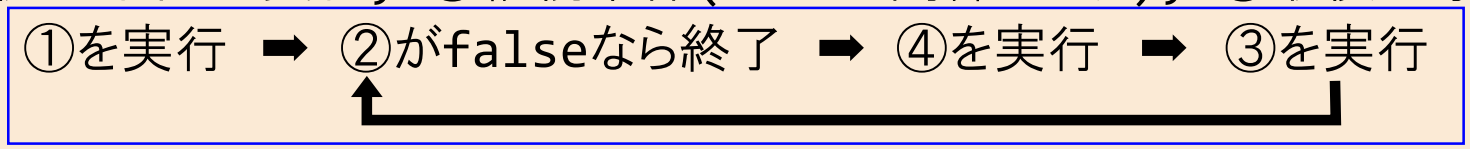
画像処理
(4J)

第04回
～C言語の復習（4）～
『ポインタ』

第1回のまとめ

- 標準ライブラリヘッダの取り込みと、main()関数の書き方
  - #include <stdio.h>
  - int main(void){ }
- 変数宣言など、基本的なCプログラムの書き方
  - int a=10;　　"文字列リテラル"　　基本的な演算子　　大文字と小文字の区別等
- printf()関数の基本的な使い方
  - 書式指定文字 %d, %s
- for文の使い方
  - for (①最初に1回だけ実行; ②継続条件(trueの間繰り返す); ③最後に毎回実行) { ④ }



- 変数のスコープ
  - ローカル変数(局所変数) ⇔ グローバル変数(大域変数)

第2回のまとめ

●変数のスコープの話(再)

●変数の宣言

- 型、定数　`static / const / unsigned / {short, long} / {char, int, float, double}`
  `65 == 0101 == 0x41 == 'A'` (※10進数, 8進数, 16進数, 文字定数)
- 初期化　`int a[5] = {1, 2, 3, 4, 5};` ･･･配列宣言時の初期化
- 暗黙的な型変換、明示的な型変換(キャスト)
  ･･･同じ型同士なら結果も同じ型に、異なる型なら表現力の高い型になる。
  (型名) を頭に付けると、明示的に型変換を指定できる。(=キャスト)

●条件式

- 真(true)と偽(false)　･･･0==偽(False)、1==真(True)　(※判断の際は、0以外は真 と扱われる)
- if文　･･･ if (条件) {真の場合の処理} else {偽の場合の処理}
- switch文　･･･ break; が無いとそのまま下の行に処理が継続することに注意
- 三項演算子　･･･「(条件式)? trueの場合 : falseの場合;」 式全体を値として使える

~~●sizeof演算子~~　~~･･･配列の要素数は sizeof(a)/sizeof(a[0]) で得られる~~

第3回のまとめ

- sizeof演算子　　・・・配列の要素数は sizeof(a)/sizeof(a[0]) で得られる
- その他これまで説明せずに使っていたもの
  - コメント　　・・・「/*」から、最初に現れた「*/」まで。または「//」から行末まで。
  - 各種演算子等　　・・・ a++; ++a; a=++b; a=b++; a+=b;
- 関数
  - 返り値の型、引数リスト　　・・・戻り値の型　関数名(　引数の型と名前　)　、プロトタイプ宣言、値渡し
- 標準入力／標準出力
  - printf(), scanf(), (scanf_s())
- 標準ライブラリ関数
  - 算術処理　　・・・ sin(), cos(), tan(), atan(), atan2(), pow(), sqrt()　　math.h
  - 疑似乱数を使う　　・・・ rand(), srand(), time()　　・・・実行の度に異なる乱数列の発生
    stdlib.h　　time.h　　任意の範囲の乱数への変換方法

「¥」記号について

●日本語の環境では、多くの場合　¥　(円記号) として表示されますが、
同じ文字を多言語で表示すると　＼　(backslash) で表示されます。

➡ この2つは、全く同じもの(文字コード)ですが、
それぞれの環境の違いによって、表示(見え方)が異なっています。

これまでに、 ¥n とか ¥0 として説明してきましたが、
環境によっては ＼n とか ＼0 と表示されます。

● 英語配列

● 日本語配列

●第一部●
ポインタとは

ごめんなさい、
関連情報の一覧性を
優先したため、
今回文字が小さめです
見にくい場合はMoodleに置いた
pdf資料も参照して
下さい。

C言語でわからん！ってなりがちなポインタ

●メモリ と アドレス と ポインタ

- プログラム(命令)も、データ(変数)も、メモリに格納されています。(➡ノイマン型コンピュータ)
- メモリには アドレス(番地) が付いていて、それによってデータを特定します。

```
char data[10]={0,2,4,6,8,10,12,14,16,18};
```

と宣言した場合の例。

※ sizeof(char) = 1 byte

char 1つが 1byte なので、"変数1つ"が「アドレス1つ分のメモリ領域」を使っている

+1

アドレス(番地):	中身
0x7ffc2b03bc70:	0x00
0x7ffc2b03bc71:	0x02
0x7ffc2b03bc72:	0x04
0x7ffc2b03bc73:	0x06
0x7ffc2b03bc74:	0x08
0x7ffc2b03bc75:	0x0A
0x7ffc2b03bc76:	0x0C
0x7ffc2b03bc77:	0x0E
0x7ffc2b03bc78:	0x10
0x7ffc2b03bc79:	0x12

