」というのは、「int 型１つ分先に進める」という意味になる。すなわち、１増やすといっても実際のアドレスは２バイト整数ならば２大きくなるし、４バイト整数ならば４大きくなっていることになる。従って、同じインクリメント演算子（またはデクリメント演算子）を用いても、「１増える（減らす)」という意味が、ポインタ変数の型によって異なる。

例

```
int a[2], *ip;
char c[2], *cp;

ip = a;                 /* ip = &a[0] と同じ */
cp = c;
printf("ip=%p,   ",ip);
ip++;
printf("ip+1=%p¥n",ip);

printf("cp=%p,   ",cp);
cp++;
printf("cp+1=%p¥n",cp);
```

実行例（int 型が４バイト整数の場合）

```
ip=1e20,   ip+1=1e24
cp=1e28,   cp+1=1e29
```



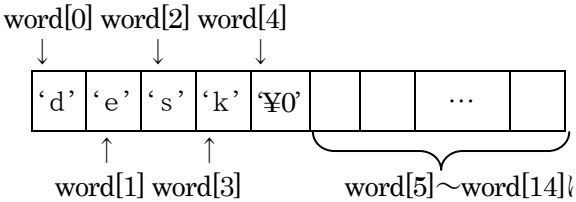
◎　文字列の読み込み方

［その１］　scanf( )を用いる場合

```
char  word[15];      15 文字まで格納できる文字配列変数を宣言しておく。( '¥0' があるため実際には 14 文字分)
scanf("%s",word);
```

※ %s は文字列を扱うための書式、word の前に&をつけないこと！

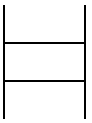
ここで例えば、desk と入力してリターンキーを押すと、word[0]から順に以下のように格納される。



注) scanf を用いる時は入力する文字列が上の場合で 15 文字を越えると、文字列の
切れ目である¥0 が入らなくなるので結局文字列として扱えなくなってしまうことになる。

［その２］　getchar( )（１文字入力関数）を用いる場合
getchar( )の最も簡単な使い方は以下の通りである。

```
int  c;
c=getchar( );
```



入力された文字が入る。

getchar( )は自分が呼ばれるとキーボード（以外の時もあるが）から入力された文字（のアスキーコード）を関数値として返す。上の場合は、「結果を c に代入せよ」とあるので、結局入力された文字が c に格納される。

以上はあくまで、１文字の入力のためであり、これを用いて文字列の入力をするには、例えば次のように行なう。

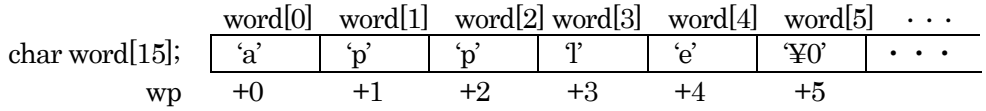
ここではリターンキー('¥n'という文字)が入力されるまでの文字を文字列として取り込む方法を示す。

```
int     i,  ch;
char  word[15];

for( i=0; (ch=getchar( ))! = '¥n'; i++) word[i] = ch;
word[i] = '¥0';
```

ポインターを利用した方法

```
int     i,  ch;
char  word[15], *wp;

wp = word;      /* wp = &word[0]  と同じ  */
while( (ch=getchar( )) != '¥n' ) *(wp++) = ch;
*wp = '¥0';
```

apple を読込んだ時



となる。

ポインターを使用した時は、wp が+5 まで行っていることに注意。

*(wp++)について。
意味：wp のポインターの指している場所の中に１文字入れて、その後 wp に１を加える。

以上のように getchar( )を応用して文字列を読み込む場合には'¥0'を自分で付加するようにプログラムを作らねばならない。

・ポインター使用の別プログラム

```
char  word[15], *wp;
wp=word;
while( (*(wp++)=getchar( )) != '¥n' );
*(--wp)='¥0';
```

［その３］　gets()（１行入力関数）を用いる場合

```
char line[100];

gets(line);
```

キーボードから入力される改行文字までの１行（空白文字の部分も含む）を文字配列に読み込むのに、最も簡潔な記述である。ただ、読み込みと同時に各文字について調べる必要がある場合には、getchar( )を使う方が便利な場合が多い。

◎　文字列の操作

☆文字列をコピーする場合

word1[ ]の文字列を word2[ ]にコピーするには、

```
（ int i;   char word1[15], word2[15];）
for(i=0; word1[i] != '¥0'; i++) word2[i] = word1[i];
word2[i] = '¥0';
```

別プログラム　　for(i=0; word2[i]=word1[i]; i++);　　// == ではないことに注意。

word1[ ]中の文字列の終了は'¥0'のはずであり、'¥0'はアスキーコード 0 の文字、すなわちデータとして 0 であるので、'¥0'を word2 の方へコピーした所で、代入式全体の値が 0、すなわち偽となり、for ループを抜け出す。

あるいはポインタを用いて

```
 char *wp1, *wp2, word1[15], word2[15];
wp1 = word1;   wp2 = word2;
while( *(wp2++)=*(wp1++) );
```

ループの回り方は上と同じ。

☆文字列を比較する場合

文字列１ word1	'm'	'o'	'o'	'n'	'¥0'		
文字列２ word2	'm'	'o'	'n'	'd'	'a'	'y'	'¥0'

要するに、各文字列の先頭から１文字ずつ取出して等しいかどうか比較しすべて等しいままどちらも同時に'¥0'に到達したら、完全に２つの文字列は等しいことになる。

【参考例】

```
(int n;)

① for(n=0; word1[n]; n++);   /* word1 に'¥0'が出てくるまでまわるループ */
② for(n=0; word2[n]; n++);   /* word2 に'¥0'が出てくるまでまわるループ */
③ for(n=0; word1[n] == word2[n]; n++);
                              /* word2 と word1 に同じ文字が出てくる間、n を増やしていくループ */
```

文字列の比較（等しいか否か）を判定するには上の①、②、③のループをどのように１つにまとめれば良いか。また、そのループを抜けた後２つの文字列が等しくあるためにはどのような条件が成立っていなければならないか、これは各自で考えてみよ。

（①②③と３つループが必要に見えるが、ループは①のみの１個で作成すること。）

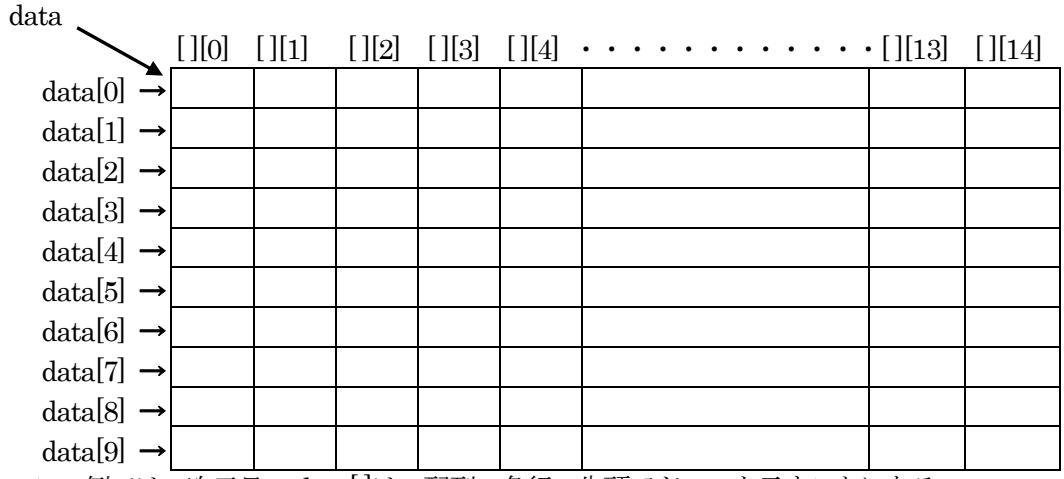
・for(n=0; word1[n]; n++); は for(n=0; word1[n] != '¥0'; n++); のことです。

二次元配列のアドレス

以下のような二次元配列の宣言を行った場合、アドレスとの関係は次のようになる。

int data[10][15];

配列名は、宣言した二次元配列全体の先頭アドレスを示すアドレス定数である。



この例では一次元目の data[ ]は、配列の各行の先頭アドレスを示すことになる。

しかし、メモリは本来一次元的な広がりを持つものである。二次元配列のイメージとしては上の図のようになるが、この配列は実際のメモリ内では、一次元的に確保される。その様子を調べるには例えば次のようなプログラムで確認できる。

［演習７］実行結果を見てどのように確保されたのか考察せよ。

```
#include <stdio.h>
int main( )
{
    int data[10][15];
    int i,j;
    printf("data :%X¥n", data);
    for(i=0; i<10; i++){
        printf("data[%d]:%X ",i,data[i]);
        if((i+1)%5 == 0) printf("¥n");
    }
    printf("¥n");
}
```

実行結果

```
data :51B94
data[0]:51B94 data[1]:51BD0 data[2]:51C0C data[3]:51C48 data[4]:51C84
data[5]:51CC0 data[6]:51CFC data[7]:51D38 data[8]:51D74 data[9]:51DB0
```

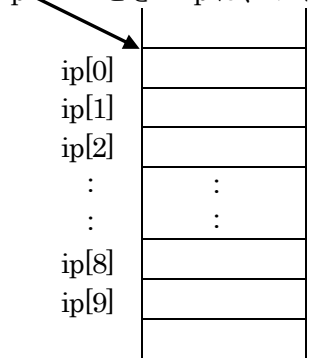
ポインタ配列

ポインタ自身を配列変数で宣言することができる。たとえば、整数型の変数を指す１０個のポインタが必要な場合、

```
int *ip[10];
```

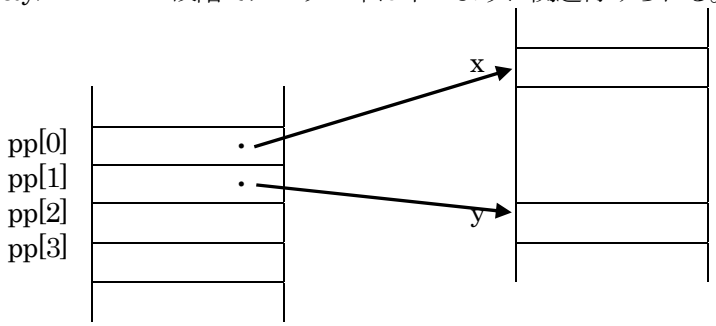
と宣言すると、ip[0]～ip[9]までの１０個のポインタが用意できる。

ip　このときの ip は、ポインタ配列の先頭アドレスを指すアドレス定数。



これらの変数はポインタであるから、他の変数のアドレスを格納することができる。
逆に、他の変数のアドレスを格納してからでなければこれらのポインタは役に立たない。

【例】
```
int x, y, *pp[4];
pp[0]=&x;
pp[1]=&y;
```
この段階でメモリの中は下のように関連付けられる。



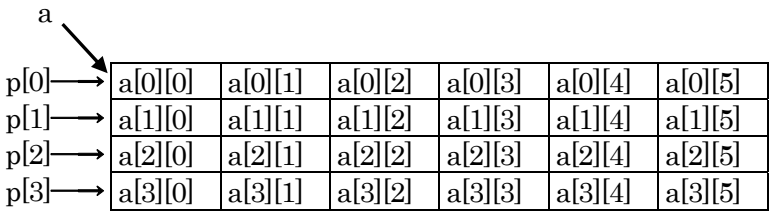
この例では、pp[0]と pp[1]は他の変数のアドレスをセットされたので、x,y の内容を参照したり変更したりするのに利用できるが、pp[2]と pp[3]については指すアドレスを格納していないのでまだ役に立っていない。

ポインタ配列の利用例

アドレスを格納するポインタを配列として宣言すると、例えば次のようになる。

```
int a[4][6], *p[4], n;
for(n=0; n<4; n++) p[n]=a[n];
```

これにより、ポインタ配列の各要素は、次のように二次元配列の各行を指すポインタとなる。



a[2][3]を参照する場合、p[2]を用いると、

(1) *(p[2]+3) で参照できる。

(2) 現在 p[2]は a[2][0]を指しているので、p[2]を３回インクリメントする。
（または p[2]+=3 を行う）

のどちらかの方法を使うことができる。(1)はポインタ p[2]の指す場所は変わらないが、(2)の方では変わってしまう。どちらを使うかは、使う状況によって選ぶ。

◎ 文字列とポインタ配列

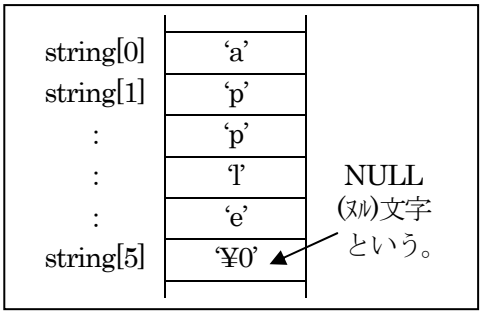
次のような文字配列の初期化宣言をするとメモリ内では右下のようになる。

◇ 一次元配列の場合

```
char string[ ]="apple";
```

↑ 初期化するときは文字数は入れなくても良い。

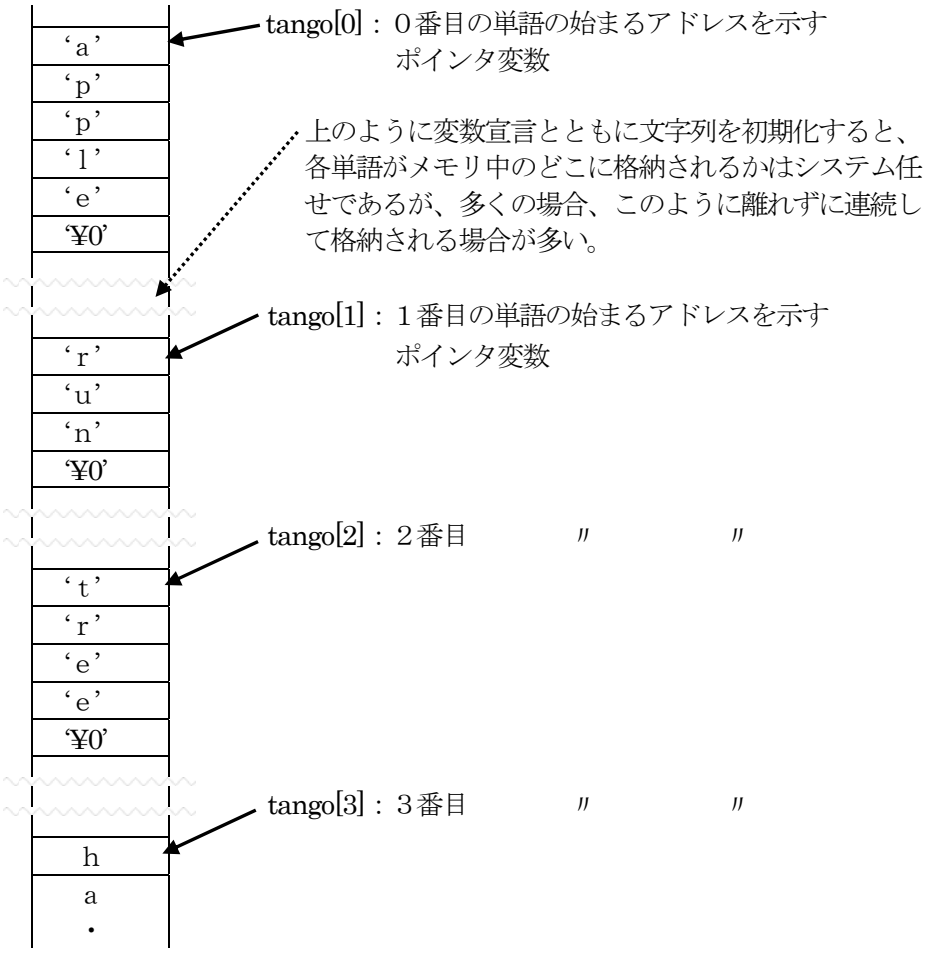
実際には string[0]には a という文字のアスキーコード（６１$_{(16)}$）が入っている。 また文字列の最後には"…"を使った場合、¥0（アスキーコードでも０$_{(16)}$）が自動的に付加される。



