プログラミング１　第９回
構造体(2)・応用

- 構造体へのポインタ
- 構造体のネスト
- 関数と構造体ポインタ
- 良くあるミス

この資料にあるサンプルプログラムは
`/home/course/prog1/public_html/2007/HW/lec/sources/`
下に置いてありますから、各自自分のディレクトリに
コピーして、コンパイル・実行してみてください

構造体へのポインタ(1)

- 構造体も変数ですから、そのポインタは以下のように宣言できる:

```
struct 構造体タグ名 *ポインタ変数名;
（例） struct roll *p;
```

- 構造体のアドレス参照は、以下の形式で行なう。

```
&構造体変数名
```

- 構造体ポインタへのアドレスの格納は、従来のポインタ処理と同じである。即ち、以下のように行う

```
構造体ポインタ変数 ＝ &構造体変数名 ;
（例） p = &my_data;
```

この代入以降、p には my_dataのアドレスが保持される。

構造体へのポインタ(2)

- 構造体ポインタpがある時に、*pでそのポインタが指し示す構造体の内容を得ることが出来る。
  - 「*」を間接演算子と呼ぶ
- 構造体メンバーをポインタでアクセスする場合には、

```
(*p).name  (*p).birth  (*p).address
```

のように書く。**p = &my_data** である時、これは

```
my_data.name my_data.birth
my_data.address
```

と同じ意味になる。

何故(*p)のように括弧が必要かと言うと、「.」演算子が「*」より優先順位が高いためで、*p.nameとすると、*(p.name) の事になってしまうからである。（ちなみにこれはコンパイルエラーとなる。何故ならpは構造体変数ではなく、構造体roll型のポインタであるから、その後ろに「.」で続けてメンバー名が来るのはあり得ないからである）

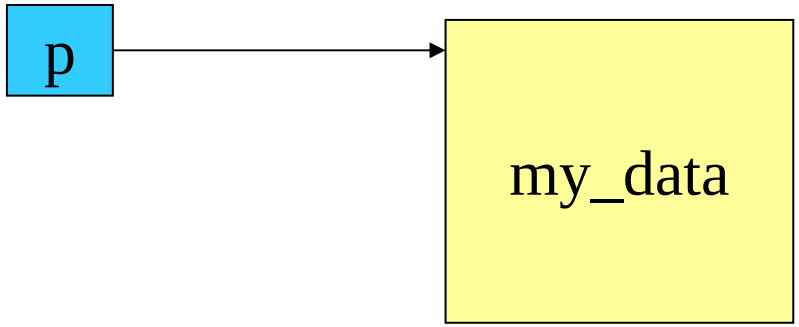
構造体へのポインタ(3)

- メンバー参照の2つ目の方法は、アロー演算子「->」を用いて、以下の形式で行なう。（‘ー’と‘＞’続けて書く）

構造体ポインタ変数名->メンバ名

- `p = &mydata` である時以下の3つは同一の物である。

```
p->name
(*p).name
my_data.name
```



構造体へのポインタ(4)

- 構造体メンバの出力を行うサンプルプログラムを示す。3種類の方法での出力は同じ結果となる

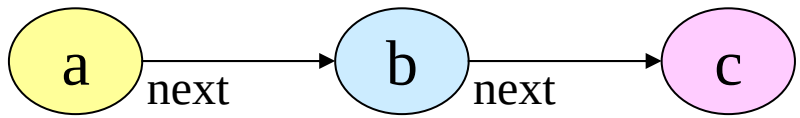
```
#include <stdio.h>
struct roll {
  char name[30];
  int  birth;
  char address[80];
  int  gender;
};

main()
{
  struct roll *p,my_data={初期化データ};
  p = &my_data;

  /* 構造体変数そのまま */
  printf("%s¥n", my_data.name);
  printf("%d¥n", my_data.birth);
  printf("%s¥n", my_data.address);
  printf("%d¥n", my_data.gender);
```

```
  /* ポインタ＋間接演算子＋ピリオド */
  printf("%s¥n", (*p).name);
  printf("%d¥n", (*p).birth);
  printf("%s¥n", (*p).address);
  printf("%d¥n", (*p).gender);

  /* ポインタ＋アロー演算子 */
  printf("%s¥n", p->name);
  printf("%d¥n", p->birth);
  printf("%s¥n", p->address);
  printf("%d¥n", p->gender);
}
```