1つの番地には、
1 byte = 8 bit
のデータが入る

[8bit]
00000000～11111111
00～FF (16進数表記)

C言語でわからん！ってなりがちなポインタ

●メモリ と アドレス と ポインタ

- プログラム(命令)も、データ(変数)も、メモリに格納されています。(➡ノイマン型コンピュータ)
- メモリには アドレス(番地) が付いていて、それによってデータを特定します。

```
int data2[10]={0,2,4,6,8,10,12,14,16,18};
```

と宣言した場合の例。

※ sizeof(int) = 4 byte

アドレス(番地): 中身

0x7ffcb8788f10:	0x00 0x00 0x00 0x00
0x7ffcb8788f14:	0x00 0x00 0x00 0x02
0x7ffcb8788f18:	0x00 0x00 0x00 0x04
0x7ffcb8788f1c:	0x00 0x00 0x00 0x06
0x7ffcb8788f20:	0x00 0x00 0x00 0x08
0x7ffcb8788f24:	0x00 0x00 0x00 0x0A
0x7ffcb8788f28:	0x00 0x00 0x00 0x0C
0x7ffcb8788f2c:	0x00 0x00 0x00 0x0E
0x7ffcb8788f30:	0x00 0x00 0x00 0x10
0x7ffcb8788f34:	0x00 0x00 0x00 0x12

+4

int 1つが4byteなので、"変数1つ"が「アドレス4つ分のメモリ領域」を使っている

【ちゃんと書くとこんな感じ・・・】

アドレス(番地): 中身

0x7ffcb8788f10:	0x00
0x7ffcb8788f11:	0x00
0x7ffcb8788f12:	0x00
0x7ffcb8788f13:	0x00
0x7ffcb8788f14:	0x00
0x7ffcb8788f15:	0x00
0x7ffcb8788f16:	0x00
0x7ffcb8788f17:	0x02
0x7ffcb8788f18:	0x00
0x7ffcb8788f19:	0x00
0x7ffcb8788f1a:	0x00
0x7ffcb8788f1b:	0x04 …

※ビッグエンディアンの場合

参考

●前ページ、前々ページの
char / int 型変数の
アドレスと内容を確認するための
サンプルプログラム。

※printf()で %p を指定すると、
アドレス値が16進数で出力される。
(変数名の前の"&"の意味は後述。)

※配列は、必ずメモリ内の
連続した領域に、順番に配置される。
(例えば data[0] と data[1] の
番地は &data[0] < &data[1]
となり、 &data[0], …, &data[9]
の間隔は一定(=1)。
(&data2[0], …, &data2[9]
の場合も同様で、間隔は4)

https://paiza.io/projects/HSsWwsNrwjAzajbR-lEHrA

```
// Example: 04-01

#include <stdio.h>
int main(void){
    char data[10]={0,2,4,6,8,10,12,14,16,18};
    int data2[10]={0,2,4,6,8,10,12,14,16,18};

    printf("(char = %d byte)¥n", sizeof(char));
    for (int i=0; i<10; i++) {
        printf("%p: %X¥n", &data[i], data[i]);
    }

    printf("¥n(int = %d byte)¥n", sizeof(int));
    for (int i=0; i<10; i++) {
        printf("%p: %X¥n", &data2[i], data2[i]);
    }

    return 0;
}
```

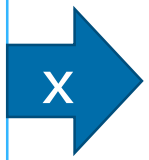
アドレス？

※ sizeof(char) = 1 byte


```
char x = 3;
```
と宣言

メモリ上の“どこか”の番地に
変数 x の領域が
割り当てられる。
↓
例えば、 0x7ffc2b03bc70 番地
に割り当てられたとする。

「変数 x」とは、
0x7ffc2b03bc70 番地
に入っている char型の値



アドレス(番地):	中身
0x7ffc2b03bc70:	0x03
0x7ffc2b03bc71:	…
0x7ffc2b03bc72:	…

このとき、
「x」は、“0x7ffc2b03bc70 番地 の
中身の 0x03 (8bitのbit列*) を
「char型」として解釈した値”を意味する。

また、「&x」は、xの値が入っている番地である
0x7ffc2b03bc70 という値の「char型のポインタ」
を意味する。 ➡ 「char*型」とも言う

＊「変数x は char型」で、char型の大きさは8bitと分かっているので、
「8bitのbit列」をひとかたまりとして解釈する。

アドレス？

※ sizeof(char) = 1 byte

`char x = 3;`
と宣言

「変数 x」とは、
0x7ffc2b03bc70 番地
に入っている char型の値

x

アドレス(番地):	中身
0x7ffc2b03bc70:	0x03
0x7ffc2b03bc71:	…
0x7ffc2b03bc72:	…

https://paiza.io/projects/poqaNZF0UJqhZ46_1cUBIg

```
// Example: 04-02
#include <stdio.h>
int main(void){
    char x = 3; // char型の変数
    char *p;    // char型のポインタ

    p = &x;     // 変数x のアドレスをpに代入

    printf("x = %d, *p = %d¥n", x, (*p)); // "中身"を表示
    printf("&x = %p¥np  = %p¥n", &x, p);  // "アドレス"を表示

    return 0;
}
```