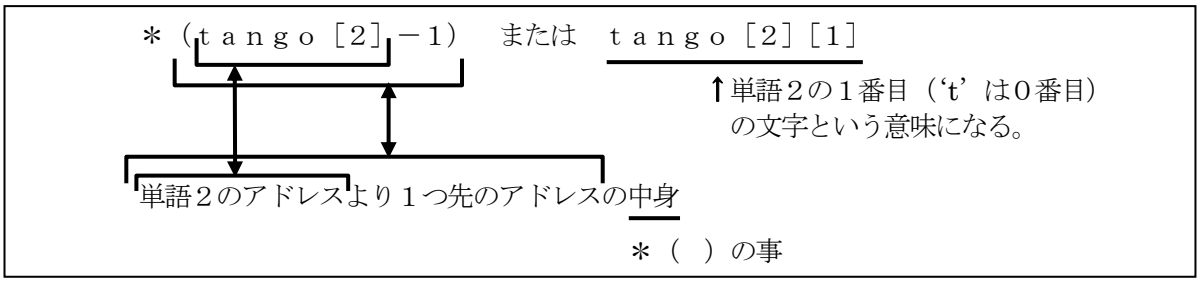
◇ ポインタ配列を複数文字列の先頭アドレスで初期化（最後のセミコロンは必要！）

```
char *tango[ ] = {
        "apple","run","tree","happy", ...
};
```

と宣言すると、



例えば "tree" という文字の２つ目の文字ｒを取出すには



Ⅶコマンドラインの取得

今までの例題・課題では、main 関数は引数を持たない関数として定義してきた。ここでは、main 関数がとる引数について説明する。main 関数の引数は、プログラムの実行を開始するときに指定されていればそれらが渡されるので、コマンドラインの取得とも呼ばれる。そして main の引数の個数やその内容には特別な意味がある。

例１　氏名をローマ字で入力すると、イニシャルを表示するプログラム initial.c を考える。

①main が引数をとらない今までのやり方

```
int main( )
{
        char snd_name[10],fst_name[10];
        scanf("%s%s",snd_name,fst_name);
        printf("Your Initial ==> %c.%c¥n",fst_name[0],snd_name[0]);
}
```

そして実行の仕方は

```
% initial ・          実行ファイル名を入力後、scanf のため入力待ちになるので、氏名を入力する。
Yamada Takashi ・
Your Initial ==> T.Y
```

②main に引数を渡して実行する方法（コマンドラインの取得）

```
int main(int argc, char *argv[ ])
{
        printf("Your Initial ==> %c.%c¥n",argv[2][0],argv[1][0]);
}
```

実行方法は

```
% initial Yamada Takashi ・　このように実行ファイル名と一緒に引数を指定する。
Your Initial ==> T.Y
```

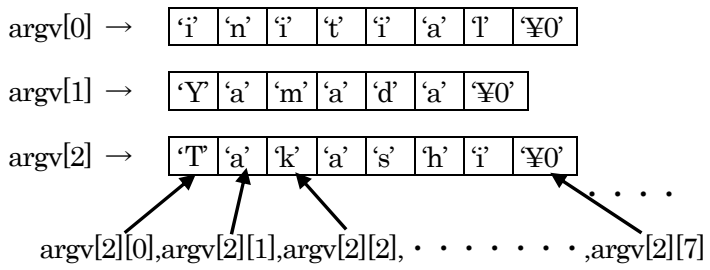
main の引数の持つ意味は次の通りである。

argc　:コマンドライン上の引数の個数（コマンド名の分も含まれる）

argv[ ] :コマンドライン上の各文字列へのポインタ

上の例では、

argc　= 3



Ⅷファイルへの入出力

１．高水準入出力関数と低水準入出力関数

Ｃでは、ディスクファイルへの入出力もデバイス (ディスプレイとかプリンタ等) への入出力も同じものと考え、すべてファイルへの入出力としている。

ファイルへの入出力機能

・低水準入出力関数
　使用するシステムに依存する部分が多い　→　移植性が低い

・高水準入出力関数
　いくつかのシステムコール（低水準入出力関数など）を組合わせたもの　→　移植性が高い

以下では主に高水準入出力関数について説明する。

２．ファイル入出力の手順

①ファイルをオープンする

1)システムにオープン内容を知らせる。
　ファイル名（入出力パス名）とオープン形式（リードかライトか等）を知らせる。

2)システムからオープン処理結果が返される。
　オープン成功の時、ファイルディスクリプタが返り、
　オープン失敗の時、エラー通知が返る。

ファイルディスクリプタ:高水準入出力関数では、ファイル型構造体へのアドレスがリターンされるので、それをポインタ（ファイルポインタ）に受け取ることになる。

②ファイルの入出力を行なう。

1)ファイルディスクリプタに対して、ファイルアクセスを行なう。

2)システムからアクセス結果が返される。

③ファイルをクローズする。

ファイルディスクリプタに対して、ファイルクローズを行なう。

３．高水準入出力ファイルへの入出力

３．１　ファイル型の宣言

FILE *fp;　関数 fopen( )のリターン値をセットする（fopen は後述)。

FILE 型構造体

・ファイルを読み書きしている位置（データへのポインタ）
・バッファ中のデータ残数
・バッファの先頭へのポインタと大きさ
・ファイルを扱うモード
・ファイル番号（Filds）

３．２　ファイルのオープン

fp = fopen(path, mode);

path　‥‥ オープンしようとするファイル名を示すパス名（char * ）

mode ‥‥ アクセスモード（char *）

"r"　リードオープン（入力）
"w"　ライトオープン（出力）
"a"　アペンドオープン（追加出力）
他にも "r+", "w+", "a+" などがある。

fp ‥‥‥ ファイルポインタ（FILE *）

成功の場合：オープンしたファイルへのポインタ（≠0）
失敗の場合：0（NULL と等価）が返される。

３．３　ファイル入出力

◎指定したファイルから１文字入力する

c = getc(fp);
c = fgetc(fp);
　　fp ・・・・・・ ファイルポインタ（FILE *）
　　c　・・・・・・ 結果（int）　成功：入力された文字,
　　失敗：－1（EOF と等価）

getc と fgetc は処理結果は同じ。ただし、getc の方が処理スピードが速く、fgetc の方がメモリ効率が高い。

c = getchar( );　標準入力（stdin）から１文字入力
　　c　・・・・・・ 結果（int）　成功：入力された文字,
　　失敗：－1（EOF と等価）

◎指定したファイルに１文字出力する。

err = putc(c, fp);
err = fputc(c, fp);
　　c　・・・・・・ 出力する文字（int）
　　fp ・・・・・・ ファイルポインタ（FILE *）
　　err　・・・・ 結果（int）　成功：出力した文字
　　失敗：－1

err = putchar(c);　標準出力（stdout）へ１文字出力
　　c　・・・・・・ 出力する文字（int）
　　err　・・・・ 結果（int）　成功：出力した文字
　　失敗：－1

◎１行単位の入出力

err = gets(s);　標準入力から s に '¥n'または EOF が入力されるまで読み込み、
　　'¥n'または EOF の代わりに '¥0' が付けられる。
　　s　・・・・・・ 入力バッファ（char *）　←十分な大きさを持つように。
　　err　・・・・ 結果（char *）　成功： s (入力バッファの先頭アドレス)
　　失敗： NULL ポインタ（=0)

err = fgets(s, n, fp);　指定したファイルから入力バッファ s に、s が一杯になるか、
　　'¥n'または EOF が入力されるまで読み込み、'¥n'あるいは EOF の後に'¥0'が付加される。
　　s　・・・・・・ 入力バッファ（char *）
　　n　・・・・・・ 入力要求文字数（int）
　　fp ・・・・・・ ファイルポインタ（FILE *）
　　err　・・・・ 結果（char *）　成功： s ,　失敗： NULL ポインタ

err = puts(s);　s の中の文字を、標準出力（stdout）へ'¥0'（NULL）が見つかるまで出力し、
　　'¥0'の代わりに'¥n'を出力する。
　　s　・・・・・・ 出力する文字列（char *）
　　err　・・・・ 結果（int）　成功： ≧0　,　失敗： EOF(=－1)

err = fputs(s, fp);　s の中の文字を指定したファイルへ'¥0'（NULL）が見つかるまで出力し、
　　'¥0'は出力しない。
　　s　・・・・・・ 出力する文字列（char *）
　　fp ・・・・・・ ファイルポインタ（FILE *）
　　err　・・・・ 結果（int）　成功： ≧0　,　失敗： EOF(=－1)

◎書式付き入出力

err = scanf(format, arg, …);
　　format ‥ 書式指定文字列（char *）
　　arg　‥‥ 変換入力する変数のポインタ（型 *）
　　err　‥‥ 結果（int）
　　成功：入力された個数（コンパイラによっては 0）
　　失敗：－1（EOF と等価）
　　EOF　：－1

err = fscanf(fp, format, arg, …);
　　fp ‥‥‥ ファイルポインタ（FILE *）
　　他は scanf と同じ。

len = printf(format, arg, …);
　　format ‥ 書式指定を含む文字列（char *）
　　arg　‥‥ 変換出力するデータ（型）
　　len　‥‥ 結果（int）
　　成功：出力したバイト数
　　失敗：＜0

len = fprintf(fp, format, arg, …);
　　fp ‥‥‥ ファイルポインタ（FILE *）
　　他は printf と同じ。

３．４　ファイルのクローズ

err = fclose(fp);　fopen でオープンしたファイルは必ずクローズすること。
　　fp ‥‥‥ ファイルポインタ（FILE *）
　　err　‥‥ クローズ結果（int）
　　成功：　0
　　失敗：－1

【参考】

標準入出力用に次の３つのファイルポインタがシステムによりあらかじめ予約されている。

　　stdin　　：標準入力
　　stdout　：標準出力
　　stderr　　：標準エラー出力

従って、例えば printf を使用した以下の例

　　printf("Answer　X=%d　Y=%d¥n", x, y);

という出力文は、

　　fprintf(stdout, "Answer　X=%d　Y=%d¥n", x, y);

のように、fprintf 文で、ファイルポインタを stdout に指定したものと等価となる。

４．　応用例

◎UNIX の cat コマンド（あるいは MS-DOS の type コマンド：ファイルの内容表示）と同等のプログラム
　プログラム名を show.c にしたとすると実行形式は、
　　% show　ファイル名
　となる。

実現法その１

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[ ])
{
      int c;
       FILE *fpr;
       if(argc!=2){
            printf("コマンドの使い方が違います¥n");
           exit(1);
       }
       if( (fpr=fopen(argv[1],"r")) ==NULL ){
            printf("指定したファイルがオープンできません¥n");
            exit(1);
       }
       while( (c=getc(fpr))!=EOF ) putchar(c);
       fclose(fpr);
}
```

実現法その２

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[ ])
{
     char line[256];
     FILE *fp;
     void errmsg(int);

     if(argc!=2) errmsg(0);
     if( (fp=fopen(argv[1],"r")) == NULL ) errmsg(1);

     while( fgets(line,256,fp) != NULL ) puts(line);   /* gets と fgets の EOF 判断は、NULL で行うことに注意 */

      fclose(fp);
}
void errmsg(int code)
{
      switch(code){
           case 0: printf("コマンドの使い方が違います¥n"); break;
           case 1: printf("指定したファイルがオープンできません¥n");
      }
      exit(1);
}
```

◎ファイルを別の名前のファイルにコピーするプログラム（UNIX の cp,MS-DOS の copy コマンド相当）
プログラム名を filecp.c にしたとすると、実行形式は次のようになる。
% filecp file1 file2　（file1 を file2 という名のファイルにコピーする）

実現法その１

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[ ])
{
    int c;
    FILE *fpr, *fpw;
    if(argc!=3){
        printf("使用法: %s 入力ファイル名 出力ファイル名¥n",argv[0]);
        exit(1);
    }
    if( (fpr=fopen(argv[1],"r")) ==NULL ){
        printf("入力ファイル[%s]がオープンできません。¥n",argv[1]);
        exit(1);
    }
    if( (fpw=fopen(argv[2],"w")) ==NULL ){
        printf("出力ファイル[%s]がオープンできません。¥n",argv[2]);
        exit(1);
    }
    while( (c=getc(fpr))!=EOF ) putc(c,fpw);
    fclose(fpr);
    fclose(fpw);
}
```

実現法その２（main 関数以降のみ記載）

```
int main(int argc, char *argv[ ])
{
    char word[256];
    FILE *fpr, *fpw;
    if(argc!=3){
        printf("使用法: %s 入力ファイル名 出力ファイル名¥n",argv[0]);
        exit(1);
    }
    if( (fpr=fopen(argv[1],"r")) ==NULL ){
        printf("入力ファイル[%s]がオープンできません。¥n",argv[1]);
        exit(1);
    }
    if( (fpw=fopen(argv[2],"w")) ==NULL ){
        printf("出力ファイル[%s]がオープンできません。¥n",argv[2]);
        exit(1);
    }
    while( fgets(word,256,fpr) != NULL ) fputs(word, fpr); /* gets と fgets の EOF 判断は、NULL で行う */
    fclose(fpr);
    fclose(fpw);
}
```

［演習８］コマンドラインから、ある文字列とファイル名を読込むと、そのファイル名の中に、その文字列がいくつあるか探すプログラムを作成せよ。
なお、使用するファイルは、/home/usr2/tmp/cprg/2001/ensyu8.dat からコピーして使用すること。

チェックデータ（必ずこのデータを使用して提出すること）
are　6 個,　he　17 個,　aa　12 個,　bb　12 個,　　について実行して結果を確かめよ。

Ⅸ関数

関数とは：機能的に意味を持つ一連のコードを、独立性・再利用性を目的に１つにまとめたもの。

関数は、他の関数から呼び出され、特定の仕事をする。

関数は、その呼び出され方に２種類ある。

Call by value

Call by reference

関数（その１）　Call by value

関数を作るために必要なこと

(1) 何をさせるための関数か。

(2) 関数に仕事をさせるためには何を与えればよいか（引数の決定)。

(3) その関数は呼び出された結果として、何を返してくれるのか（リターン値の決定)。

(4) 与えられたものから、返すものを求めるために何をすれば良いか（関数本体の決定)。

関数のプロトタイプ宣言

プロトタイプ宣言は、その関数を呼び出す関数の中で行うか、いくつかの関数の中から共通に呼 び出すのならば、関数外部で宣言する。

リターン型　　関数名(引数１のデータ型, 引数２のデータ型,・・・)

リターン型はその関数は呼び出し側に返す値のデータ型を指定する。リターンされる値　は１つだけである。

関数の定義

定義される関数は、プロトタイプ宣言されたもの同じ関数名・引数・リターン値を持ち、さらに 関数本体の記述を行う。

【例１】円の面積を求めるための関数 circle

