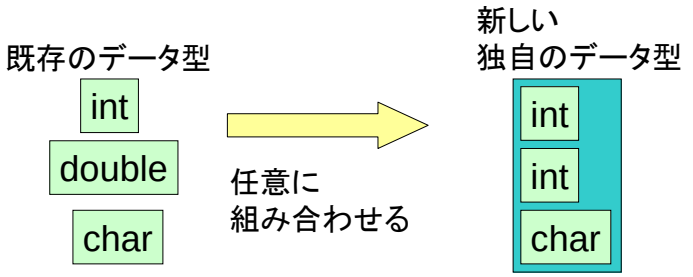
プログラミング演習
構造体（2）

阿萬裕久
aman@cs.ehime-u.ac.jp

構造体

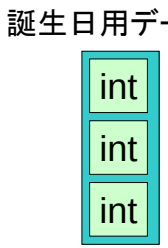
- ソフトウェア設計者が**独自に作る**ことのできる **新しい型**である



（例）誕生日

- 誕生日を表現するには
  - 年，月，日

  という3つの整数値が必要



（例）誕生日

- C 言語で書くと

```
struct birthday {
   int  year;
   int  month;
   int  day;
};
```

命名は自由

struct birthday
という名前の
新しい型として使える

```
struct birthday  taro;
struct birthday  jiro;
taro.year = 1986;
taro.month = 6;
taro.day = 7;
jiro = taro;
```

例題1

- 2 次元のベクトルを表現するための構造体 vector を定義しなさい.
  そして, その内容を

    (x, y)

  のかたちで表示するための関数 printVector を作りなさい.
  ただし, ベクトルの各成分は int 型とする.

例題1 : 解説

Web ページのコーディング例を参照

- 構造体(新たに struct vector という型)を宣言します:

9~12行目

```
struct vector {
    int x;
    int y;
};
```


例題1 : 解説

- main 関数の中で変数を 1 個用意し,

22行目

```
struct vector data;
```

- scanf 関数を使ってベクトルの内容を読み込みます

22行目

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


例題1 : 解説

- 単純に表示するだけならそのまま printf 関数を使えば終わりです

```
printf("(%d, %d)¥n", data.x, data.y);
```

- しかし, この問題では, 表示するための関数 printVector を作りなさい となっています

例題1：解説

- 関数といっても，小さなプログラムを代行するだけ

15～19行目

```
void printVector( struct vector v ){
    printf("( %d, %d )¥n", v.x, v.y );
}
```

26行目

```
printVector( data );
```

コピー & 実行

v を仮引数（かりひきすう）
data を実引数（じつひきすう）という

例題2

- 例題 1 で説明した関数 printVector を次の条件に従って書き換えなさい：
  - 仮引数をポインタ変数に変更
  - ベクトルの表示機能はそのまま

C 言語ワンポイントレッスン

- ポインタの基礎

ポインタ（pointer）

- メモリ上の番地を使って変数にアクセスするもののこと
  - 変数の読み書きは，その変数の名前を使うのが最も簡単な方法
  - ポインタでは，名前の代わりに番地（格納場所）を使う

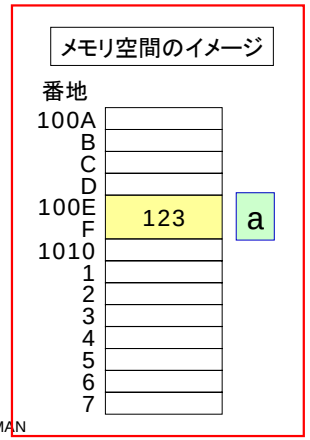
簡単な例

- 次のプログラムでは，変数 a に 123 が代入される

① `int a;`
② `a = 123;`

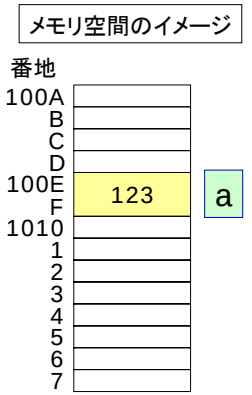
① int 型の変数をメモリ上に確保する．
右の例では 100E 番地から2バイト分の領域で，これを「a」と名付ける．