自己参照的構造体

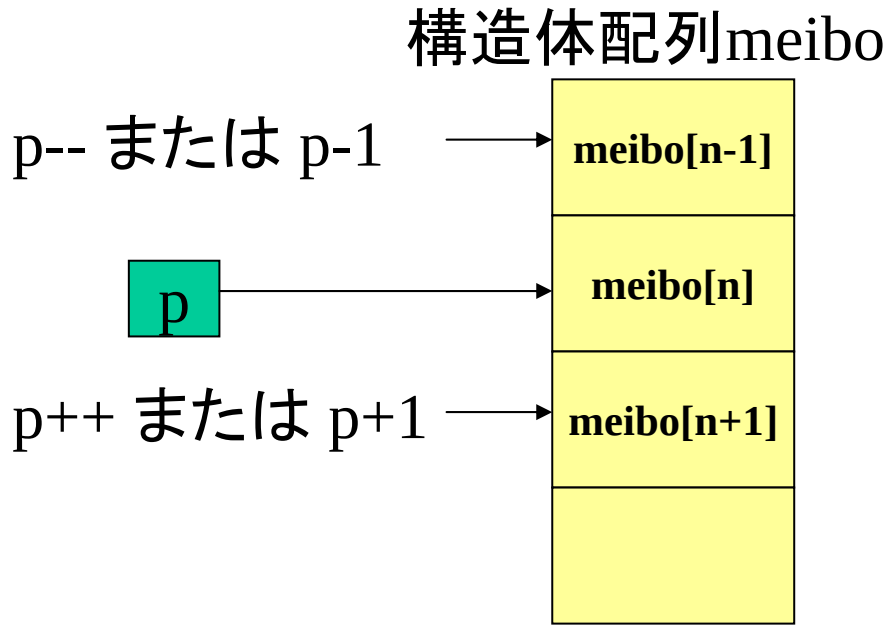
- 構造体内に自分の型のポインタを置く場合がある
- これを「自己参照的構造体」と呼ぶ
- 以下のような場合、
  - `a.next` は `b` を指す
  - `b.next` は `c` を指す
  - `a.next->next` は `c` を指す
- このようにデータが順に繋がっているデータ構造を連結リストと呼び、来週の授業で更に詳しく説明する

```
struct roll {
  char name[30];
  int  birth;
  char address[80];
  int  gender;
  struct roll *next;
};

struct roll a,b,c;
a.next = &b;
b.next = &c;
```



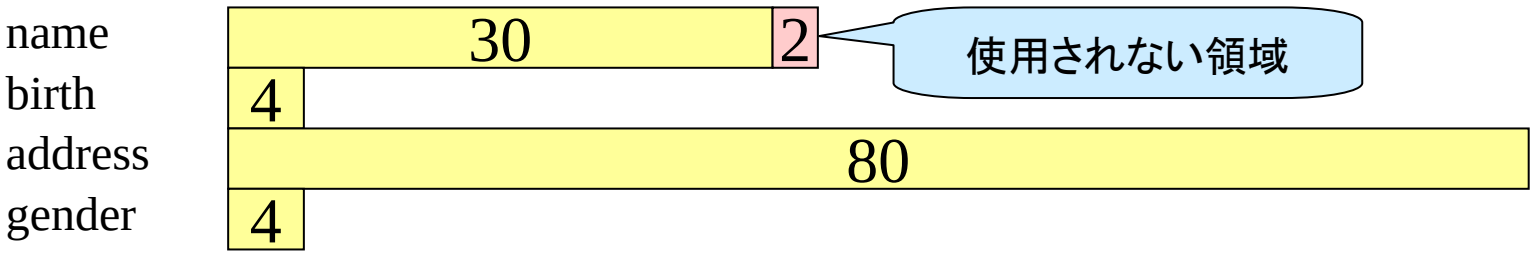
構造体のポインタ演算

- 構造体の場合も通常の配列同様に、ポインタに対して加算・減算・インクリメント・デクリメント演算を行なうことが出来る。
- インクリメント処理によって増えるアドレスの量は構造体配列の要素1個分の大きさである。
（つまり**`sizeof(meibo[0])`**、 **`roll`**型の場合**`120`**バイト）



構造体の大きさ

- なお、構造体の大きさは必ずしもメンバーの大きさの合計にはならない。
- 例えば、**roll**型の場合、
  - 単純な合計は **(30+80)*sizeof(char) + 2*sizeof(int) = (30 + 80) + 2*4 = 118**
  - コンピュータ**(std1ss1)**上で**sizeof(meibo[0])**によって表示させると120と表示された。
  - 以下のプログラムで各メンバの先頭アドレスの差を表示させたところ、文字配列**name**の後に2バイトの穴があることが分かる。（このプログラムは参考に載せたものなので、意味が理解できなくても差し支えない）