```
#include <stdio.h>
int main( )
{
    double r, s;
    double circle(double);     /*  関数 circle( )のプロトタイプ宣言  */

    printf("Input a radius => ");
    scanf("%lf", &r);
    s=circle(r);                    /*  関数 circle( )の呼び出し  */
    printf("Size=%17.8lf ¥n", s);
}

double circle(double r)
{
    double ans;

    ans = r * r * 3.1415926;
    return ans;     /*  呼び出し側へ、値を返して、circle は終了  */
}
```

関数が呼び出し側に return で返せる値は１つのみ。従って、この例で、円周の長さも一緒に求めて、円の面積と両方返すことはできない。

【例２】整数ｎを与えられると、ｎの約数を小さい順にすべて表示する関数 factor( )。

```
#include <stdio.h>
int main( )
{
    void factor(int);          /*  関数 factor( )のプロトタイプ宣言  */
    int n;

    printf("Input a number :");
    scanf("%d", &n);
    factor(n);                 /*  関数の呼び出し  */
    printf("      ....end.¥n");
}

void factor(int a)
{
    int k;
    for(k=1; k<=a; k++)
        if(a%k==0) printf("%d    ",k);
    printf("¥n");
}
```

X関数（その２）　Call by reference

C++では、デフォルト引数、参照引数（プロトタイプ宣言の引数と関数側の引数に&を付加する）というのがありましたが、Ｃでは、デフォルト引数はありません。また、参照渡しにするためには、アドレスを送ることになります。

関数が呼び出し側の関数に返すことのできる値の個数は、return 文を使うことから明らかなように、最大１つである。しかし時には、呼び出される関数に複数の値を求めてもらい、それを呼び出し側で参照したりすることが要求される場合がある。この場合、return 文ではその要求を満たすことはできない。また、配列変数のような複数の値から成るものを、関数に渡したい場合はどうすれば良いのだろうか?

このような２つの要求がある例として、いくつかのデータを格納した整数配列を渡し、その中の最大値と最小値を求めてくれる関数 maxmin を作ることを考える。maxmin に渡したい配列の一例として、

int a[100]={68,39,73,28,49,84,91,58,34,73,29,36,18,96,83,81,-1}

を考える。データはすべて正の整数で、データの最後として-1 という値が格納されている。さて関数 maxmin に配列 a をどのように引数として渡せば良いだろうか？　配列にデータがいくつ入っているか分からないので、まさかデータ個数分だけ引数に int 型の引数変数を並べるわけには行かない。次にこの関数のための適切なプロトタイプ宣言の一例を示す。

```
void maxmin(int *, int *, int *);          /* maxmin のプロトタイプ宣言 */
```

maxmin の第１引数で、配列の先頭アドレスを受け取る。第２引数では最大値を格納する変数のアドレスを、第３引数では最小値を格納する変数のアドレスを受け取る。このようにアドレスを引数として渡す関数呼び出しを、Call by reference という。

Call by value で呼び出される関数には呼び出し側の値がコピーされて渡されるので、呼び出された側で受け取った局所(ローカル)変数の値を変更しても、呼び出し側の変数には何の影響もないが、アドレスを関数に渡した場合は、呼び出された関数が渡されたアドレスを通して呼び出し側の変数の内容を変更することもあり得るので、注意が必要である。

このような maxmin と main 関数を含めた全体を示すと以下のようになる。

【例１】整数配列の要素の最大値と最小値を求める関数 maxmin

```
#include <stdio.h>
int main( )                          /*（ー１もデータに入れてしまうのはちょっと違和感がありますが...）*/
{
    int a[100], i, max, min;
    void maxmin(int *, int *, int *);

    for(i=0; i<100; i++){
        printf("正の整数を入力してください--> ");
        scanf("%d", &a[i]);
        if(a[i] ==-1) break;
    }
    printf("データの入力が終了しました。¥n");
    maxmin(a, &max, &min);
    printf("最大値=%d,  最小値=%d¥n", max, min);
}

void maxmin(int *data, int *px, int *pn)          /* maxmin 本体の定義 */
{
    *px = 0;                         /*  最大値格納のための初期値設定 */
    *pn = *data;                     /*  最小値格納のための初期値設定 */
    for( ; *data!=-1; data++){
        if( *px < *data ) *px = *data;
        if( *pn > *data ) *pn = *data;
    }
}
```

解説：main( )の方に領域を確保し、そのアドレスを関数に渡している。関数内では、そのアドレスを介して、直接、main( )側の変数に値を入れている。

上の例１ですが、以下のように書くこともできます。

【別例１】整数配列の要素の最大値と最小値を求める関数 maxmin

```
#include <stdio.h>
int main( )                    /*（ー１もデータに入れてしまうのはちょっと違和感がありますが...）*/
{
    int a[100], i, max, min;
    void maxmin(int [ ], int [ ], int [ ]);

    for(i=0; i<100; i++){
        printf("正の整数を入力してください--> ");
        scanf("%d", &a[i]);
        if(a[i] ==-1) break;
    }
    printf("データの入力が終了しました。¥n");
    maxmin(a, &max, &min);
    printf("最大値=%d,  最小値=%d¥n", max, min);
}

void maxmin(int data[ ], int px[ ], int pn[ ])        /* maxmin 本体の定義 */
{
    int i;
    px[0] = 0;                              /*  最大値格納のための初期値設定 */
    pn[0] = data[0];                        /*  最小値格納のための初期値設定 */
    for( i=0; data[i]!=-1; i++){
        if( px[0] < data[i] ) px[0] = data[i];
        if( pn[0] > data[i] ) pn[0] = data[i];
    }
}
```

つまり、＊１個は[　]１対に対応します。（２次元配列以上はちょっと異なりますが。）

ポインターは、宣言しただけでは使えません。中にアドレスを入れて初めて使い物になるのです。
ちょうど、int sum;を初期宣言しないと使えないのと似ています。

演習：上記のプログラムで、maxmin は void 型の関数として定義されているが、渡された配列のデータの平均値を double 型で返す関数に変更する場合、main および maxmin を適切に修正せよ。

【例２】２つの変数の中身を入れ替える関数を作るには

Call by value　だけでは不可能

↓

```
#include <stdio.h>
int main( )
 {
        int   a,b;
        void swap(int,int);

        printf("input a=");
        scanf("%d",&a);
        printf("input b=");
        scanf("%d",&b);
        swap(a,b);
        printf("new a=%d¥n",a);
        printf("new b=%d¥n",b);
}
void swap( int x,int y)
{
        int temp;
        temp=x;
        x=y;
        y=temp;
}
```

Call by reference ならば可能

↓

```
#include <stdio.h>
int main( )
{
        int   a,b;
        void swap(int *,int *);

        printf("input a=");
        scanf("%d",&a);
        printf("input b=");
        scanf("%d",&b);
        swap(&a,&b);
        printf("new a=%d¥n",a);
        printf("new b=%d¥n",b);
}
void swap( int *x,int *y)
{
        int temp;
        temp=*x;
        *x=*y;
        *y=temp;
}
```

【例３】Call by value で取り上げた関数 circle を円周も求められるように変更する。

```
#include <stdio.h>
#include <math.h>
int main( )
{
        double r, s, l;
        double circle(double, double *);
                /*  第１引数で半径を受け取り、第２引数で円周を入れる変数のアドレスを受け取る  */
        printf("Input a radius => ");
        scanf("%lf", &r);
        s=circle(r, &l);                /*  関数 circle( )の呼び出し  */
        printf("Size=%17.8lf ¥n", s);
}
double circle(double r, double *pl)
{
        double ans;

        ans = r * r * M_PI;
        *pl = 2 * r * M_PI;             /*  円周は main の変数に直に代入しておく  */
        return ans;                     /*  面積は return で main に返す  */
}
```

【例４】英小文字から成る文字列を与えると、指定された文字のみを大文字に変更する関数 change_c

```
#include <stdio.h>
int main( )
{
    char line[80], a, *p;
    int   char1;
    void change_c( char *, char );

    printf("英小文字から成る１文を入力せよ。¥n");
    for( p=line; (char1=getchar( ))!='¥n'; p++) *p = char1;  /*  文字列を入力 */
    *p = '¥0';                                  /*  文字列末尾の¥n をヌル文字に変更 */
    printf("大文字に変更したい英小文字を入力せよ---> ");
    a = getchar( );              /* EOF は入らないという条件で char を使用している */
    change_c(line, a);           /*  文字列と大文字変換する文字を渡して関数 change_c へ */
    printf("変換結果は次の通りです。¥n%s¥n", line);
}
void change_c(char *str, char c)
{
    for( ; *str; str++)
        if( *str==c ) *str += 'A'-'a';     /*  指定された文字を大文字に変換 */
}
```

【例５】m 次の正方行列の転置行列を求める関数 trans

```
#include <stdio.h>
int main( )
{
    int matrix[10][10], i, j, m;
    void trans(int[ ][ ], int);

    printf("正方行列のサイズを入力してください---> ");
    scanf("%d", &m);     /*  正方行列のサイズを読み込む */
    printf("行列の要素の値を行優先で入力せよ。¥n");
    for(i=0; i<m; i++)           /*  行列の入力 */
        for(j=0; j<m; j++)
            scanf("%d", &matrix[i][j]);
    trans(matrix, m);            /*  転置行列を求める。結果は matrix 自身に作られる */
    for(i=0; i<m; i++){          /*  転置行列を表示 */
        for(j=0; j<m; j++)
            printf("%4d", matrix[i][j]);
        printf("¥n");
    }
}
void trans(int mtrx[ ][10],int msize)   /*  この場合、列サイズは指定しなければならない */
{
    int i, j, tmp;

    for(i=0; i<msize; i++)
        for(j=i; j<msize; j++){
            tmp=mtrx[i][j];
            mtrx[i][j]=mtrx[j][i];
            mtrx[j][i]=tmp;
        }
}
```

変数の種類と宣言位置による有効範囲（＝変数のスコープ）の関係

変数の種類	宣言場所	有効範囲	初期値	初期化
自動変数	関数内	関数内またはブロック内	不定	毎回
外部変数	関数外	全域	0	初回のみ
静的変数（内部）static	関数内	関数内	0	初回のみ
静的変数（外部）static	関数外	コンパイル単位内	0	初回のみ
レジスタ変数	関数内	関数内またはブロック内	不定	毎回

【練習】　変数のスコープ：次のプログラムの出力結果を示せ。

```
#include <stdio.h>
int sum=0;
int main( )
{
      int n,i,ans1,ans2;
      int func1(int),func2(int);

      n=10;
      for(i=1; i<=n; i++) sum +=1;
      ans1=func1(n/2);
      ans2=func2(sum);
      printf("Sum3=%d, Ans1=%d, Ans2=%d¥n",
                  sum, ans1, ans2);
}
int func1(int n)
{
      int k,i=0;
      for(k=1; k<=n; k++) sum+=1;
      printf("Sum1=%d¥n",sum);
      i += k;
      return( sum/k );
}
int func2(int sum)
{
      int n;

      for(n=0; n<10; n++) sum+=1;
      printf("Sum2=%d¥n",sum);
      return n;
}
```

［演習９］整数配列の先頭ポインターと、データ個数を渡すと、最大値、最小値、平均値を返す関数を作成せよ。
この時、最大値と最小値に関しては、int maxmin[2];という配列を main 側で作成し、関数へは、
その maxmin のアドレスを渡すことにせよ。従って、関数へ渡すのは、引数４個です。

```
int maxmin[2];      /* maxmin[0]を最大値として使用、maxmin[1]を最小値として使用 */
```

これを使用して、5 人の学生の点数を読込んで、最高点、最低点、平均値を出力するプログラムを作成せよ。
データ個数は読込み、最大、最小の制限はないものとるす。

※関数の中では入出力は使用しないこと。

チェックデータ（必ずこのデータを使用して提出すること）

出力例)
データ個数を読込んで下さい。: 5
4　6　2　8　3
最大値は 8 です。　最小値は 2 です。　平均は 4.6 です。

関数へのポインタ

関数名は、その関数を実行するプログラムの先頭アドレスを表している。そこで、関数へのポインタを定義し、そのポインタ変数に関数のアドレスを代入することによって、複数の関数間で、変数を共有するのと同様に関数も共有することができる。すなわち、“他の関数に関数を渡す”ことができる。

・関数へのポインタの宣言形式
　データ型　（＊変数名）（ ）;
・関数へのポインタ配列の宣言形式
　データ型　（＊変数名［要素数］）（ ）;

いずれも最後の（ ）が変数ではなく関数へのポインタであることを表している。

［例］キーボードから+,-,*,/を入力して、２数（２０と１０）の和差積商を求める。

```
#include <stdio.h>
int main( )
{
        int comp( int a, int b, int (*func)( ) );
        int add( int c, int d), sub( int c, int d);
        int mul( int c, int d), div( int c, int d);
        int (*func[4])( ) = { add, sub, mul, div };
        int x, y, z;

        x = 20;     y = 10;
        while( (z=getchar( )) != EOF ) {
            switch(z) {
                case '+' : z = comp( x, y, func[0] ); break;
                case '-' : z = comp( x, y, func[1] ); break;
                case '*' : z = comp( x, y, func[2] ); break;
                case '/' : z = comp( x, y, func[3] ); break;
            }
            printf("%d¥n", z);
        }
}

int comp( int a, int b, int (*func)( ) )
{
        return( (*func)(a, b) );
}

int add( int c, int d)
{
        printf("%d + %d = ¥n", c, d);
        return( c+d );
}

int sub( int c, int d)
{
        printf("%d - %d = ¥n", c, d);
        return( c-d );
}

int mul( int c, int d)
{
        printf("%d * %d = ¥n", c, d);
        return( c*d );
}

int div( int c, int d)
{
        printf("%d / %d = ¥n", c, d);
        return( c/d );
}
```

◎関数の再帰（Recursion）

再帰とは：　ある関数の定義の中で、その関数自身を呼び出すこと。

簡単な再帰の例

数学の漸化式で表現される計算手順は、プログラム上は非常に素直な再帰で表現される。

【例１：整数Nの階乗Ｎ！を計算する例】

n の階乗を計算する関数をＦで表すと、その漸化式表現は、

$$F(0)=1$$
$$F(n)=n\times F(n-1) \quad (n\geqq 1)$$

である。

この漸化式に従ったプログラムを以下に示す。