※割り当てられるメモリ領域は、その都度違うので、%pで表示したアドレスは毎回異なるかも知れない。しかし、&x と p は常に同じアドレスになっているはず。

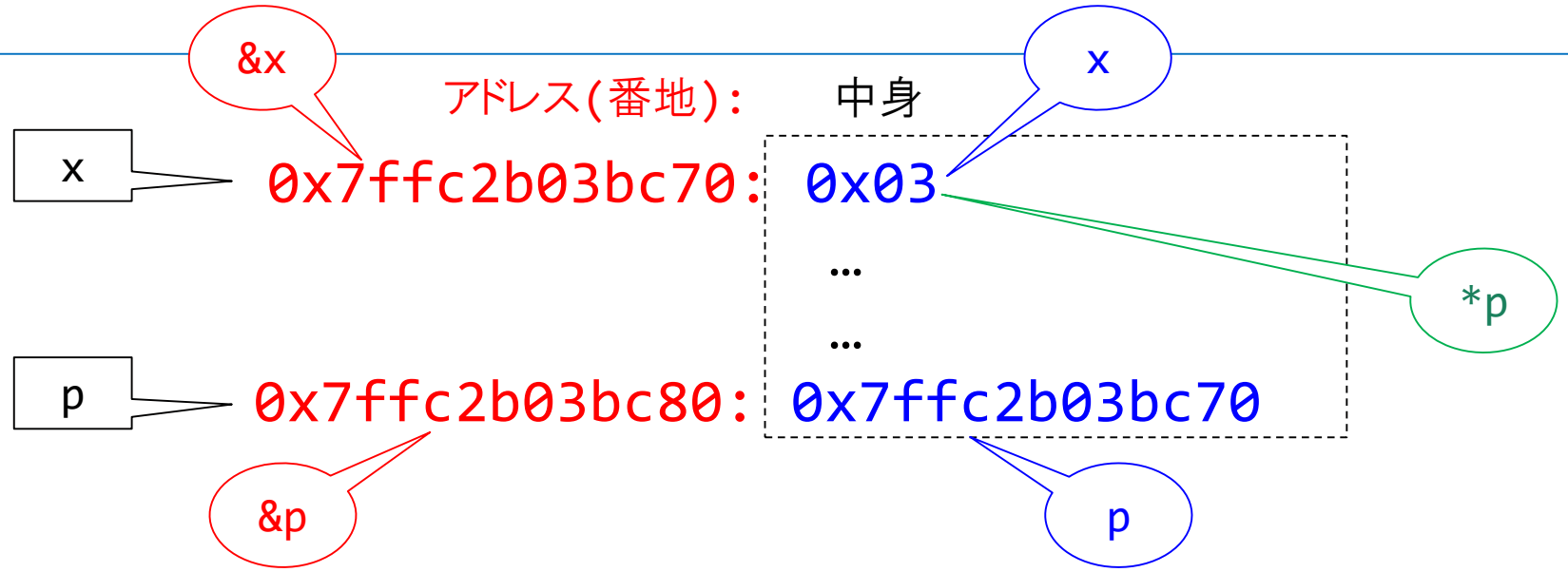
通常の型に「*」を
付けて宣言すると、
その型の
ポインタ変数となる。

ポインタ変数に
「*」を付けると、
そのアドレスの
"中身"の意味になる。

まとめ


```
char x = 3;
char *p = &x;
```

・・・とした場合。



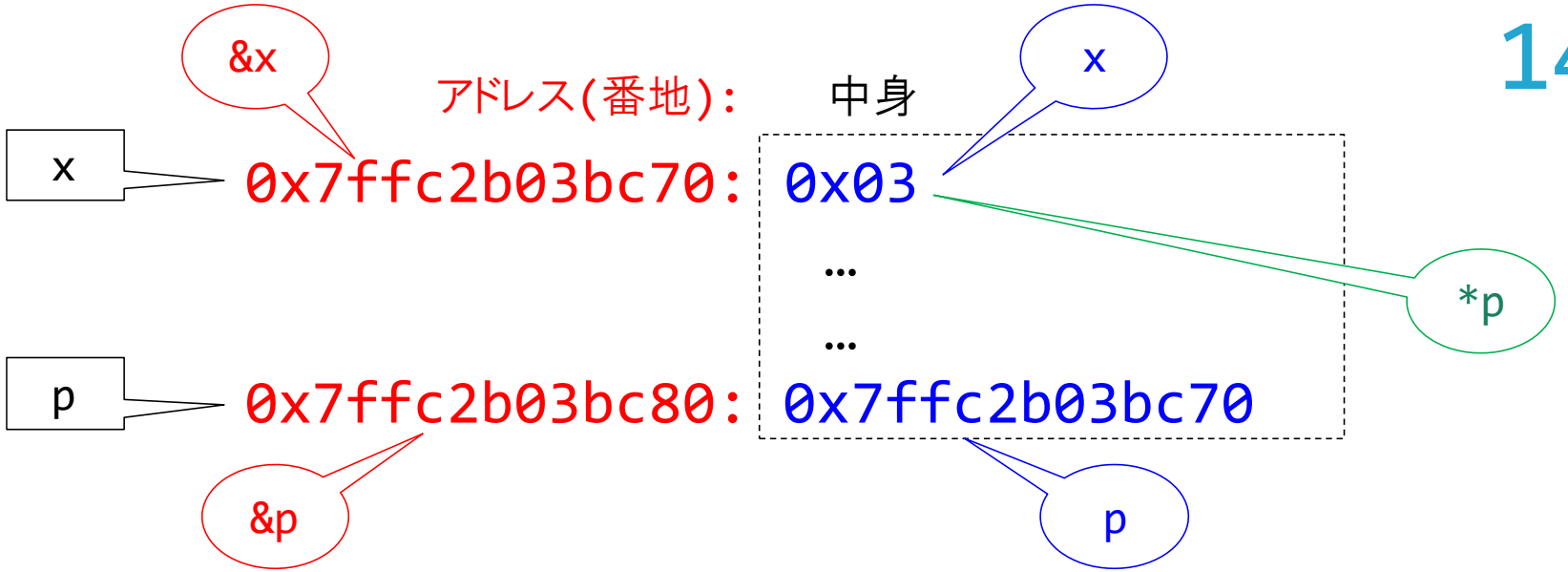
x ・・・ 3 「値(char値)」(中身)