② 「a」に対して 123 を代入する．



名前の代わりに番地を

- この例の場合，変数 a は 100E 番地の変数という見方もできる
- 人間に例えると
  - 「愛大 太郎」
  - 「松山市文京町3番に住んでいる人」



これがポインタの考え方

番地の調べ方

- 変数が格納される領域の番地は自動的に割り当てられる
  - 常に固定されているわけではない
- 変数名の前に＆（アンパサンド）を付けると，その番地を調べることができる

```
&a
```

ただ，これだけだと使いようがない．．．
番地を記録するための変数が必要

ポインタ変数

- 番地を格納するための変数
- 名前の前に＊（アスタリスク）を付けて宣言

```
int a;
int* p;

a = 123;
p = &a;
```

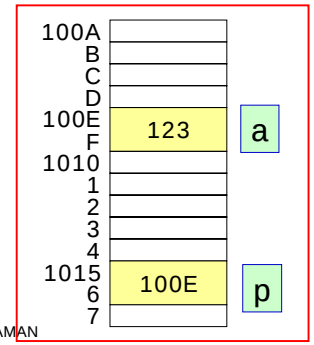
```
int*  p;
int   *p;
```
の2種類の書き方があるがどちらも可

変数 p には変数 a の番地が格納される

改めてメモリ空間のイメージ図

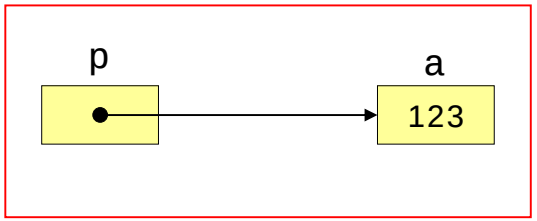
- int 変数 a とポインタ変数 p のイメージ
  - p にも自動的にその領域が割り当てられる
  - p の内容は a が置かれている番地となっている

```
int  a;
int* p;
a = 123;
p = &a;
```



よく使われるイメージ図

- 変数 a の格納番地がポインタ変数 p に代入されている場合



p の内容は a の番地（場所）を指しているという意味

※もともと「ポインタ」という言葉は「指し示すもの」という意味

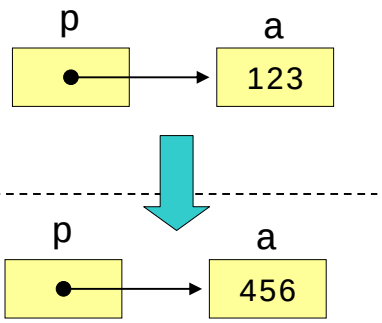
ポインタを使ったアクセス

- ポインタ変数の前に＊を付けると，参照先の内容になる
- つまり

ポインタを使ったアクセス例

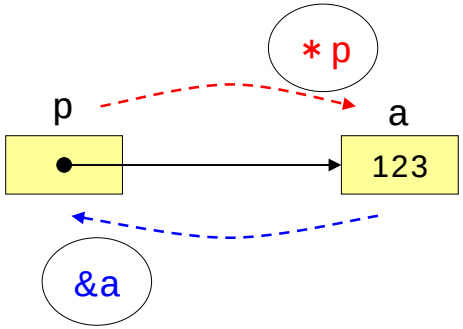
- 簡単な例

```
int  a;
int* p;

a = 123;
p = &a;

*p = 456;
```