```
int main( )
{
    int n;
    long fact( );
    scanf("%d", &n);
    printf("%d! = %ld¥n", n, fact(n));
}

long fact(int i)
{
    if( i==0 ) return(1);
    else return( i*fact(i-1) );
}
```

関数 fact の動きを見てみると、例えば 4!を計算するとき、

①main が fact(4)を呼び出す。
②fact(4)が 4*fact(3)を求めるため、fact(3)を呼び出す。
③fact(3)が 3*fact(2)を求めるため、fact(2)を呼び出す。
④fact(2)が 2*fact(1)を求めるため、fact(1)を呼び出す。
⑤fact(1)が 1*fact(0)を求めるため、fact(0)を呼び出す。
⑥fact(0)は、1 を⑤の fact(0)に返す。
⑦以後、⑤の fact(1)の値がその上に返される、ということを順次繰り返し、①の fact(4)の値が求められる。

n の階乗を求めるために、(n-1)の階乗の計算を必要とする。このように、ある関数がその中で自分自身を必要とする関数が再帰関数である。

注意：再帰処理では、ほぼ自動変数を使用し、外部変数は普通使わない。再帰を用いるとプログラムが簡潔になり見やすくなるが、一方、処理効率は悪くなる。

【例２：キーボードから入力された文字列を逆順に表示する】

通常は、大きなバッファをとり、そこに文字を読み込んで、読み込みが終了したら最後から順に表示していく。
しかし、再帰の場合は、次のような関数を定義すればよい。

関数 inverse_string
①キーボードから一文字読み込む。
②読み込んだ文字が EOF ならば、何も表示せずに終了する。
③読み込んだ文字が EOF でなければ、inverse_string を呼び出し、
それ以後に入力される文字を逆順に表示してもらう。
④最後に、①で読み込んだ文字を表示して、終了する。

このアルゴリズムに従った inverse_string を含むプログラムを以下に示す。

```
#include <stdio.h>
int main( )
{
        void inverse_string( );

        inverse_string( );
        printf("¥n");
}

void inverse_string( )
{
        int c;

        if( (c=getchar( ))!=EOF ){
                inverse_string( );
                putchar(c);
        }
}
```

再帰関数を作る際のポイント
1)再帰の境界条件部分を作る。（２番目の例の②）
2)再帰呼び出しをするときは、その関数が完成されたものであると考えて呼び出す。

再帰関数の応用例
クイックソート（Quick Sort）法のアルゴリズム

例として整数配列を昇順に並べ替える場合を考える。

int 型配列 data[100]にデータが１００個入っている。これをソートさせるため、quicksort 関数には次のような引数を準備する。

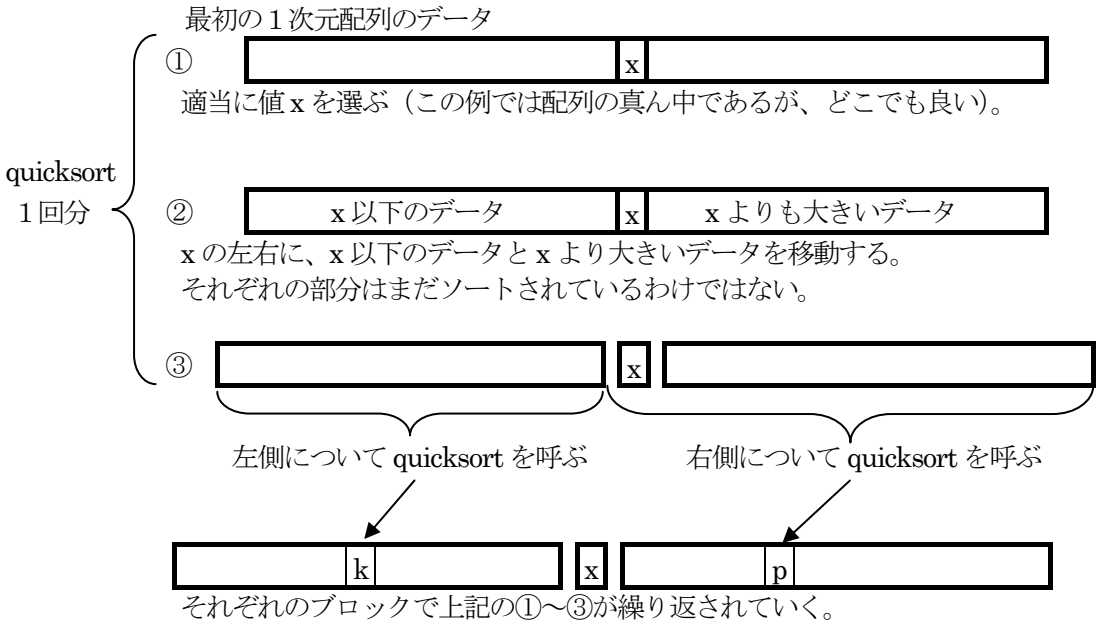
　　void quicksort(int array[ ], int left, int right)

配列 array の left から right まで入っているデータを並べ替える。つまり、上の data 配列をソートさせるには、quicksort(data, 0, 99) のように呼び出して使用することになる。

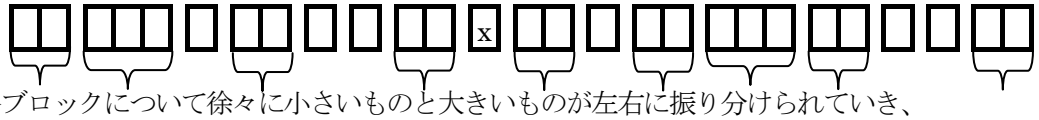
この quicksort の再帰アルゴリズムは次の通りである。

(1)指定された left から right までのデータ数が１個以下ならば、そのままでソートは終了している。
　すなわち、left>=right ならばソートは終了しているということ。
(2)(1)の場合以外なら、left から right までの値に次のような処理を行う。
　(2-1)left と right の真ん中（無い場合はその１つ左側）を代表として選び、それよりも小さい
　　データをその左側に、大きいデータは右側に集める。
　　（注：左右それぞれの側で並べ替えているわけではない。ただ振り分けるだけ。）
　(2-2)左側部分と右側部分について、quicksort に並べ替えてもらって終了。

quicksort のイメージ



quicksort の再帰呼び出しの最終段階では、配列は次のようになっていく。



各ブロックについて徐々に小さいものと大きいものが左右に振り分けられていき、
データ数が１個になったブロックはソートが終了していることになる。
全てのブロックの分解が終了したとき、全体のソートが終了している。

【quicksort のプログラム演習】　以下の空欄を埋めて完成せよ。

```
#include <stdio.h>
#include <stdlib.h>
int main( )
{
   int i,data[100],left,right;
   void quicksort(int [ ], int, int);
   void printarray(int [ ],int);

/* データを乱数で１００個生成 */
   for(i=0;i<100;i++) data[i]=rand( )%1000;
/* データ確認のため配列の内容を表示 */
   printarray(data, 100);
/* ソートさせる */
   quicksort(data,0,99);
/* ソート結果の確認のため配列の内容を表示 */
   printarray(data, 100);
}

void printarray(int a[ ],int n){
   int i;
   for(i=0;i<100;i++){
        printf("%5d",a[i]);
        if(i%10==9) printf("¥n"); /* 10 個ずつ表示して改行 */
   }
   printf(“¥n”);
}

void quicksort(int da[ ], int left, int right){
      void swap(int *, int *);
      int partition(int [ ], int, int);
      int pos;   /* pos:左右振り分けとなる代表値の位置 */

      if( left>=right ) return;   /* 境界条件 */
/* 代表値を選び、その左右に小さいものと大きいものを振り分ける */
      pos=partition(da, left, right);
/* 振り分けられた左右部分それぞれについて quicksort させる */
      [                                                    ]
}

void swap(int *a, int *b) /*  ２数を入れ替える */
{
      int temp;
      if(a!=b){temp=*a; *a=*b; *b=temp;}
}

int partition(int a[ ], int le, int ri){
      [                                                    ]
}
```

［演習 10］

空欄を作成し、結果を確認せよ。

［演習 11］

整数ｎを読込むと、フィボナッチ数Ｆ（ｎ）の値を出力するプログラムを作成せよ。

Ｆ（０）＝Ｆ（１）＝１

Ｆ（ｎ）＝Ｆ（ｎ－２）＋Ｆ（ｎ－１）　　（ｎ＝２、３、－－－）

ただし、フィボナッチ数の計算を行う部分は、**再帰を用いた関数**と、**再帰を用いない関数**と両方の場合のプログラムを作成すること。なお、出力には、この実行例を使用すること。

（実行例）　４を入力すると、F(4)＝5
１を入力すると、F(1)＝1
７を入力すると、F(7)＝21

XI Cの構造体

１．構造体とは・・・・・・・・・・・・・・

複数のデータ型を集めた論理的な１つの処理単位であり、一種のデータ型である。

構造体	名前	年齢	身長
	char 型配列	int 型	float 型

参考：配列は同じ種類のデータ型を集めたもの

配列	要素１	要素２	要素３	・・・・・・・・・・・・・・・	要素ｎ

要素すべて同じデータ型（例えば int 型）

２．構造体のデータ型の定義

```
struct 構造体タグ名 {
        メンバ１;
        メンバ２;
          ・・・・
        メンバｎ;
};
```
（；必要）

例
```
struct person {
    char   name[20];
    int    age;
    float  height;
};
```
この定義により構造体のテンプレートは作られるが、記憶領域はまだとられない。

３．構造体変数の宣言

struct 構造体タグ名　変数１［, 変数２, ・・・・］;

例　struct person player, singer;

変数には配列を用いることもできる（構造体配列）。

構造体の定義と変数宣言を以下のように同時に行うこともできる。

```
struct  person {
        char   name[20];
        int    age;
        float  height;
} player, singer;
```
この場合、構造体タグ名を省略することもできる。

構造体変数の初期化

```
struct person  player={ "ﾀｶﾔﾏ ﾋﾃﾞｱｷ", 19, 169.4 };
struct person  driver[ ]={
                        {"ﾀﾅｶ ｵｻﾑ", 20, 173.5},
                        {"ﾖｼﾀﾞ ﾀｶｼ", 22, 163.6},
                        {"ｺｼﾛ ﾕｳｽｹ", 19, 168.3}
                    };  /* ；が必要です。 */
```
これで driver[0]～driver[2]が初期化される。

４．構造体メンバの参照の仕方

構造体メンバの直接参照

```
player.age    player.height
player.age=26;    player.height=171.2;
```
ここで使われている「.」は構造体演算子と呼ばれる。

【プログラム例１】２つの構造体の内容の入れ換え

```
int main( )
{
    struct person {
        char   name[20];
        int    age;
        float  height;
    };
    struct person  player1={ "ﾀﾅｶ ｵｻﾑ",  20, 173.5 },
                   player2={ "ﾖｼﾀﾞ ﾀｶｼ", 22, 163.6 };
    struct person dummy;

    printf("¥t%-10s %3d %7.1f¥n", player1.name, player1.age, player1.height);
    printf("¥t%-10s %3d %7.1f¥n", player2.name, player2.age, player2.height);

    dummy = player1;
    player1 = player2;      構造体がデータを１まとまりに扱える事
    player2 = dummy;        を示している。

    printf("¥t%-10s %3d %7.1f¥n", player1.name, player1.age, player1.height);
    printf("¥t%-10s %3d %7.1f¥n", player2.name, player2.age, player2.height);
}
```

５．関数への構造体の受け渡し

構造体は、データの１つの固まりであるので、関数の引数として渡すと、構造体の中身がすべて呼び出された関数へコピーされる（Call by value によって関数を呼び出す場合)。同様に、構造体を１つのデータとしてリターンすることもできる。

【プログラム例２】ｘｙ座標系の２点Ｐ，Ｑの中点Ｍの座標を求めるプログラム

```
#include <stdio.h>
struct point {          /* グローバルに定義しているのでどの関数でも */
     float x, y;        /* この構造体を利用できる。                  */
};
int main( )
{
    struct point p, q, m;
    struct point middle(struct point, struct point); /* 中点を求める関数のプロトタイプ宣言 */

    printf("点Ｐの座標を入力してください。---> ");
    scanf("%f,%f", &p.x, &p.y);  /* x,y 座標をカンマで区切って入力 */
    printf("点Ｑの座標を入力してください。---> ");
    scanf("%f,%f", &q.x, &q.y);

    m=middle(p,q);
    printf("中点Ｍの座標は(%f,%f)です。¥n", m.x, m.y);
}

struct point middle(struct point a, struct point b)
{
    struct point c;
    c.x = (a.x+b.x)/2;
    c.y = (a.y+b.y)/2;
    return( c );
}
```

【参考】typedef による新しいデータ型の定義

前例の【プログラム例２】において typedef という機能を用いると構造体のような長い変数の宣言が以下のように簡潔になる。

```
#include <stdio.h>
struct point {
      float x, y;
};
typedef struct point POINT;          /* 新しいデータ型POINT の定義 */
int main( )
{
    POINT p, q, m;          /* 新しいデータ型POINT で変数を宣言 */
    POINT middle(POINT, POINT);

    printf("点Ｐの座標を入力してください。---> ");
    scanf("%f,%f", &p.x, &p.y);
    printf("点Ｑの座標を入力してください。---> ");
    scanf("%f,%f", &q.x, &q.y);

    m=middle(p,q);
    printf("中点Ｍの座標は(%f,%f)です。¥n", m.x, m.y);
}

POINT middle(POINT a, POINT b)
{
    POINT c;
    c.x = (a.x+b.x)/2;
    c.y = (a.y+b.y)/2;
    return( c );
}
```

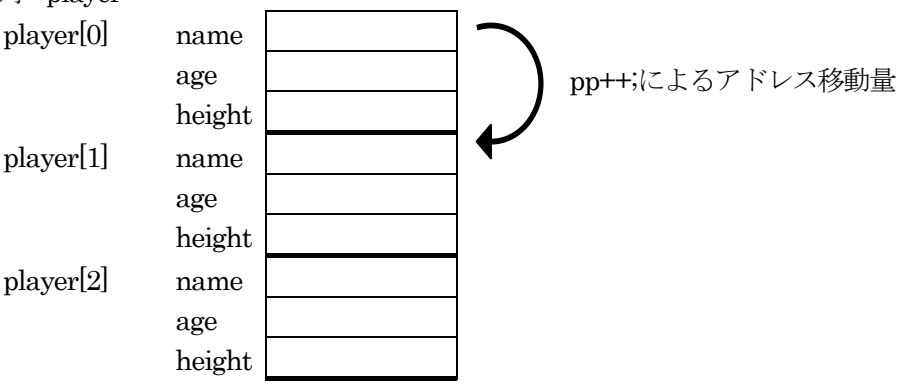
XII 構造体配列とポインタ

１．構造体配列とポインタ

構造体配列変数は、普通の配列変数と同様にメモリ内の連続した領域に確保される。そして、構造体配列名はその配列の先頭アドレスを示す定数となる。