&x ・・・ 0x7ffc2b03bc70 xが入っている「アドレス」

p ・・・ 0x7ffc2b03bc70 「値(アドレス値)」(中身)

&p ・・・ 0x7ffc2b03bc80 pが入っている「アドレス」

*p ・・・ 3 「アドレス値が指す番地の"中身"」



https://paiza.io/projects/KlCFiGFiiPLZo64_6ifBlw

```
// Example 04-03
#include <stdio.h>
int main(void){
    char x = 3;
    char *p = &x;

    printf("x = %d, *p = %d¥n", x, (*p));
    printf("&x = %p, p = %p¥n", &x, p);
    printf("&p = %p¥n", &p);

    return 0;
}
```

※割り当てられるメモリ領域は、
その都度違うので、
&x と &p の値は図とは異なります。

演習(1)

- Example: 04-02, 04-03 を実行(+改造等)して、
  ポインタの動作について実際に確認してみましょう。

アドレス？

※ sizeof(char) = 1 byte

12

`char x = 3;`
と宣言

「変数 x」とは、
0x7fffc2b03bc70 番地
に入っている char型の値

x

アドレス(番地): 中身

0x7fffc2b03bc70: 0x03
0x7fffc2b03bc71: …
0x7fffc2b03bc72: …

https://paiza.io/projects/poqaNZF0UJqhZ46_1cUBIg

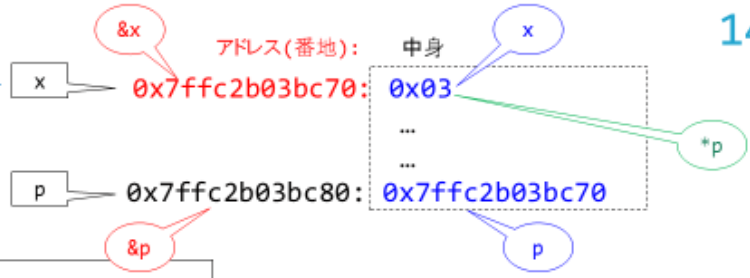
```
// Example: 04-02
#include <stdio.h>
int main(void){
    char x = 3; // char型の
    char *p;    // char型のポインタ

    p = &x;     // 変数x のアドレスをpに代入

    printf("x = %d, *p = %d¥n", x, (*p)); // "中身"を表示
    printf("&x = %p¥np  = %p¥n", &x, p);  // "アドレス"を表示

    return 0;
}
```

※割り当てられるメモリ領域は、その都度違うので、%pで表示したアドレスは毎回異なるかも知れない。しかし、&x と p は常に同じアドレスになっているはず。

通常の型に「*」を付けて宣言すると、その型のポインタ変数となる。

ポインタ変数に「*」を付けると、そのアドレスの"中身"の意味になる。

14



https://paiza.io/projects/KlCFiGFiiPLZo64_6ifBlw

```
// Example 04-03
#include <stdio.h>
int main(void){
    char x = 3;
    char *p = &x;

    printf("x = %d, *p = %d¥n", x, (*p));
    printf("&x = %p, p = %p¥n", &x, p);
    printf("&p = %p¥n", &p);

    return 0;
}
```

※割り当てられるメモリ領域は、
その都度違うので、
&x と &p の値は図とは異なります。

ポインタを使った変数値の書き換え

●ポインタ変数に「＊」を付けることで、
その"中身"にアクセスできる。　　・・・　値を書き換えることもできる！

https://paiza.io/projects/vppm6NdYHnlABJfyZWrV6g

```
// Example 04-04
#include <stdio.h>
int main(void){
    char x = 3;
    char *p = &x;

    printf("x = %d¥n", x);          // xの値は3
    printf("(*p) = %d¥n¥n", (*p)); // (*p) はxの値そのもの

    (*p) = 100;                 // pの指すアドレスの中身を100に
    printf("x = %d¥n", x);  // xの中身が書き換わっている

    return 0;
}
```

x = …
の代入は一切
行われていないのに、
xの値が書き換えられた！

「x」と「(*p)」は、
メモリ上の全く同じ場所を
見ていることになる。
➡ xへの代入・参照と、
(*p)への代入・参照は
完全に等価。
一方を書き換えると、
もう一方も書き換わる。
(同じデータを見ている)