構造体のポインタ演算

- ポインタ演算を行うサンプルプログラムを示す。

```
#include <stdio.h>
struct roll {
  char name[30];
  int  birth;
  char address[80];
  int  gender;
};
main()
{
  struct roll *p, meibo[2] = {
    {"要素0初期化データ"},
    {"要素1初期化データ"},
  };

  p = &meibo[0]; /* p = meibo; でも良い */
  printf("meibo[0]: %p %p¥n",p, &meibo[0]);
  p++;
  printf("meibo[1]: %p %p¥n",p, &meibo[1]);
  printf("sizeof(meibo) = %d , sizeof(meibo[1]) = %d¥n",
         sizeof(meibo), sizeof(meibo[1]));
}
```

```
s1000001{std0ss0}1: ./.a.out
meibo[0]: effff9f8 effff9f8
meibo[1]: effffa70 effffa70
sizeof(meibo) = 240 , sizeof(meibo[1]) = 120
s1000001{std0ss0}2:
```

構造体のポインタ演算

- 構造体配列メンバの出力を行うサンプルプログラムを示す。4種類の方法での出力は同じ結果となる

```
#include <stdio.h>
struct roll {
  char name[30];
  int  birth;
  char address[80];
  int  gender;
};
main()
{
  int i;
  struct roll *p, *q, meibo[2] = {
    {"要素0初期化データ"},
    {"要素1初期化データ"},
  };
  p = meibo;
  for(i = 0; i < 2; i++){
    printf("%s¥n", (*(p + i)).name);
    printf("%d¥n", (*(p + i)).birth);
    printf("%s¥n", (*(p + i)).address);
    printf("%d¥n", (*(p + i)).gender);
  }
  for(i = 0; i < 2; i++){
    printf("%s¥n", (p + i)->name);
    printf("%d¥n", (p + i)->birth);
    printf("%s¥n", (p + i)->address);
    printf("%d¥n", (p + i)->gender);
  }
  for(q = p; q < p + 2; q++){
    printf("%s¥n", (*q).name);
    printf("%d¥n", (*q).birth);
    printf("%s¥n", (*q).address);
    printf("%d¥n", (*q).gender);
  }
  for(q = p; q < p + 2; q++){
    printf("%s¥n", q->name);
    printf("%d¥n", q->birth);
    printf("%s¥n", q->address);
    printf("%d¥n", q->gender);
  }
}
```

関数への構造体のアドレス渡し

```
#include <stdio.h>
struct xy {
  float  x; /* x座標 */
  float  y; /* y座標 */
};
void swap(struct xy *, struct xy *);
main()
{
  struct xy data1 = {1.0,2.0}, data2 = {3.0,4.5};
  swap(&data1,&data2);
  printf("data1:(%3.1f,%3.1f) data2(%3.1f,%3.1f)\n",
         data1.x,data1.y,data2.x,data2.y);
}
void swap(struct xy *a, struct xy *b)
{
  struct xy tmp;
  tmp = *a;
  *a = *b;
  *b = tmp;
  printf("a:(%3.1f,%3.1f) b(%3.1f,%3.1f)\n",a->x,a->y,b->x,b->y);
}
```

構造体の入れ子構造

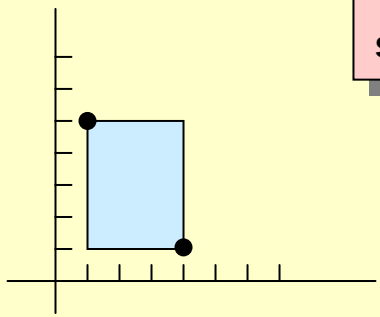
- 構造体宣言の中に構造体の定義があるような構造体の構造を「入れ子」と呼んでいる
- これは既にある構造体を含んで更に別のデータのまとまりを作り上げるときに有効である。
  - 例えば次頁の例は「平面の点」構造体を2つ使用してx軸、y軸に平行な辺を持つ長方形の構造体を宣言している
  - この例の場合はただの「構造体配列」でも実現可能だが、構造体の入れ子の方が応用範囲が広い。
- 入れ子の構造体のメンバーへのアクセスは以下のように書く

外側の構造体名.内側の構造体.メンバ名

構造体の入れ子構造

- 先週(lec8-19)平面上の２点を対角点とする長方形を構造体配列を使って考えたが、ここでは「平面上の点」構造体二点をメンバーとして持つ構造体」として考える。

```
#include <stdio.h>
#include <math.h>
struct xy {
  float  x; /* x座標 */
  float  y; /* y座標 */
};
struct rect {
  struct xy p1;
  struct xy p2;
};

main()
{
  struct rect  rect1 = {{1.0,5.0},{4.0,1.0}};
  float area;

  area = (float)fabs((rect1.p1.x - rect1.p2.x) *
                     (rect1.p1.y - rect1.p2.y));
  printf("The area of the rectangle is %f\n",area);
}
```



```
s1000001{std0ss0}1: ./a.out
The area of the rectangle is 12.000000
s1000001{std0ss0}1:
```