```
struct person {
    char   name[20];
    int    age;
    float  height;
};        /* この構造体の定義はグローバルに行っておく。 */


struct person player[3], *pp;  /* これらはローカル変数で良い。 */
pp = player;        ←ポインタへの初期値代入
```

構造体配列　player



ポインタによる構造体メンバの間接参照

pp が player[0]の構造体を指していると時、その年齢や身長は

```
pp->age
pp->height
```

で参照される。

ここで、「－>」も「.」と同じく構造体演算子である。

※もともと、pp -> age は、*pp.age なのですが、“＊”と“.”の優先順位は“.”の方が高いので、
このまま記述すると、*(pp.age)ということに、なってしまいます。

pp -> age と同等にするためには、(*pp).age とする必要があります。
この記述は、ちょっと大変なので、(*pp).age と書く代わりに、pp->age とすることになっています。

【プログラム例１】名前・身長・体重を入力し、一覧表を作るプログラム。

```
#include <stdio.h>
#define  NREC 4
struct person {
        char   name[20];  /* 関数の外部で定義されているので、この */
        int    age;       /* テンプレートはすべての関数で共通に使 */
        float  height;    /* 用できる。                           */
};

int main( )
{
    int kbinput(struct person *p);
    void disp(struct person *p);
    struct person player[NREC], *pp;

     for(pp = player; pp < player+NREC; pp++)   pp -> name[0] = '¥0' ;

     for(pp=player; pp < player+NREC; pp++)
         if(kbinput(pp) == NULL) break;

     for(pp=player; pp < player+NREC; pp++) {
         disp(pp);
         if(pp->name[0] == '¥0') break;
     }
}

int kbinput(struct person *p)
{
    printf("Input data ----->¥n");
    if( scanf("%s%d%f", p->name, &p->age, &p->height) == EOF )
    return( NULL );
    else  return( 1 );
}

void disp(struct person *p)
{
    printf("¥t%-10s %4d %7.1f¥n", p->name, p->age, p->height);
}
```

【プログラム例２】英文を読み込み、そこに現れる英単語の出現頻度を調べ、結果を英単語のアルファベット順に出力するプログラム

```
#include <stdio.h>
#include <string.h>        /* 文字列関連の関数を使用するためのヘッダファイル */
struct word_data {                  /* 単語と出現頻度を１つにまとめる構造体 */
    char word[15];
    int freq;
};
typedef struct word_data ST_WORD;
int main( )
{
    ST_WORD words[50];
    char term[15];
    int i=0, nw=0, j, c;
    void sort_alpha(ST_WORD [ ], int);
                /* 指定された構造体配列をアルファベット順にソートする関数のプロトタイプ宣言 */
    while( (c=getchar( ))!=EOF ){
        if('A'<=c&&c<='Z') c+='a'-'A';
        if('a'<=c&&c<='z') term[i++]=c;     /* アルファベットなら保存 */
        else if(i>0){
            term[i]='¥0';           /* １単語の入力終了 */
            for(j=0; j<nw; j++)     /* 登録済みの単語かどうかの検査 */
                if(strcmp(term, words[j].word) ==0) break;
            if(j==nw){              /* 未登録単語の登録処理 */
                strcpy(words[j].word, term);
                words[j].freq=1;
                nw++;
            }else  words[j].freq++;         /* 登録単語のため頻度のみ更新 */
            i=0;
        }
    }
    sort_alpha(words, nw);          /* アルファベット順にソートする */
    for(j=0; j<nw; j++)             /* 単語と頻度の一覧表示 */
        printf("%-15s %3d¥n", words[j].word, words[j].freq);
}

void sort_alpha(ST_WORD tango[ ], int kosuu)
{
    int i, j;
    ST_WORD temp;

    for(i=0; i<kosuu-1; i++)
        for(j=kosuu-2; j>=i; j--)
            if( strcmp(tango[j].word,tango[j+1].word)>0 ){
                temp=tango[j];
                tango[j]=tango[j+1];
                tango[j+1]=temp;
            }
}
```

［演習 12］

学生の氏名、授業出席数（全 20 回）、遅刻数、早退数を読込み、その評価を出力するプログラムを以下の注意に従って作成せよ。（なお、プログラム例１のようにせずに、１人分ずつ読込み・処理・出力とすると簡潔になる）

・struct には、１人分の情報として必要なものをメンバーとして入れておくこと。
・struct の構造記述だけは、グローバルとして main より上に置くこと。
・学生は 100 人を最大として、配列に保存し、何人でも入れられるようにせよ。
・関数として作成するもの
（１）１人分を読込む関数（引数：１人分の情報）
ただし、出席数は 20 を越えないものとする。

遅刻、早退の場合も出席はしているものとする。

（２）評価を行う関数
評価方法：遅刻と早退は、３回で１回欠席と換算すること。
（３回未満は切り捨て）

Ａ：欠席３回以下
Ｂ：欠席７回以下
Ｃ：欠席１０回以下
Ｄ：欠席１１回以上

（３）１人分の情報を出力する関数

・なお、main で平均出席数と、平均欠席数を出力せよ。

データは以下のものを使用すること。

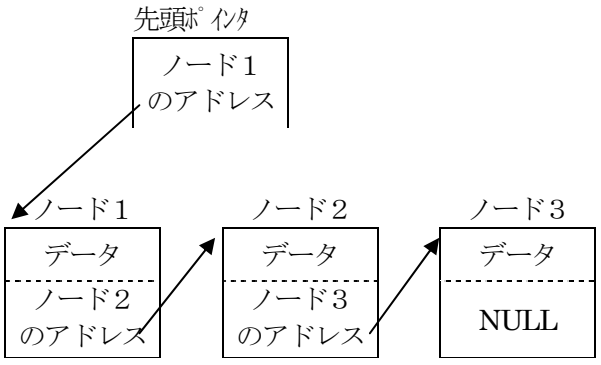
```
tanaka   19  2  1
toyoda   20  0  0
kanda    17  1  4
andoh    10  4  0
satoh    15  0  2
kakata    4  1  1
akita    18  5  2
watabe   16  2  4
kamata   16  0  0
furuta   18  3  3
```

XIII リスト構造と構造体

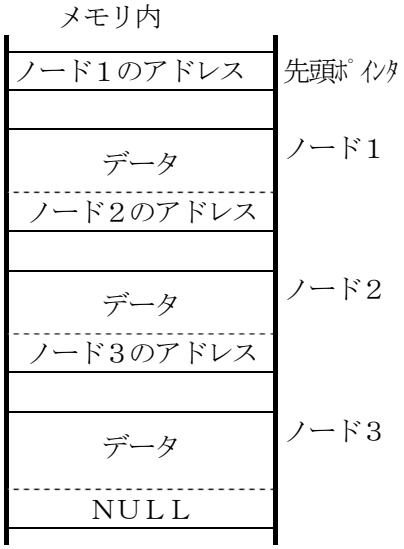
１．リスト構造とは

ポインタを用いて、次のデータ位置を指し示す構造のこと。

データのまとまりをノードをいう。



次のノードが無いことを示すために、NULL（0と同等）にする



２．構造体の再帰的記述

構造体のメンバーに同一構造体のポインタを持たせたもの。

ノード（データのまとまり）を例えば以下のように宣言する

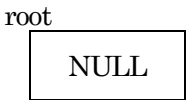
```
struct  record {
        int data;                       /* データ部 */
        struct  record  *rec;       /* ポインタ部 */
} ;
```

この様な構造体を自己参照構造体と呼ぶ。

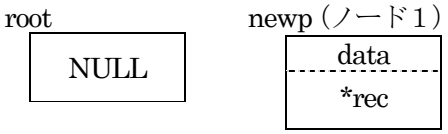
３．リスト構造を実現するには

```
struct  record  {
        int data;                       /* データ部 */
        struct  record  *rec;           /* ポインタ部 */
};
struct  record  *root,*newp;
```

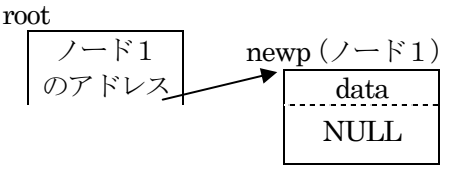
(1) root は最初のノードを指し示すポインタ　　(NULL（0 と同等）で初期化する)



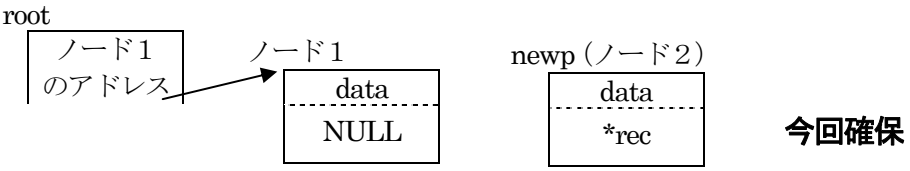
(2) malloc 関数など（ 次ページ参照 ）で、必要なノード（構造体）の領域を確保
新しく確保された領域（ノード１）のアドレスを newp とする



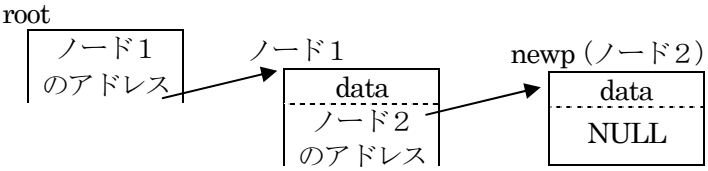
(3) 最初のポインタに新しく確保された構造体のアドレスを設定する



(4) malloc 関数などで、必要なノード（構造体）の領域を確保
新しく確保された領域（ノード２）のアドレスを newp とする



(5) ノード１のポインタ部にノード２のアドレスを設定する



(6) このようにすると配列になっていなくても次の構造体の場所がわかり、データの書込・参照が出来る
必要なときに、必要な領域を確保する

配列では、固定数確保していた！！！

(7) リストの最後は、（リストの終わりを検出できるように）NULL（0）とする

４．メモリ領域の動的な確保

すなわち、実行中に確保する
コンパイル時には、（個数がわかっている配列と違い）確保する領域の量はわからない
メモリの空いているところに、指定したバイト数分だけ割り当てられる
メモリが一杯で、領域を確保できないときには　NULL（0）が返る

①： malloc 関数

```
char  *malloc( int size )
```

指定サイズ（size バイト）のメモリ領域を取って、その先頭番地を返す関数。
領域は初期化されない。

②： calloc 関数

```
char  *calloc( int n, int size )
```

指定サイズ（size バイト）の n 個分のメモリ領域を取って、その先頭番地を返す関数。
このメモリは 0 に初期化される。

関数の使い方

返された番地は、適当な型に変換してから使用する

例１）

```
int  *p1, *p2 ;
p1＝ (int＊) malloc (sizeof (int)); /* 良い例 */
p2＝ (int＊) malloc (2) ;          /* 悪い例 */
```

例２）

```
struct   data  {
         int   a;
         char  b[20];
         double  c;
} x, y, *p1, *p2;

p1＝ (struct data *) malloc( sizeof(x) ) ;          /*  変数ｘのサイズ分 */
```

または

```
p1＝ (struct data *) malloc( sizeof(struct data) );   /*  struct data  1個分 */

p2＝ (struct data *) calloc( 3, sizeof(x));          /* 変数ｘ  3個分 */
```

または

```
p2＝ (struct data *) calloc( 3, sizeof(struct data));  /* struct data  3個分 */
```

sizeof の使い方

sizeof というのは演算子で、型名または変数名を（ ）内で書くと、そのマシンにおけるその型または変数が占めるバイトサイズに変換する。
これは、マシンによって同じ int でも占めるバイト数が違う為、じかに 2 バイトあるいは 4 バイトと指定して、プログラムの移植性を損なうことのないように用意されている。
できるだけこれを使用すること。

５．メモリ領域の解放

①： free 関数

```
void  free( void *p )
```

ｐが示すメモリ領域を解放する。
ｐは、calloc,malloc,realloc などで以前に割り当てられた領域のスペースへのポインタでなければならない。
ｐが NULL のときは、なにも起こらない。

②： free を忘れてても、main 関数終了時に free される。

６．正の整数値を読み込み、リスト構造にするプログラム

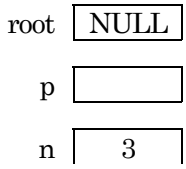
```
#include <stdio.h>
int  main( )
{
    struct node  {
         int num;
         struct node *np;
    };
         struct node  *root=NULL;   /* 先頭のポインタを初期化する */
         struct node  *newp;        /* 新しいノードのアドレス */
         struct node  *p;           /* 最後に追加したノードのアドレス／検索に使用  */
         int  no;                   /* 読み込んだデータ */

         printf(" 負の整数を入力するか、EOF で入力を終了します¥n");
         printf("     リストにするデータを入力して下さい --> ");

         while(1){
                  if ( scanf("%d",&no) == EOF ) break;        /* データの読み込み */                ……①
                  if ( no<0 ) {
                           printf("¥t¥t 負の数値が入力されたので終了します¥n");
                           break;
                  }
                  if ( ( newp=(struct node *)malloc( sizeof(struct node)) ) == NULL ) {
                           printf("¥n++++++++    cannot alloc area   +++++++++¥n¥n");
                           exit(1);
                  }
                  newp->num=no;                        /* データをノードのデータ部に保存 */
                  newp->np=NULL;                       /* ポインタ部を初期化 */              ……②
                  if ( root==NULL ) {                  /* まだリストになっていなかったら */        ……③
                           root=newp;
                           p=root;
                  }
                  else {   /* 最後に追加したノードのポインタ部に新しいノードのアドレスを保存 */
                           p->np=newp;                                                          ……④
                           p=p->np;                    /* 最後に追加したノードの更新 */    ……⑤
                  }
                  printf("     リストにするデータを入力して下さい --> ");
         }

         p=root;                                                /* 結果の出力 */
         printf("¥n*************  リストの結果を 出力します  **************¥n¥n");
         while ( p->np!=NULL ) {                                /* 次のノードがある間ループする */
                  printf("  %d  -->",p->num);
                  p=p->np;
         }
         printf("  %d¥n¥n",p->num);                    /* 最後のノードのポインタ部を出力 */
}
```

【実行結果】

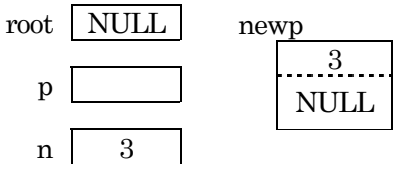
```
cserv% ex6
   負の整数を入力するか、EOF で入力を終了します
     リストにするデータを入力して下さい --> 3
     リストにするデータを入力して下さい --> 9
     リストにするデータを入力して下さい --> 0
     リストにするデータを入力して下さい --> 4
     リストにするデータを入力して下さい --> 6
     リストにするデータを入力して下さい --> 1
     リストにするデータを入力して下さい --> -5
                負の数値が入力されたので終了します