ってことは・・・

●関数の引数として、変数アドレスを渡すと…

➡ 関数内で、呼び出し元の変数の"値"を書き換えることができる！

```
// Example 04-05
#include <stdio.h>

void func(int a);      // プロトタイプ宣言
void funcP(int *p);    // プロトタイプ宣言

int main(void){
    int x = 5;

    printf("1: x = %d¥n", x);

    func(x);
    printf("2: x = %d¥n", x);

    funcP(&x);         // 変数のアドレスを渡す
    printf("3: x = %d¥n", x);

    return 0;
}
```

通常の変数の場合の「値渡し」に対して、
アドレスで呼び出す場合を
【参照渡し】と呼ぶ

https://paiza.io/projects/VnsJEmlxUA7yWf37ie0MTg

```
void func(int a) {          // 値渡し
    a *=  2;    // a = a *2; と同じ。値を2倍にする。
    return;
}

void funcP(int *p) {        // 参照渡し
    (*p) *=  2;    // (*p) = (*p) *2; と同じ。値を2倍にする。
    return;
}
```

参考: ポインタの型の違いって･･･？

- int*型 も char*型 も double*型 も、
  あるいは void*型 であっても、アドレス値が入っているだけ
  ･･･ どれも同じ長さの整数値

- 「*」を付けて"値"を取り出す時に、どう解釈されるか？が異なる。
  - 例: char*型なら、格納されているアドレス値から1byte分のbit列を「char型」と解釈した"値"の意味になる。 ※sizeof(char) == 1
  - 例: double*型なら、格納されているアドレス値から8byte分のbit列を「double型」と解釈した"値"の意味になる。※sizeof(double) == 8

参考

https://paiza.io/projects/lJfq_hDjIGWhTolk8S7KmA

```
// Example 04-06
#include <stdio.h>
int main(void){
    int x = 100;

    int   *ip = &x;   // int型ポインタとして、xのアドレスを入れている
    float *fp = &x;   // float型ポインタとして、xのアドレスを入れている(入る値は同じ)

    printf("*(int *)   [%p] = %d¥n", ip, *ip);      // アドレス値は一緒だが
    printf("*(float *) [%p]= %d¥n", fp, *fp);       // デコードされる値が違う

    return 0;
}
```

演習(2)

●Example: 04-04, 04-05 を実行(+改造等)して、
ポインタの動作について実際に確認してみましょう。

ポインタを使った変数値の書き換え

15

●ポインタ変数に「*」を付けることで、
その"中身"にアクセスできる。　　・・・　値を書き換えることもできる!

https://paiza.io/projects/vppm6NdYHnlABJfyZWrV6g

```
// Example 04-04
#include <stdio.h>
int main(void){
    char x = 3;
    char *p = &x;

    printf("x = %d¥n", x);  // xの値は3

    (*p) = 100;             // pの指すアドレスの中身を100に
    printf("x = %d¥n", x);  // xの中身が書き換わっている

    return 0;
}
```

x = …
の代入は一切
行われていないのに、
xの値が書き換えられた!

ってことは・・・

16

●関数の引数として、変数アドレスを渡すと…
➡ 関数内で、呼び出し元の変数の"値"を書き換えることができる!

```
// Example 04-05
#include <stdio.h>

void func(int a);     // プロトタイプ宣言
void funcP(int *p);   // プロトタイプ宣言

int main(void){
    int x = 5;

    printf("1: x = %d¥n", x);

    func(x);
    printf("2: x = %d¥n", x);

    funcP(&x);      // 変数のアドレスを渡す
    printf("3: x = %d¥n", x);

    return 0;
}
```

通常の変数の場合の「値渡し」に対して、
アドレスで呼び出す場合を
【参照渡し】と呼ぶ

https://paiza.io/projects/VnsJEmlxUA7yWf37ie0MTg

```
void func(int a) {        // 値渡し
    a *=  2;   // a = a *2; と同じ。値を2倍にする。
    return;
}

void funcP(int *p) {      // 参照渡し
    (*p) *=  2;   // (*p) = (*p) *2; と同じ。値を2倍にする。
    return;
}
```

●第二部①●
配列とポインタ

これが分かればOK

https://paiza.io/projects/StgEvUtk-9su5KR4WYjMng

```c
// Example 05-01
#include <stdio.h>
int main(void){
    char str[] = "NMLKJIHGFEDCBA";
    char *p1 = str;
    int *p2 = str;

    printf("str = %s¥n", str);
    printf("str    : %p¥n", str);
    printf("&str[0]: %p¥n", &str[0]);
    printf("p1 = %p¥n", p1);
    printf("p2 = %p¥n", p2);
    printf("sizeof(char) = %d, ", sizeof(char));
    printf("sizeof(int) = %d¥n¥n", sizeof(int));

    printf("str = %s¥n", str);
    printf("str[3] = %c¥n", str[3]);
    printf("*str+3 = %c¥n", *str+3);
    printf("*(str+3) = %c¥n¥n", *(str+3));

    printf("p1 = %s¥n", p1);
    printf("p1[3] = %c¥n", p1[3]);
    printf("*p1+3 = %c¥n", *p1+3);
    printf("*(p1+3) = %c¥n¥n", *(p1+3));

    printf("p2 = %s¥n", p2);
    printf("p2[3] = %c¥n", p2[3]);
    printf("*p2+3 = %c¥n", *p2+3);
    printf("*(p2+3) = %c¥n", *(p2+3));
    return 0;
}
```