構造体メンバーは
「構造体名１.構造体名２.メンバー」
のようにアクセスする

構造体の特徴

- 構造体を使う利点は、
  - データの取り扱いが明確になり可読性が向上する。
  - 扱う変数の個数が少なくなり、プログラムの簡略化を図ることが出来る。
  - データをまとめて扱うことができる。これをデータのカプセル化と呼ぶ。
    - x座標、y座標を持つ平面上の点を「点」として一括して取り扱える
  - 継承するデータは、階層的に構造体を定義すると非常に有効的となる。
    - 「点」の集まりとして四角形などの図形を考えることが出来る

- 構造体を使う注意点
  - メンバーをアクセスするときに、名前はながくなりやすい
  - 代入などの操作を平気に使ってしまい、余計に計算時間がかかる場合もある
  - ドット、アロー、などを使うときに、間違いやすい

良くあるプログラミングミス(1)

- 構造体を使用した場合の良くあるプログラムの間違いを挙げてみた。
- なお、この節の例は全て構造体タグxyを使用するので、構造体タグの定義は省略した。また#includeも省略してある

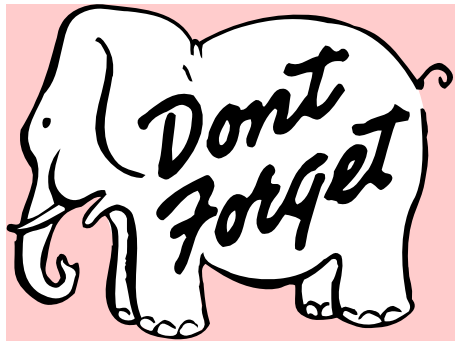
1. ポインタと「*」を使って構造体のメンバーをアクセスする時には、必ずカッコが必要。構造体ポインタpがある時、

○	×
`(*p).x`	`*p.x`

2. 構造体と構造体ポインタをはっきり区別する。構造体変数pointと構造体ポインタpがある時、

○	×

`(*p).x`
`point.x`
`p->x`
	`*p.x`
	`(*point).x`
	`point->x`



良くあるプログラミングミス(2)

3. アロー演算子(->)の間に空白を入れない(- >のように)

```
main()
{
  struct xy *p, data1 = {1.0,2.0};
  p = &data1;

  printf("x : %f, y : %f¥n", p- >x, p->y);
}
```

コンパイラのエラーメッセージ:

parse error before `>'

4. 大きな構造体を引数にする場合は、アドレス渡しの方が速い場合がある
   - 値渡しの場合は、大きな構造体を関数間でコピーするのに時間がかかるため
   - ただし「速い」からといって、どんな時でもアドレス渡しをするのはいけない。見易さ、理解し易さも考えて、どちらを使用すべきかを考える必要がある。
   - 次頁に挙げたプログラムは値渡しとアドレス渡しの速さを比較するために掲載した極端な例である(Cの実験室上級ラボ編、林著より)。自分でも試して体感してみると良いだろう。

良くあるプログラミングミス(3)

時間に関する関数群の定義
詳しくはman clock などを参照のこと

```
#include <stdio.h>
#include <time.h>
#define LOOP 200000

struct test { /* 5000文字の文字配列がメンバーの構造体 */
  char  a[5000];
};
void fv(struct test);
void fr(struct test *);
main()
{
  int i;
  struct test testdata;
  time_t start,end; /* 時間計測用の変数 */
  double keika;

  start = clock();
  for(i = 0; i < LOOP; i++) fv(testdata);     /* 関数呼び出し */
  end = clock();
  keika = (end-start)/(double)CLOCKS_PER_SEC;
  printf("Call by value   : %f sec\n",keika); /* 経過時間の表示 */

  start = clock();
  for(i = 0; i < LOOP; i++) fr(&testdata);    /* 関数呼び出し */
  end = clock();
  keika = (end-start)/(double)CLOCKS_PER_SEC;
  printf("Call by address : %f sec\n",keika); /* 経過時間の表示 */
}
```

```
/* 値渡しの関数 */
void fv(struct test t1) {
  t1.a[0] = 'A';
}

/* アドレス渡しの関数 */
void fr(struct test *t2)
{
  t2->a[0] = 'A';
}
```

実行結果

```
std1ss40(Blade100 model500 500MHz Solaris 8)
Call by value   : 5.820000 sec
Call by address : 0.010000 sec

std3ss20(Blade 150 model550 550MHz Solaris 8)
Call by value   : 4.720000 sec
Call by address : 0.010000 sec

std5ss1(Blade 150 model650 650MHz Solaris 8)
Call by value   : 4.400000 sec
Call by address : 0.010000 sec
```