*************  リストの結果を 出力します  **************

  3  -->  9  -->  0  -->  4  -->  6  -->  1
cserv%
```

前述のプログラムのリストの動き

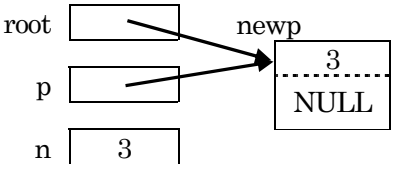
①



②

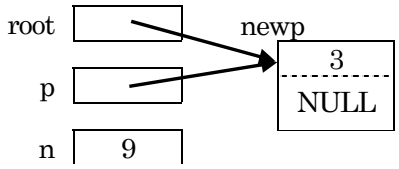


③

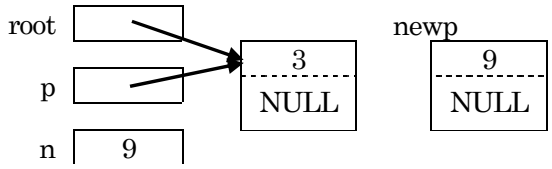


---

①

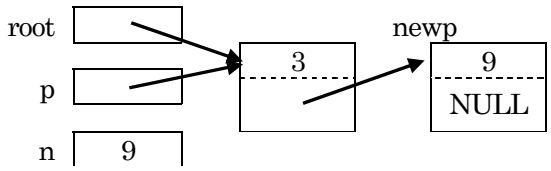


②

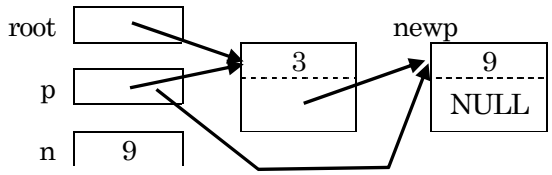


---

④

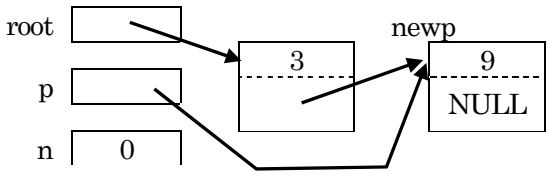


⑤

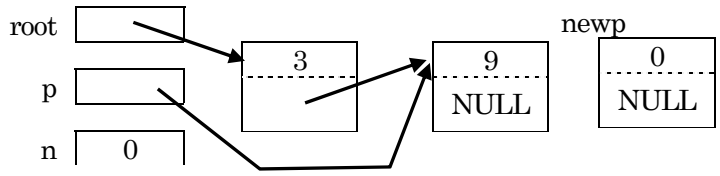


---

①

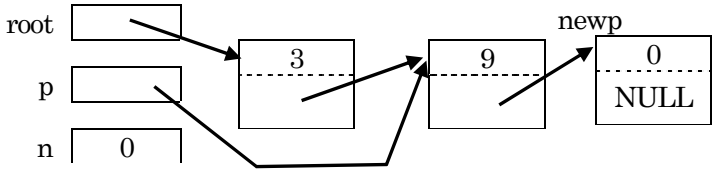


②



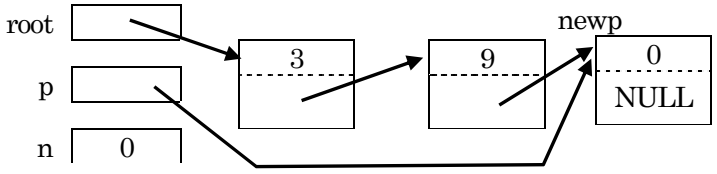
---

④



---

⑤



７．整数値を読み込み、リスト構造にするプログラム
　　リストの印刷を関数にし、構造体をグローバルに宣言

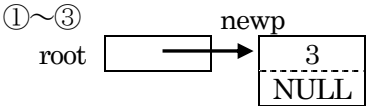
```
#include <stdio.h>
struct node {
        int num;
        struct  node  *np;
};
int  main( )
{
        int n;                                          /* 読み込むデータ */
        struct node *root=NULL;                         /* リストの先頭を NULL で初期化 */
        struct node *newp;                                      /* 新しく確保したノードのアドレス */
        struct node *p=NULL;                            /* リストの最後尾のノードのアドレス */
        void  printnode(struct node *);                 /* リストを出力する関数  */

        printf("  input number! --> ");
        while(scanf("%d",&n) != EOF) {
                newp = (struct node *) malloc (sizeof(struct node));
                if(newp == NULL) {
                        printf("malloc error¥n");
                        exit(1);
                }
                newp->num=n;                            /* 新しいノードにデータを保存 */
                newp->np=NULL;                          /* 新しいノードのデータ部を初期化 */

                if( p == NULL ) {                       /* まだリストが無かったら */
                        root=newp;                      /* 新しいノードを先頭ノードとする */
                        p=root;                         /* 最後尾のノード＝先頭ノード */
                }
                else {
                        p->np = newp;      /* 最後尾のノードのアドレス部に新しいノード野アドレスを設定 */
                        p=p->np;                        /* 最後尾のノードの更新 */
                }
                printf("  input number! --> ");
        }
        printnode(root) ;                               /* 先頭ノードを引数とする */
}

void  printnode(struct node *rootp)
{
        struct node  *p = rootp;

        printf("¥n *******  LIST= ");
        while( p != NULL ) {                            /* ノードがある間 */
                printf(" %d ",p->num);                  /* ノードのデータ部を出力 */
                p = p->np;                              /* ノードの更新 */
        }
        printf("¥n");
}
```

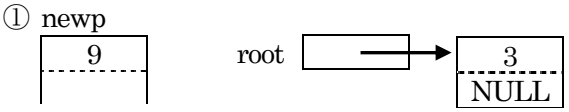
【実行結果】

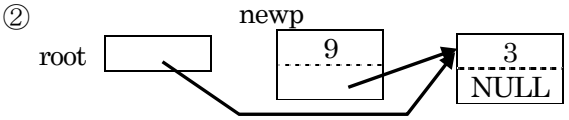
```
cserv% ex7
  input number --> 3
  input number --> 9
  input number --> 0
  input number --> 4
  input number --> 6
  input number --> 1
  input number --> CTRL+D

 *******  LIST=  3  9  0  4  6  1
cserv%
```

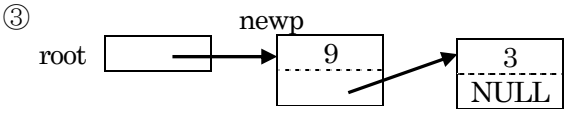
８．リストの先頭にノードを付け加えるプログラム

```
#include <stdio.h>
int  main( )
{
        struct node  {
                int num;
                struct node *np;
        };
        struct node  *root=NULL;          /* 先頭のポインタを NULL で初期化 */
        struct node  *newp;               /* 新しく確保したノードのアドレス */
        struct node  *p;                  /* 検索に使用  */
        int n;

        printf("  input number --> ");
        while(scanf("%d",&n) != EOF) {                          /* データの入力 */
        newp = (struct node *)malloc(sizeof(struct node));      /* 新しい領域の確保 */
                if( newp == NULL ) {                            /* 確保出来なかったら */
                        printf("malloc error¥n");
                        exit(1);
                }
                newp->num=n;      /* 新しいノードにデータを保存 */                          ……①
                newp->np=root;    /* 新しいノードのポインタ部を今までの先頭ノードにする */  ……②
                root=newp;        /* 先頭ノードの更新 */                                    ……③

                printf("  input number --> ");
        }

        p=root;                           /* 結果の出力  */
        printf("¥n¥n *******   LIST= ");
        while(p!=NULL) {
                printf(" %d ",p->num);
                p=p->np;
        }
        printf("¥n");
}
```



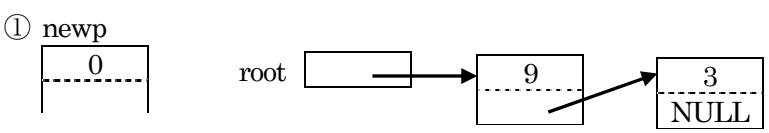


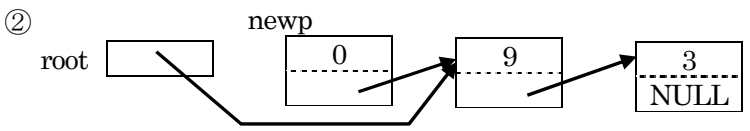


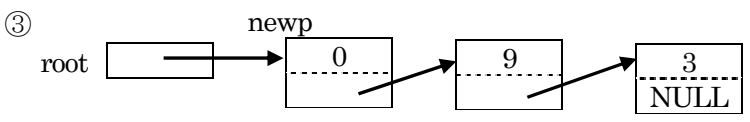


【実行結果】

```
cserv% ex7
  input number --> 3
  input number --> 9
  input number --> 0
  input number --> 4
  input number --> 6
  input number --> 1
  input number --> CTRL+D

*******   LIST=  1  6  4  0  9  3
serv%
```





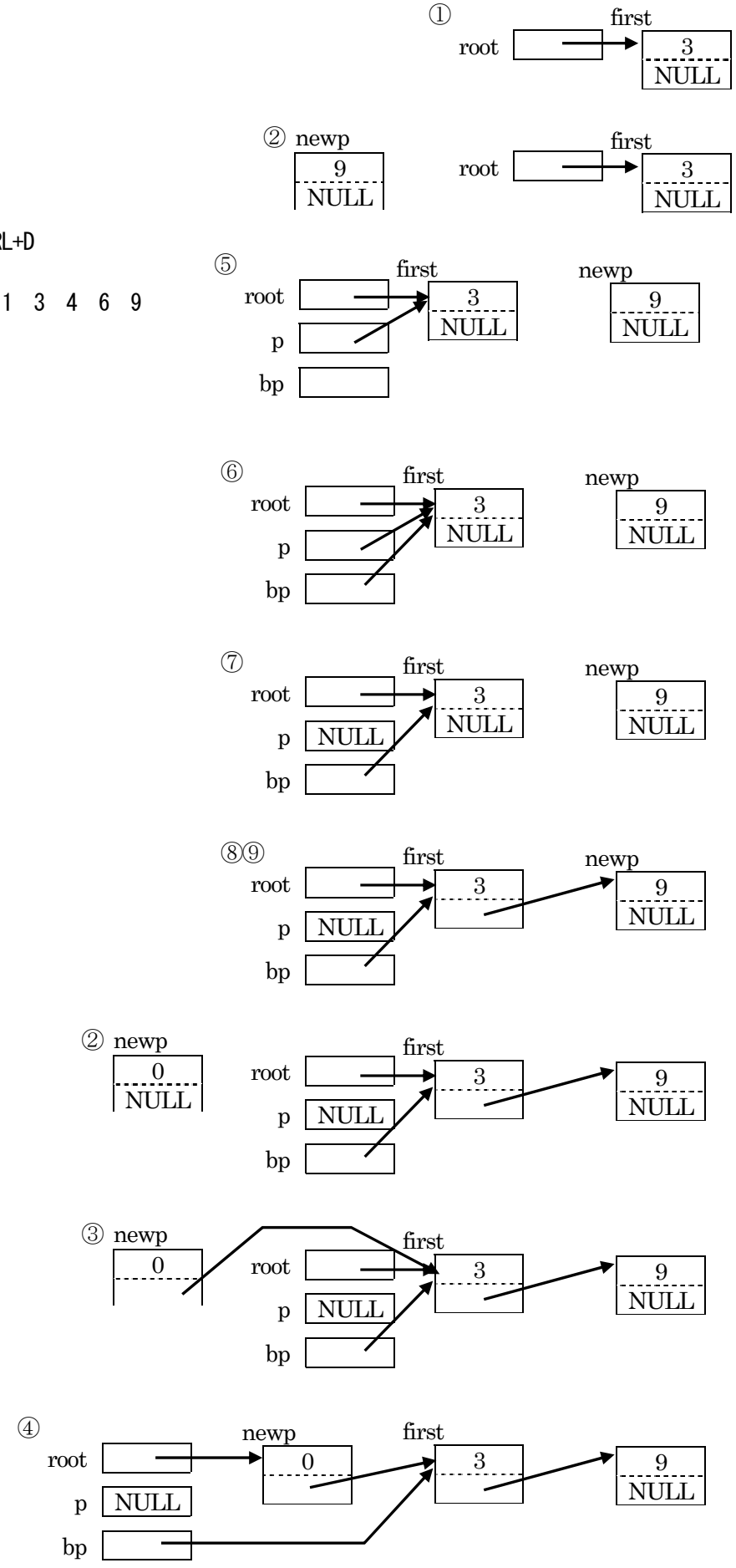


９．数字の小さい順にノードをつないでリストにするプログラム

```
#include <stdio.h>
struct node  {
        int num;
        struct node  *np;
};
int  main( )
{
        int n;
        struct node  first;  /* 最初のノードだけあらかじめ領域を取っておく */
        struct node  *root;  /* リストの先頭 */
        struct node  *newp;  /* 新しいノード */
        struct node  *p,*bp; /* 挿入する際に使用 */
        void printnode(struct node *);  /* リストを印刷する関数 */
        printf("  input number!  --> ");
        scanf("%d",&first.num);  /* 最初のノードについて */
        first.np=NULL;
        root=&first;   /* 先頭として取っておく */                              ・・・・・・・・①
        printf("  input number!  --> ");
        while(scanf("%d",&n) != EOF) {
                newp = (struct node *)malloc(sizeof(struct node));
                if(newp == NULL) {
                        printf("malloc error¥n");
                        exit(1);
                }
                newp->num=n;     /* データを新しいノードに保存 */
                newp->np=NULL;    /* 新しいノードのポインタ部を初期化 */       ・・・・・・・・②

                if(root->num > newp->num) {                  /* 新しいノードが先頭になる時 */
                        newp->np=root;    /* 今までの先頭を新しいノードのアドレス部に保存 */   ・・・・・・③
                        root=newp;                  /* 先頭の入れ換え */                    ・・・・・・④
                }
                else {                              /* リストの中間か最後に挿入 */
                        p=root;                                                             ・・・・・・⑤
                        while( p!=NULL && p->num <= newp->num) {
                                bp=p;  /* １つ前のノードを保存 */                           ・・・・・・⑥
                                p=p->np;  /* 次のノードのアドレスをセット */                ・・・・・・⑦
                        }
                        bp->np=newp;  /* １つ前のノードのポインタ部に新しいノードを保存 */    ・・・・・・⑧
                        newp->np=p;  /* 新しいノードのポインタ部に次のノードを保存 */        ・・・・・・⑨
                }
                printf("  input number! --> ");
        }
        printnode(root);
}
void  printnode(struct node *rootp)
{
        struct node *p = rootp;

        printf("¥n¥n   +++++++  LIST= ");
        while(p!=NULL) {
                printf (" %d ",p->num);
                p=p->np;
        }
        printf("¥n¥n");
}
```

【実行結果】

```
cserv% ex7
  input number --> 3
  input number --> 9
  input number --> 0
  input number --> 4
  input number --> 6
  input number --> 1
  input number --> CTRL+D

   +++++++  LIST=  0  1  3  4  6  9
```



１０．ノードの挿入・削除を含むリスト処理プログラム

実行時にコマンドを入力し、そのコマンドでデータの小さい順に挿入・削除・印刷・終了指示する。root (全体のノードの先頭アドレスを持つ) を変更する（可能性がある）関数では、root の更新値を返値としている