実行例:

```
str = NMLKJIHGFEDCBA
str    : 0x7ffcc758cb00
&str[0]: 0x7ffcc758cb00
p1 = 0x7ffcc758cb00
p2 = 0x7ffcc758cb00
sizeof(char) = 1, sizeof(int) = 4

str = NMLKJIHGFEDCBA
str[3] = K
*str+3 = Q
*(str+3) = K

p1 = NMLKJIHGFEDCBA
p1[3] = K
*p1+3 = Q
*(p1+3) = K

p2 = NMLKJIHGFEDCBA
p2[3] = B
*p2+3 = Q
*(p2+3) = B
```

配列の復習 (1)　～宣言と初期化～

●宣言のみだと、値は不定

- `int a[5];` ➡ 中身は何か分からない(ただし `static` の場合は `0` に初期化される)
  裏技的?だが、0で初期化したい場合は、 `int a[5] = {0};` などとする。

●宣言時のみ、{}で列挙して **初期化** できる。

- `int a[5] = {1,2,3,4,5};` ➡ `a[0]==1, a[1]==2, … , a[4]==5`
  `int a[5] = {1,2};` ➡ `a[0]==1, a[1]==2, a[2]==0, a[3]==0, a[4]==0`
  `int a[] = {1,2,3,4,5};` ➡ `int a[5]={1,2,3,4,5};` と全く同じ
  ・・・要素数を省略して宣言すると、右辺の要素数に合わせたことになる。

●文字列リテラルでの初期化

- `char str[] = "ABC";` ➡ `char str[4] = {'A','B','C','¥0'};` と同じ
  ※A,B,Cの3文字に加え、文字列の終わりを示す、ヌル文字('¥0')を含めた 4 要素分を宣言したことになる。

配列の復習 (2)　～配列は連続領域～

●配列は、実行時に、メモリ上に **必ず連続領域** として確保されます。

```
int data2[10]={0,2,4,6,8,10,12,14,16,18};
```

と宣言した場合の例。

※ sizeof(int) == 4 byte

アドレス(番地):　中身

+4

0x7ffcb8788f10:	0x00 0x00 0x00 0x00
0x7ffcb8788f14:	0x00 0x00 0x00 0x02
0x7ffcb8788f18:	0x00 0x00 0x00 0x04
0x7ffcb8788f1c:	0x00 0x00 0x00 0x06
0x7ffcb8788f20:	0x00 0x00 0x00 0x08
0x7ffcb8788f24:	0x00 0x00 0x00 0x0A
0x7ffcb8788f28:	0x00 0x00 0x00 0x0C
0x7ffcb8788f2c:	0x00 0x00 0x00 0x0E
0x7ffcb8788f30:	0x00 0x00 0x00 0x10
0x7ffcb8788f34:	0x00 0x00 0x00 0x12

int 1つが4byteなので、"変数1つ"が「アドレス4つ分のメモリ領域」を使っている

【ちゃんと書くとこんな感じ･･･】

アドレス(番地): 中身

0x7ffcb8788f10:	0x00
0x7ffcb8788f11:	0x00
0x7ffcb8788f12:	0x00
0x7ffcb8788f13:	0x00
0x7ffcb8788f14:	0x00
0x7ffcb8788f15:	0x00
0x7ffcb8788f16:	0x00
0x7ffcb8788f17:	0x02
0x7ffcb8788f18:	0x00
0x7ffcb8788f19:	0x00
0x7ffcb8788f1a:	0x00
0x7ffcb8788f1b:	0x04 …

※ビッグエンディアンの場合

配列(1)

```
char *p = data;
```
と
```
char *p;
p = data;
```
が同じ意味なので、混乱しないように！


```
char data[10]={0,2,4,6,8,10,12,14,16,18};
char *p = data;
```
…と宣言した場合の例。

※ sizeof(char) = 1 byte

- このとき、単に data と書くと、
  これは 先頭要素のアドレス を意味する。…[0x7ffc2b03bc70]
  つまり ➡ data == &data[0]
- また、p に data を代入しているので、 …[0x7ffc2b03bc70]
  ➡ p == &data[0]
- このとき、 p+1 と書くと、 …[0x7ffc2b03bc71]
  (char型の)1要素分後ろのアドレス を意味する。
  つまり ➡ p+1 == &data[1]

アドレス(番地):	中身	
0x7ffc2b03bc70:	0x00	← data[0]
0x7ffc2b03bc71:	0x02	← data[1]
0x7ffc2b03bc72:	0x04	← data[2]
0x7ffc2b03bc73:	0x06	← data[3]
0x7ffc2b03bc74:	0x08	← data[4]
0x7ffc2b03bc75:	0x0A	← data[5]
0x7ffc2b03bc76:	0x0C	← data[6]
0x7ffc2b03bc77:	0x0E	← data[7]
0x7ffc2b03bc78:	0x10	← data[8]
0x7ffc2b03bc79:	0x12	← data[9]

配列(2)


```
int data[10]={0,2,4,6,8,10,12,14,16,18};
int *p = data;
```

…と宣言した場合の例。

※ sizeof(int) = 4 byte

- このとき、単に data と書くと、
  これは 先頭要素のアドレス を意味する。…[0x7ffc2b03bc70]
  つまり ➡ p == data == &data[0]