＊ と ＆ が持つ働き

- ＊は矢印の向きに従ってたどっていく
- ＆は矢印の向きに逆らってたどっていく



ポインタ変数の型

- 参照先の変数の型（int, double 等）を明確にしておく必要がある

```
int＊  p;
```

ポインタ変数 p は，
int 型変数へのアクセス専用となる

見方を変えると，＊p が int 型変数を参照することになる

```
int  ＊p;
```


メモリ管理の注意点

- ポインタ変数を宣言しただけでは，肝心のデータ領域が存在しない！

```
int＊ p;
＊p =  3;
```

← 間違い！

この時点では，変数 p には何が入っているか分からない

つまり，メモリ上のどこを指しているかは不明なのに，
その指定先に「3」を代入しようとしている！

メモリエラーを引き起こす
Segmentation fault（セグメントエラー）

正しいメモリ管理

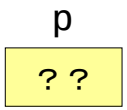
- 自動変数（変数宣言されたもの）を使う

```
int＊ p;
int  a;
p = &a;
＊p =  3;
```

正しいメモリ管理

- 自動変数(変数宣言されたもの)を使う

```
int* p;
int  a;
p = &a;
*p =  3;
```



正しいメモリ管理

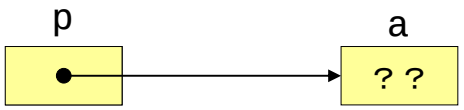
- 自動変数(変数宣言されたもの)を使う

```
int* p;
int  a;
p = &a;
*p =  3;
```

正しいメモリ管理

- 自動変数(変数宣言されたもの)を使う

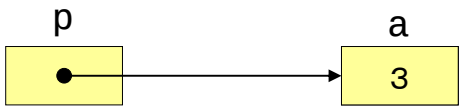
```
int* p;
int  a;
p = &a;
*p =  3;
```



正しいメモリ管理

- 自動変数(変数宣言されたもの)を使う

```
int* p;
int  a;
p = &a;
*p =  3;
```

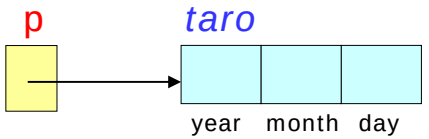


まとめると

- ポインタは番地を使って変数にアクセスする
- 番地を調べる: ＆変数名　（例） &a
- アクセスする ： ＊ポインタ　（例）＊p

ポインタを使った構造体へのアクセス

```
struct birthday  taro;
struct birthday＊  p;
p = &taro;
```

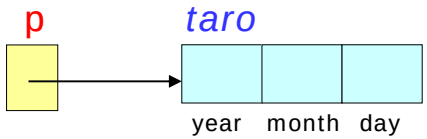


ポインタを使った構造体へのアクセス

```
taro.year
taro.month
taro.day
```

⇔

```
(＊p).year
(＊p).month
(＊p).day
```



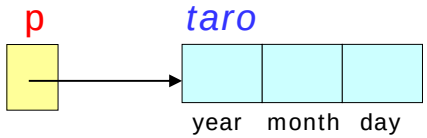
ポインタを使った構造体へのアクセス

- 構造体へのポインタでは特別な表記が許されている

```
(＊p).year
(＊p).month
(＊p).day
```

⇔

```
p->year
p->month
p->day
```



マイナス(ー)と大なり記号(＞)を組み合わせて**矢印**を表現

話題を例題2に戻します

例題2

Web ページのコーディング例を参照

- 例題 1 で説明した関数 printVector を次の条件に従って書き換えなさい:
  - **仮引数をポインタ変数に変更**
  - ベクトルの表示機能はそのまま

例題2:解説

Web ページのコーディング例を参照

- 仮引数をポインタ変数に変更

```
void printVector( struct vector v ){
   printf("( %d, %d )¥n", v.x, v.y );
}
```

15～19行目

```
void printVector( struct vector* v ){
   printf("( %d, %d )¥n", v->x, v->y );
}
```


例題2:解説

- 関数を呼び出す側もこれに合わせて変更

例題1

```
void printVector( struct vector v ){
```

```
printVector( data );
```

26行目

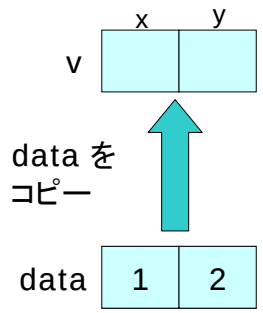
例題2

```
void printVector( struct vector* v ){
```

```
printVector( &data );
```