```
#include <stdio.h>
struct node {
        int num;
        struct node *np;
};
int  main( )
{
        struct node *root=NULL;    /* 先頭アドレスを初期化 */
        char com[10];              /* 挿入／削除／印刷／終了を指示するコマンド */
        int n;                     /* 読み込みするデータ */
        int flag=1;                /* flag が 0 の時 コマンド終了  */

        void printnode(struct node *);      /* 出力する関数 */
        struct node *insert(struct node *,int);     /* 挿入を行なう関数*/
        struct node *delete(struct node *, int);    /* 削除を行なう関数*/

        printf(" データを小さい順にリストにします。¥n");
        printf("¥t¥t¥t¥t¥t 挿入 : i¥n");
        printf("¥t¥t¥t¥t¥t 削除 : d¥n");
        printf("¥t¥t¥t¥t¥t 出力 : p¥n");
        printf("¥t¥t¥t¥t¥t 終了 : q¥n");
        printf("    のコマンドを入力して下さい¥n");
        while (flag) {
                printf("¥n input command ( i  or  d  or  p  or  q )! —> ");
                scanf("%s",com);
                switch (com[0]) {
                        case 'i':printf("¥n¥t¥t リストにするデータを入力して下さい  —>  ");
                                 scanf("%d",&n);            /* 挿入 */
                                 root=insert(root,n);break;
                        case 'd':printf("¥n¥t¥t 削除するデータを入力して下さい  —>  ");
                                 scanf("%d",&n);            /* 削除 */
                                 root=delete(root,n);break;
                        case 'p':printnode(root);break;     /* 出力 */
                        case 'q':flag=0;break;              /* 終了 */
                        default: printf("¥n¥t¥t ***** command input error*****¥n");
                }
        }
}

void  printnode(struct node *rootp)
{
        struct node *p = rootp;

        printf("¥n¥n   ++++++++   LIST = ");
        while(p!=NULL) {
                printf (" %d ",p->num);
                p=p->np;
        }
        printf("¥n");
}
```

```
struct node *insert( struct node *root,int n)        /* 挿入を行なう関数 */
{
        struct node *newp,*p,*bp;

        newp=(struct node *)malloc(sizeof(struct node));
        if(newp==NULL) {
                printf("malloc error¥n");
                exit(1);
        }
        newp->num=n;                /* 新しいノードにデータを保存 */
        newp->np=NULL;              /* 新しいノードのアドレス部を初期化 */

        if ( root==NULL ) return(newp);     /* リストが空なら先頭ノードを新しいノードのアドレスにする */
        else {
                if(root->num > newp->num) {         /* 先頭ノードとして挿入する */
                        newp->np=root;
                        root=newp;
                        return(newp);               /* 先頭ノードを新しいノードにする */
                }
                else {                      /* リストの中間または最後に挿入 */
                        p=root;             /* 新しいノードを挿入するところを探索 */
                        while(p!=NULL && p->num <= newp->num) {
                                bp=p;    /* １つ前のノードを bp に保存 */
                                p=p->np;
                        }
                        bp->np=newp;        /* １つ前のノードのポインタ部を新しいノードのアドレスにする */
                        newp->np=p;         /* 新しいノードのポインタ部を次のノードのアドレスにする */
                        return(root);       /* 先頭のアドレスを返す */
                }
        }
}

struct node *delete(struct node *root,int n)
{
        struct node *p,*bp;
        if(root==NULL) {            /* リストがまだ空の時 */
                printf("¥nlist is empty!!!¥n¥n");
                return(root);
        }
        if( root->num==n ) {        /* リストの先頭ノードを削除する */
                p=root->np;                 /* ２番目のノードのアドレスを保存 */
                free(root);                 /* 先頭のノードを解放 */
                printf("¥n          %d is deleted¥n" ,n);
                return(p);                  /* 新しい先頭のノードとしてリターン */
        }
        else {   /* 削除が中間または最後の時 */
                p=root;
                while(p!=NULL && p->num!=n) {       /* 削除指定された数字をリストの最初から検索*/
                        bp=p;    /* １つ前のノードを保存 */
                        p=p->np;
                }
                if(p==NULL) {    /* リストに対応する数字が無い場合 */
                        printf("¥n          %d is not found in list¥n",n);
                }
                else {                      /* 削除するノードがある時 */
                        printf("¥n          %d is deleted¥n" ,n);
                        bp->np=p->np;   /*１つ前のノードのポインタ部に次のノードのアドレスを代入 */
                        free(p);            /* ノードを削除 */
                }
                return(root);       /* 先頭の更新は無し */
        }
}
```

【実行結果】

```
cserv% ex10
 データを小さい順にリストにします。
                                          挿入 : i
                                          削除 : d
                                          出力 : p
                                          終了 : q
    のコマンドを入力して下さい

 input command ( i  or  d  or  p  or  q )! --> i
                リストにするデータを入力して下さい  -->  3
 input command ( i  or  d  or  p  or  q )! --> i
                リストにするデータを入力して下さい  -->  9
 input command ( i  or  d  or  p  or  q )! --> i
                リストにするデータを入力して下さい  -->  0
 input command ( i  or  d  or  p  or  q )! -->
                リストにするデータを入力して下さい  -->  4
 input command ( i  or  d  or  p  or  q )! --> i
                リストにするデータを入力して下さい  -->  6
 input command ( i  or  d  or  p  or  q )! --> i
                リストにするデータを入力して下さい  -->  1
 input command ( i  or  d  or  p  or  q )! --> p

   ++++++++   LIST =  0  1  3  4  6  9


 input command ( i  or  d  or  p  or  q )! --> d
                削除するデータを入力して下さい  -->  11
         11 is not found in list

 input command ( i  or  d  or  p  or  q )! --> d
                削除するデータを入力して下さい  -->  4
         4 is deleted

 input command ( i  or  d  or  p  or  q )! --> p

   ++++++++   LIST =  0  1  3  6  9

 input command ( i  or  d  or  p  or  q )! --> d
                削除するデータを入力して下さい  -->  0
         0 is deleted

 input command ( i  or  d  or  p  or  q )! --> p

   ++++++++   LIST =  1  3  6  9

 input command ( i  or  d  or  p  or  q )! --> x
                 ***** command input error*****

 input command ( i  or  d  or  p  or  q )! --> q
cserv%
```

[演習 13] 10 個の数字をリスト構造に作成せよ。なお、偶数値がリストの前半、奇数値がリストの後半になるようにせよ。勿論、入力値、結果出力もして下さい。

なお、売り物のシステムを作る時には、動的メモリの割り当て（malloc( ),calloc( )）を使用してプログラムを作成することはありません。malloc, free を多用することによって、穴ぼこだらけになる上、実行途中でメモリー不足によって止まる危険があります。

使用する場合は、次のようにする。

（1）プログラムの最初に、使用する全ての領域を、malloc()で確保する。（参考：付録２）
（2）これを、未使用リストとして全てをつなげておく。
（3）実際に使用する時は、その未使用リストから外して使用する。
（4）不要になったら、また未使用リストにつないでおく。
（5）もし、沢山使用して、未使用リストが無くなった時は、エラーメッセージを出して終了させる。

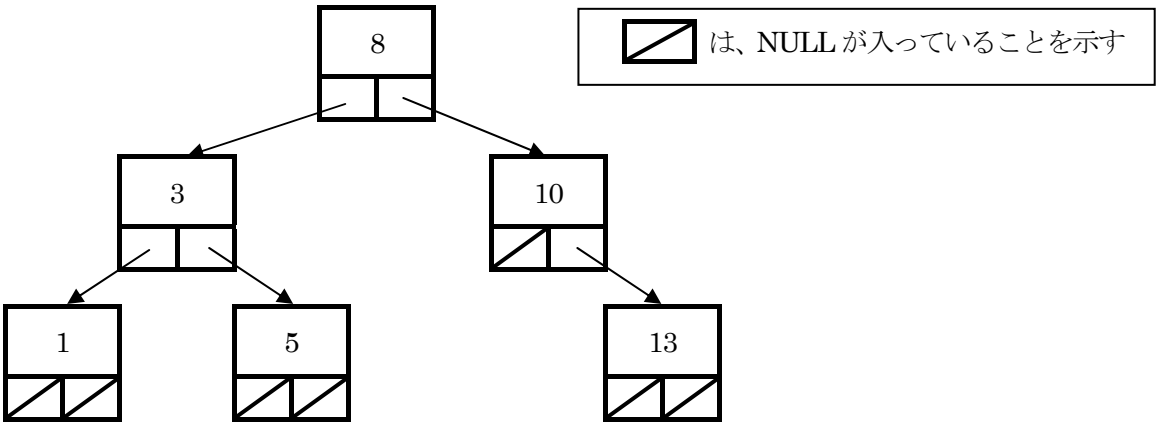
[演習 14]（6．正の整数値を読み込み、リスト構造にするプログラム）を修正して、上の手法になるように、プログラムを書き直せ。

XIII 構造体と２進木(binary tree)

２進木によるデータの整理の仕方を具体例を用いて説明する。

例：任意の数（EOF まで）入力された整数を、小さい順に出力するプログラムを考える。　　入力されるデータは下図のような２進木を作って保持される。２進木の各ノードは、整数格納のための場所と、このノードに入っている整数より大きい整数の入っているノードを指すポインタ、およびこのノードより小さい整数の入っているノードを指すポインタとから成る構造体で定義される。

入力された整数が 8, 3, 10, 1, 13, 5 の場合。



入力を終了すると２進木をたどって整数の小さい順から表示することになる。
２進木へのノードの追加や、表示は再帰を利用すると見やすいプログラムが書ける。以下にプログラムの一例を挙げる。

```
#include <stdio.h>
#define NoError 1
#define Error   0
struct binnode {
         int  num;
         struct binnode *left, *right;
 };

typedef struct binnode NODE;

int main( )
{
     int flag;
     NODE *top, *node;
     void bintree(NODE *top, NODE *node);
     void disp(NODE *top);

     if( (top=(NODE *)malloc(sizeof(NODE))) == NULL ) {
          printf("構造体が確保できません¥n");  exit(1);
     }
     scanf("%d", &top->num);  top->left = top->right = NULL;

     flag = Error;
     while( node = (NODE *)malloc(sizeof(NODE)) ){
          if( scanf("%d",&node->num) == EOF ) {flag = NoError; break;}
          node->left = node->right = NULL;
          bintree(top,node);
     }
     if( flag == Error ) {
          printf("構造体が確保できません¥n");  exit(1);
     }
     disp(top);  printf("¥n");
}
```

```
void bintree(NODE *top, NODE *node)
{
     if( node->num < top->num ) {
          if( top->left == NULL ) top->left = node;
                         else  bintree(top->left, node);
     }
     else {
          if( top->right == NULL ) top->right = node;
                         else  bintree(top->right, node);
     }
}

void disp(NODE *top)
{




}
```

練習：上の disp という関数を再帰を用いて作成せよ。

[演習 15]

例題にあるプログラムを参考に、構造体に頻度統計用の int i; を１行追加し、重複した数値が存在した時には、ノードを作成するのではなく、ｉに１を加え、出現頻度をとり、出力するようにプログラムを修正せよ。

ただし、入力データとしては正の整数だけとし、ＥＯＦが来た時に読込みを終了するように作成せよ。

（データ例）

8　3　3　10　8　5　5　13　3　1　5　3

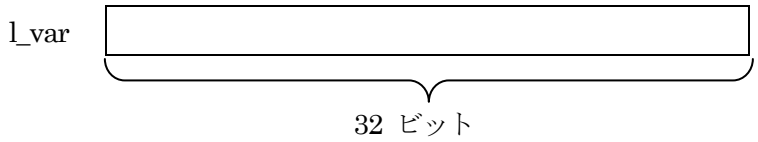
（出力例）

1	1
3	4
5	3
8	2
10	1
13	1

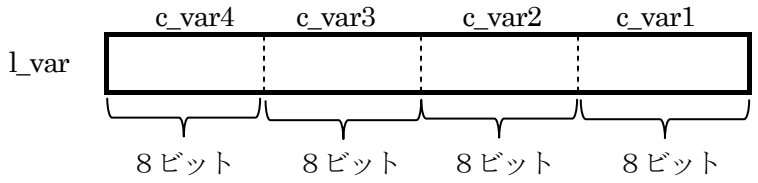
XIV 共用体

１．共用体とは・・・・・

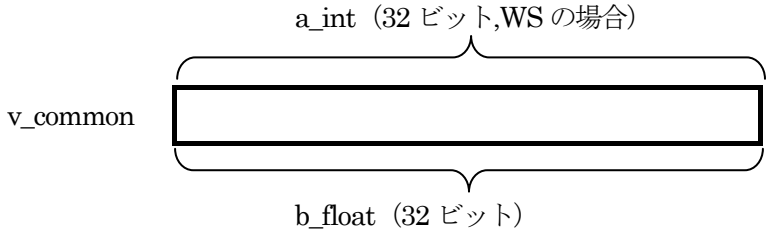
Ｃで宣言される変数はそれぞれ異なるアドレスを与えられ、同じ記憶領域（つまりアドレス）に複数の変数が同居することはない。しかし、共用体はまさしくこれを可能にするものである。例えば、１個の long 型変数 l_var が宣言されているとする。



上図のように l_var は long 型であるから、１つの変数の記憶領域として 32 ビットを確保しており、１つのアドレスを持っている（使用している領域の一番小さいアドレス）。
ところで、この 32 ビットの領域は、8 ビットの 4 つの領域がつながった大きさであると見ることもできる。そこである時には long 型の変数ではなく char 型の変数 4 つ分として使用することが可能となる。すなわち、



ということであり、l_var と c_var1 のアドレスは等しくなる。
　ただし、これは long 型の変数 l_var と char 型の変数 c_var1～4 が共存できるということを意味しているのではない。互いに１つの領域を異なる変数のために排他的に利用できるということを意味している。
　この共用体が活用できるのはどのような場合かというと、例えば、「ある数値を読込んでその数値を２倍する」という簡単なプログラムを考える。一見簡単そうに見えるが、この問題で数値が実数か整数かが不明であるとすると、int, float どちらの数値がきてもいいように対処しなければならず、そう単純ではない。本来このプログラムでは読み込む数値は１つだけなのだから、そのための変数（領域）は１つあるのが自然である。しかし、実際、今までの知識の範囲では、読み込む値のデータ型が int か float かによって、それぞれ異なる２つの変数に読み込まなければならなくなる。そこで共用体が役に立つことになる。１つの変数（例えば v_common）を宣言するだけで、その変数を場合に応じて２つの変数（例えば int 型の a_int と long 型の b_float）の好きな方で使えることになる。この場合のこの共用体変数の構造を示すと次の通りである。



２．共用体の定義

共用体の定義の仕方は、構造体のテンプレートの宣言の方法と非常によく似ている。

定義形式

```
union  共用体タグ名 {
         メンバ１;
         メンバ２;
           …
         メンバｎ;
};
```

具体例（前述の v_common のための定義）