- このとき、 p+1 と書くと、 …[0x7ffc2b03bc74]
  (int型の)1要素分後ろのアドレス を意味する。
  つまり ➡ p+1 == &data[1]

- このとき、 p+2 と書くと、 …[0x7ffc2b03bc78]
  (int型の)2要素分後ろのアドレス を意味する。
  つまり ➡ p+2 == &data[2]

アドレス(番地):	中身	
0x7ffc2b03bc70:	0x00000000	← data[0]
0x7ffc2b03bc74:	0x00000002	← data[1]
0x7ffc2b03bc78:	0x00000004	← data[2]
0x7ffc2b03bc7c:	0x00000006	← data[3]
0x7ffc2b03bc80:	0x00000008	← data[4]
0x7ffc2b03bc84:	0x0000000A	← data[5]
0x7ffc2b03bc88:	0x0000000C	← data[6]
0x7ffc2b03bc8c:	0x0000000E	← data[7]
0x7ffc2b03bc90:	0x00000010	← data[8]
0x7ffc2b03bc94:	0x00000012	← data[9]

配列　まとめ


```
int data[10]={0,2,4,6,8,10,12,14,16,18};
int *p = data;
```

・・・と宣言した場合の例。

※ sizeof(int) = 4 byte

●配列とポインタの対応

	配列での書き方	ポインタでの書き方
値	data[0]	*p
	data[1]	*(p+1)
	data[2]	*(p+2)
	…	…
アドレス	data	p
	&data[0]	p
	&data[1]	p+1
	&data[2]	p+2
	…	…
	data には代入は出来ない(参照のみ)	p には代入ができる(ポインタ変数)

p+1 は、
1つ先のアドレスではなく、
【要素1つ分先のアドレス】
であることに注意！

この例では、「int *p」と宣言されたポインタなので、
+1 すると、sizeof(int) の分先のメモリを指すことになる。

※完全に同じ意味になる書き方が2種類あると思って(ほぼ)良く、
上記の宣言の仕方で、 *(data+1) や p[1] のように
アクセスしても、正しく動作する。
(このことは、 data と p が等価であることからもわかる)

ただし、 data は、既にメモリのどこかに確保された配列の先頭アドレスを意味するので、参照しか出来ない。(つまり data = … のように代入は不可能。一方、p は p = … のように代入が可能。)

https://paiza.io/projects/UXEeykecZeYNVE8_U9UD4A

```
// Example 05-02
#include <stdio.h>

int main(void){
    char data[10]={0,2,4,6,8,10,12,14,16,18};
    char *p = data;

    int n = 3;

    // ①
    printf(" data   : %p,  *data    = %d¥n", data    , *data);
    for (int i=0; i<n; i++)
        printf("&data[%d]: %p,   data[%d] = %d¥n", i, &data[i], i, data[i]);

    // ②
    printf("¥n data   : %p,  *data    = %d¥n", data    , *data);
    for (int i=0; i<n; i++)
        printf("data+%d  : %p, *(data+%d) = %d¥n", i, data+i, i, *(data+i));

    // ③
    printf("¥n p   : %p,  *p    = %d¥n", p    , *p);
    for (int i=0; i<n; i++)
        printf("p+%d: %p, *(p+%d) = %d¥n", i, p+i, i, *(p+i));

    // ④
    printf("¥n p   : %p,  *p    = %d¥n", p    , *p);
    for (int i=0; i<n; i++)
        printf("&p[%d]: %p, p[%d] = %d¥n", i, &p[i], i, p[i]);

    return 0;
}
```

変数多すぎでわからん！ってなりそうな人は、
まずこちらを見てください。(①～④が対応)
※左のコードは、これの数値の部分を
ループカウンタiに置き換えただけ

```
/*  // ①
    printf("&data[0]: %p,   data[0] = %d¥n", &data[0], data[0]);
    printf("&data[0]: %p,   data[0] = %d¥n", &data[0], data[0]);
    printf("&data[1]: %p,   data[1] = %d¥n", &data[1], data[1]);
    printf("&data[2]: %p,   data[2] = %d¥n", &data[2], data[2]); */

/*  // ②
    printf(" data   : %p,  *data    = %d¥n", data    , *data);
    printf("data+1  : %p, *(data+1) = %d¥n", data+1, *(data+1));
    printf("data+2  : %p, *(data+2) = %d¥n", data+2, *(data+2)); */


/*  // ③
    printf("p  : %p, *p     = %d¥n", p, *p);
    printf("p+1: %p, *(p+1) = %d¥n", p+1, *(p+1));
    printf("p+2: %p, *(p+2) = %d¥n", p+2, *(p+2)); */

/*  // ④
    printf("&p[0]: %p, p[0] = %d¥n", &p[0], p[0]);
    printf("&p[1]: %p, p[1] = %d¥n", &p[1], p[1]);
    printf("&p[2]: %p, p[2] = %d¥n", &p[2], p[2]); */
```