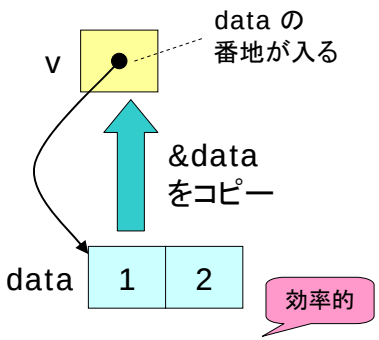
例題2：解説（仮引数と実引数の関係）

- 例題1の場合



データを全部コピーする

- 例題2の場合



データの番地をコピーする

例題3

- ベクトルの長さを調べる関数 length を作りなさい．ただし，ベクトル（構造体 vector）の各成分に対するアクセスにはポインタ変数を用いることとする．

  なお，ベクトルの長さは double 型で算出せよ．

例題3：解説

- ベクトルの長さ $|(x,y)| = \sqrt{x^2+y^2}$
  - 平方根は sqrt で求まる

構造体のポインタが v ならば

```
sqrt( (v->x)*(v->x) + (v->y)*(v->y) );
```

これを関数 length の**戻り値**に指定すればよい

C 言語ワンポイントレッスン

- 関数の戻り値

関数の戻り値

- 関数は何らかの答え（数値や文字）を求めるのが本来の仕事
- 数学でいう「関数」ならば，関数は必ず値を持つはず …
  - 例えば，sin(x) は関数の一種（三角関数）
    x に適当な値を代入すれば，この関数 sin(x) は -1 以上 1 以下の実数値をとる

実は，C 言語でいうところの関数も同じ

関数の戻り値

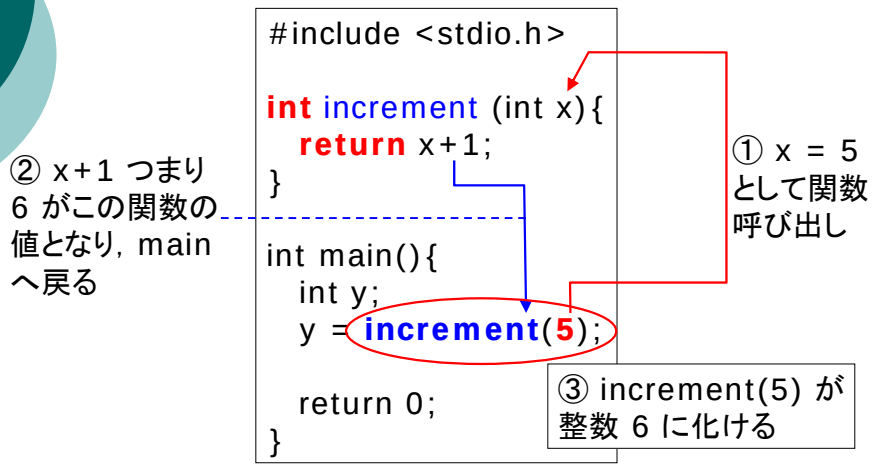
- 「この関数がどういう値を持つのか」についてその型を宣言
  ※ void は特殊な場合で「**値を持たない**」の意味

```
void printVector( struct vector v ){
   printf("( %d, %d )¥n", v.x, v.y );
}
```


関数の戻り値

- 具体的にどういう値を関数に持たせるのか？
  - **return** という命令文で定義する
  - その値のことを関数の戻り値（もどりち）という

例： 値を＋1する関数



関数の定義方法

```
戻り値の型  関数名 ( 仮引数の宣言 ){
   実行したいプログラム
   return  戻り値;
}
```

- return は場合分け(if 文等)により複数書いてもよいが, return の実行はその関数の実行終了を意味することに注意
  - 途中でも return がきたらそこで終わり

例題3:解説

Web ページのコーディング例を参照

- したがって, 関数 length とその呼び出しは

```
double  length( struct vector * v){

  return sqrt( (v->x)*(v->x) + (v->y)*(v->y) );

}
```

(main の中)

```
len = length(&data);
```

len にベクトル data の長さが代入される