```
union tag_com {
        int   a_int;
        float b_float;
};
```

共用体のメンバはすべて同一のアドレスを持つことになる。従って、次に述べる共用体変数の宣言で実際に確保されるメモリサイズは、メンバの中で最大のサイズと等しくとられる。

３．共用体変数の宣言

宣言形式

union　共用体タグ名　変数名[,変数名,･･･];

具体例

```
union tag_com v_common;
```

共用体の定義とそのような変数の宣言は以下のように同時に行うこともできる。

定義と宣言の形式

```
union tag_com {          このように直に変数の宣言まですると きには、
        int   a_int;     共用体タグ名（tag_com）は省略可能である。
        float b_float;
} v_common;
```

４．共用体メンバの参照方法

構造体と同様に直接参照とポインタを用いた間接参照とがある。

直接参照の仕方の例（３．で宣言した v_common を例に挙げる）

```
v_common.a_int
v_common.b_float
```

間接参照の仕方の例

```
union tag_com v_common, *pp;  という宣言がされているとき、
pp = &v_common; ならば
  pp->a_int あるいは pp->b_float で参照できる。
```

５．共用体を用いたプログラミングの例

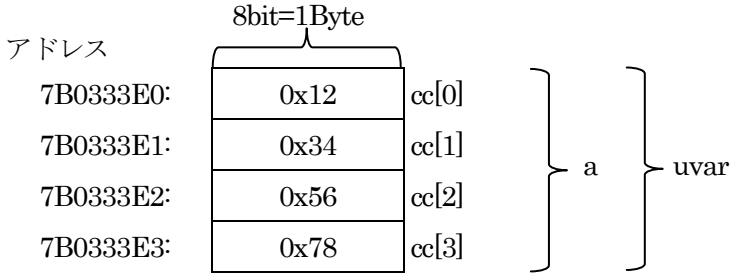
【１】int 型のサイズの領域に char 型を重ねてそれらのアドレスや格納値を調べる

```
#include <stdio.h>
union tag_union {
    int a;
    char cc[4];
};

int main( )
{
    union tag_union uvar;

    uvar.a = 0x12345678;

    printf("Address of uvar    : %X¥n", &uvar);
    printf("Address of uvar.a  : %X¥n", &uvar.a);
    printf("Address of uvar.cc[0]: %X¥n", &uvar.cc[0]);
    printf("Address of uvar.cc[1]: %X¥n", &uvar.cc[1]);
    printf("Address of uvar.cc[2]: %X¥n", &uvar.cc[2]);
    printf("Address of uvar.cc[3]: %X¥n", &uvar.cc[3]);

    printf("INT  = %X¥n", uvar.a);
    printf("CHAR = %X %X %X %X¥n", uvar.cc[0], uvar.cc[1], uvar.cc[2], uvar.cc[3]);
}
```

実行結果（Hewlett Packard の WS 上での例）

```
Address of uvar    : 7B0333E0
Address of uvar.a  : 7B0333E0
Address of uvar.cc[0]: 7B0333E0
Address of uvar.cc[1]: 7B0333E1
Address of uvar.cc[2]: 7B0333E2
Address of uvar.cc[3]: 7B0333E3
INT  = 12345678
CHAR = 12 34 56 78
```

上の実行結果より、このプログラムのデータ構造を図示すると以下のようになる。



（参考）この実行例で、int 変数である uvar.a の上位桁の方が小さいアドレスに格納されたことを表している（これをビッグエンディアンという）。

[演習 16] 上の【１】のプログラムを実行して考察せよ。
結果が異なるのですが、どういうことが考えられるかを考察せよ。

【２】int 型と float 型を同じメモリ領域にとって使い分ける例

```
#include <stdio.h>

union two_in_one {
    int a;
    float n;
};

int main( )
{
        union two_in_one single_var;

        printf("Please input an integer ==> ");
        scanf("%d",&single_var.a);
        printf("Integer single_var/3=%d¥n", single_var.a/3);

        single_var.n = single_var.a;
        printf("Float   single_var/3=%.5f¥n", single_var.n/3);
}
```

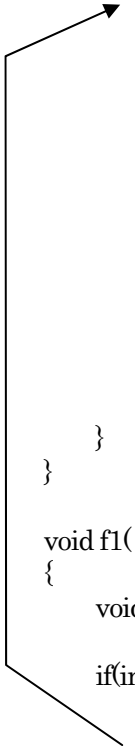
実行結果

```
  Please input an integer ==> 16
  Integer single_var/3=5
  Float   single_var/3=5.33333
```

（付録１）関数間ジャンプ

エラーの処理後などで、必ず同じ場所に戻って行くような時に便利です。
setjmp( )で戻って来る場所を指定し、longjmp( )で跳ぶ。
ヘッダーファイルとして、#include <setjmp.h>が必要です。

［プログラム例］



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

jmp_buf env;     //gloval
int inval;

int main( )
{
      int setjmpval;
      void f1( );

      while(1){
            setjmpval=setjmp(env);

            printf("now land on main() from f%d( )¥n¥n", setjmpval);
            printf("if initial or illegal val, value is 'f0'¥n");

            printf("please input (1-6) (-1:end):");
            scanf("%d",&inval);
            if(inval = = -1) exit(1);
            if(inval < 1 | | inval >6){
                  printf("specified illegal val '%d----------'¥n",inval);
                  continue;
            }
            f1( );
      }
}

void f1( )
{
      void f2( );

      if(inval = = 1){
            printf("jump from f1( ) to main( )¥n");
            longjmp(env,1);
      }
      f2( );
}

void f2( )
{
      void f3( );

      if(inval = = 2){
            printf("jump from f2( ) to main( )¥n");
            longjmp(env,2);
      }
      f3( );
}

void f3( )
{
```

```
        void f4( );

        if(inval = = 3){
            printf("jump from f3( ) to main( )¥n");
            longjmp(env,3);
        }
        f4( );
}

void f4( )
{
        void f5( ),f6( );

        if(inval = = 4){
            printf("jump from f4( ) to main( )¥n");
            longjmp(env,4);
        }

        if(inval = = 5) f5( );
        else f6( );
}

void f5( )
{
        printf("jump from f5( ) to main( )¥n");
        longjmp(env,5);
}

void f6( )
{
        printf("jump from f6( ) to main( )¥n");
        longjmp(env,6);
}
```

［実行例］

```
if initial or illegal val, value is 'f0'
please input (1-6) (-1:end):1
jump from f1( ) to main( )
now land on main( ) from f1( )

if initial or illegal val, value is 'f0'
please input (1-6) (-1:end):2
jump from f2( ) to main( )
now land on main( ) from f2( )

if initial or illegal val, value is 'f0'
please input (1-6) (-1:end):7
specified illegal val '7----------'
now land on main( ) from f0( )

if initial or illegal val, value is 'f0'
please input (1-6) (-1:end):5
jump from f5( ) to main( )
now land on main( ) from f5( )

if initial or illegal val, value is 'f0'
please input (1-6) (-1:end):-1
```

（付録２）Ｃプログラミングテクニック

例えば、２次元配列の足し算を行う場合、通常以下のようになります。(説明が簡単になるように正方行列)

```
#define   MAXM   100
int main( )
{
     int   matrixA[MAXM][MAXM], matrixB[MAXM][MAXM], matrixC[MAXM][MAXM];
     int   n, i, j;

     printf("読込む行列の大きさを入力せよ：");
     scanf("%d", &n);
     if(n>=MAXM){
          printf("読込む行列の大きさの制限 %d  を越えています。¥n", MAXM);
          exit(1);
     }
     for( i=0; i< n; i++)
          for( j=0; j< n; j++)   scanf("%d", &matrixA[i][j]);
     for( i=0; i< n; i++)
          for( j=0; j< n; j++)   scanf("%d", &matrixB[i][j]);

     for( i=0; i< n; i++)
          for( j=0; j< n; j++)   matrixC[i][j] = matrixA[i][j] + matrixB[i][j];

     for( i=0; i< n; i++){
          for( j=0; j< n; j++)   printf("%5d", matrixC[i][j]);
          printf("¥n");
     }
}
```

上の例では、100×100 の領域を３つ確保するので、メモリの無駄が多い。
下の例は、malloc( )を使用して動的に領域を確保して、メモリ効率を良くしてプログラミングした例です。

ポインターを使用しているのですが、あたかも２次元配列を使用しているかのようにしています。

ファイル名 "myalloc.h"　　ユーザ定義ヘッダーは "" で行う。

```
#define   INTALLOC   10000          // int の領域を 10000 個確保する。
```

```
#include <stdio.h>
#include "myalloc.h"          /*  ユーザ定義ヘッダーで、確保する領域の大きさが入っている  */

#define MATA(x,y)     *(matrixAp+(x)*NN+(y))       /*これ以降の部分を置き換える。( )内は引数のように使える*/
#define MATB(x,y)     *(matrixBp+(x)*NN+(y))
#define MATC(x,y)     *(matrixCp+(x)*NN+(y))

int   *int_head, *int_headsave;
int    *matrixAp, *matrixBp, *matrixCp;
int    NN;

int main( )
{
     int   i, j;

     int_headsave = int_head = (int *) malloc(INTALLOC*sizeof(int));          /*  通常、型別に確保する  */
                                                                    /*全ての配列を一度に確保してしまう。*/
     if(int_head = = NULL){
```

```
        printf("領域が確保できません。");
    }

    printf("読込む行列の大きさを入力せよ：");
    scanf("%d", &NN);

    if(NN*NN*3 > INTALLOC) printf("確保すべきメモリが取れません。");

    matrixAp = int_head;
    int_head += NN * NN;
    matrixBp = int_head;
    int_head += NN * NN;
    matrixCp = int_head;

     for( i=0; i< NN; i++)
        for( j=0; j< NN; j++)          scanf("%d", &MATA(i,j) );
    for( i=0; i< NN; i++)
        for( j=0; j< NN; j++)          scanf("%d", &MATB(i,j) );

    for( i=0; i< NN; i++)
        for( j=0; j< NN; j++)          MATC(i,j) = MATA(i,j) + MATB(i,j);

    for( i=0; i< NN; i++){
        for( j=0; j< NN; j++)          printf("%5d", MATC(i,j));
        printf("¥n");
    }
}
```

例えば、

```
     for( i=0; i< NN; i++)
        for( j=0; j< NN; j++)          scanf("%d", &MATA(i,j) );
```

は、上の#define **MATA(x,y)　*(matrixAp+(x)*NN+(y))** で展開されて、

```
     for( i=0; i< NN; i++)
        for( j=0; j< NN; j++)          scanf("%d", &*(matrixAp+(i)*NN+(j)) );
```

となる。

（付録３）#if　#else　#endif

ある条件に従って、プログラムリストの部分をコンパイルするかどうか決めることができます。

```
#if　定数式
　　プログラムリスト１
#else
　　プログラムリスト２
#endif
```

定数式が真であれば、プログラムリスト１がコンパイルされ、偽ならばプログラムリスト２がコンパイルされる。

```
#define　PLUS　1

int main( )
{
　　int i, kosu, data, sum=0;

　　printf(“個数を入力して下さい : ”);
　　scanf(“%d”, &kosu);
　　printf(“データを%d 個入力して下さい”, kosu);
　　for(i=0; i<kosu; i++){
　　　　scanf(“%d”, &data);
#if　PLUS
　　　　sum += data;
#else
　　　　sum -= data;
#endif
　　}
#if　PLUS
　　printf(“合計は、%d です。”, sum);
#else
　　printf(“マイナスした合計は、%d です。”, sum);
#endif
}
```

```
#define　PLUS　1　の時のプログラム

　　for(i=0; i<kosu; i++){
　　　　scanf(“%d”, &data);
　　　　sum += data;
　　}
　　printf(“合計は、%d です。”, sum);
```

```
#define　PLUS　0　の時のプログラム

　　for(i=0; i<kosu; i++){
　　　　scanf(“%d”, &data);
　　　　sum -= data;
　　}
　　printf(“マイナスした合計は、%d です。”, sum);
```

Ｃプログラム開発時によく用いられる手法

これは、デバッグの時だけ処理途中の変数の値を表示したりする時に使う。まず、

```
#define　DEBUG　1
```

という宣言をする。次に必要な部分で、

```
#if　DEBUG
　　printf(“x=%d　　y=%d　　z=%d¥n”, x, y, z);　#endif
```

としておくと、デバッグ中だけ、変数 x, y, z の値を出力する。
デバッグが完了したら、define 文を

```
#define　DEBUG　0
```

と変更するだけで作業が終わる。プログラムの削除をする必要もなく、後にバグが発見されて、再度デバッグする時に便利です。

（付録４）プログラムの実行時間の計測

プログラム中の任意の部分のかかる計算時間を計測したいときには、time.h というヘッダーファイル include をし、以下のように行う。

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

int main( )
{
        long start, stop;

        start = time( (long*)NULL );
        ┌──────────┐
        │  部分A   │
        └──────────┘
        stop = time( (long*)NULL );
        printf("Elapsed time %ld sec.¥n",stop-start);   ← 部分Aに要した時間が表示される（秒単位）。
        ┌──────────┐
        │  部分B   │
        └──────────┘
        stop = time( (long*)NULL );
        printf("Elapsed time %ld sec.¥n",stop-start); ← 部分Aと部分Bに要した時間が表示される（秒単位）。
}
```

要するに計測したい部分を time 関数ではさんでそれぞれの時間を記憶しておき、その差を取れば良いのである。ただし、秒単位の計測なので、あまり時間的に短い部分の計測をしても 0 秒という結果が出るだけである。

【具体例】下のプログラムは、内側の単なる二重ループの所用時間を表示していく。

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

int main( )
{
        long i,j,k,start,stop;

        start = time((long*)NULL);
        for(i=0; i<10; i++){
                for(j=0; j<10000; j++){ for(k=0; k<80; k++); }
                stop =time((long*)NULL)-start;
                printf("Time %ld sec.¥n",stop);
        }
}
```

【シェルソートの原理】

シェルソートは 1959 年に D.L.Shell により開発されたアルゴリズムである。隣接交換法では、配列の最前部近くにあるべきデータが最後尾近くにあるとすると、適切な位置に移動するまでに多くの交換が行われる可能性がある。そこで、隣接データと比較するのではなく、比較的離れた位置のデータから順に比較して交換していくことにより、大量の無秩序性を素早くなくすことができるということに着目したのがシェルソートである。離れた距離のデータを比較・交換していくので、一回では整列を終了することはできない。従って、比較交換するデータとデータの距離を少しづつ減らしていき、距離が 1 になったら、すなわち、隣接交換になったら終わることになる。ただし、隣接交換をするといっても、データの無秩序性は初期の段階でかなりなくされているので、実際には比較は行われても交換が行われることは少なくなる。

今、n 個の要素を持つ配列 int data[n]の内容をソートすることを考える。シェルソートで比較交換する 2 つのデータの距離を w とすると、プログラムは以下のようになる。

```
for(w=n/2; w>0; w/=2)
      for(i=w; i<n; i++)
  for(j=i-w; j>=0 && data[j]>data[j+w]; j-=w) {
                  temp = data[j];
          data[j] = data[j+w];
          data[j+w] = temp;
  }
```