･･･で、最初のに戻る。(1)

https://paiza.io/projects/StgEvUtk-9su5KR4WYjMng

```
// Example 05-01
#include <stdio.h>
int main(void){
	char str[] = "NMLKJIHGFEDCBA";
	char *p1 = str;
	int *p2 = str;

	printf("str = %s¥n", str);
	printf("str  : %p¥n", str);
	printf("&str[0]: %p¥n", &str[0]);
	printf("p1 = %p¥n", p1);
	printf("p2 = %p¥n", p2);
	printf("sizeof(char) = %d, ", sizeof(char));
	printf("sizeof(int) = %d¥n¥n", sizeof(int));

	printf("str = %s¥n", str);
	printf("str[3] = %c¥n", str[3]);
	printf("*str+3 = %c¥n", *str+3);
	printf("*(str+3) = %c¥n¥n", *(str+3));

	printf("p1 = %s¥n", p1);
	printf("p1[3] = %c¥n", p1[3]);
	printf("*p1+3 = %c¥n", *p1+3);
	printf("*(p1+3) = %c¥n¥n", *(p1+3));

	printf("p2 = %s¥n", p2);
	printf("p2[3] = %c¥n", p2[3]);
	printf("*p2+3 = %c¥n", *p2+3);
```

```
str = NMLKJIHGFEDCBA
str    : 0x7ffcc758cb00
&str[0]: 0x7ffcc758cb00
p1 = 0x7ffcc758cb00
p2 = 0x7ffcc758cb00
sizeof(char) = 1, sizeof(int) = 4

str = NMLKJIHGFEDCBA
str[3] = K
*str+3 = Q
*(str+3) = K

p1 = NMLKJIHGFEDCBA
p1[3] = K
*p1+3 = Q
*(p1+3) = K

p2 = NMLKJIHGFEDCBA
```

･･･で、最初のに戻る。(2)

https://paiza.io/projects/StgEvUtk-9su5KR4WYjMng

```
// Example 05-01
#include <stdio.h>
int main(void){
	char str[] = "NMLKJIHGFEDCBA";
	char *p1 = str;
	int *p2 = str;

	printf("str = %s¥n", str);
	printf("str  : %p¥n", str);
	printf("&str[0]: %p¥n", &str[0]);
	printf("p1 = %p¥n", p1);
	printf("p2 = %p¥n", p2);
	printf("sizeof(char) = %d, ", sizeof(char));
	printf("sizeof(int) = %d¥n¥n", sizeof(int));

	printf("str = %s¥n", str);
	printf("str[3] = %c¥n", str[3]);
	printf("*str+3 = %c¥n", *str+3);
	printf("*(str+3) = %c¥n¥n", *(str+3));

	printf("p1 = %s¥n", p1);
	printf("p1[3] = %c¥n", p1[3]);
	printf("*p1+3 = %c¥n", *p1+3);
	printf("*(p1+3) = %c¥n¥n", *(p1+3));

	printf("p2 = %s¥n", p2);
	printf("p2[3] = %c¥n", p2[3]);
	printf("*p2+3 = %c¥n", *p2+3);
```

```
str = NMLKJIHGFEDCBA
str    : 0x7ffcc758cb00
&str[0]: 0x7ffcc758cb00
p1 = 0x7ffcc758cb00
p2 = 0x7ffcc758cb00
sizeof(char) = 1, sizeof(int) = 4

str = NMLKJIHGFEDCBA
str[3] = K
*str+3 = Q
*(str+3) = K

p1 = NMLKJIHGFEDCBA
p1[3] = K
*p1+3 = Q
*(p1+3) = K

p2 = NMLKJIHGFEDCBA
p2[3] = B
```

･･･で、最初のに戻る。(3)

```
        printf("str = %s¥n", str);
        printf("str  : %p¥n", str);
        printf("&str[0]: %p¥n", &str[0]);
        printf("p1 = %p¥n", p1);
        printf("p2 = %p¥n", p2);
        printf("sizeof(char) = %d, ", sizeof(char));
        printf("sizeof(int) = %d¥n¥n", sizeof(int));

        printf("str = %s¥n", str);
        printf("str[3] = %c¥n", str[3]);
        printf("*str+3 = %c¥n", *str+3);
        printf("*(str+3) = %c¥n¥n", *(str+3));

        printf("p1 = %s¥n", p1);
        printf("p1[3] = %c¥n", p1[3]);
        printf("*p1+3 = %c¥n", *p1+3);
        printf("*(p1+3) = %c¥n¥n", *(p1+3));

        printf("p2 = %s¥n", p2);
        printf("p2[3] = %c¥n", p2[3]);
        printf("*p2+3 = %c¥n", *p2+3);
        printf("*(p2+3) = %c¥n", *(p2+3));

        return 0;
}
```

https://paiza.io/projects/StgEvUtk-9su5KR4WYjMng

```
p1 = 0x7ffcc758cb00
p2 = 0x7ffcc758cb00
sizeof(char) = 1, sizeof(int) = 4

str = NMLKJIHGFEDCBA
str[3] = K
*str+3 = Q
*(str+3) = K

p1 = NMLKJIHGFEDCBA
p1[3] = K
*p1+3 = Q
*(p1+3) = K

p2 = NMLKJIHGFEDCBA
p2[3] = B
*p2+3 = Q
*(p2+3) = B
```

●第二部②●
文字列とポインタ

文字列

●**文字**は　整数(文字コード)

➢「A」の文字コードは　'A'　と記述できる。　　➡　　'A' == 65

●**文字列**は　整数配列('¥0'で文字列の終端を示す)

➢以下の①～③は全く同じ。

```
① char str[] = "ABCDE";
② char str[6] = {'A', 'B', 'C', 'D', 'E', '¥0'};
③ char str[6] = { 65,  66,  67,  68,  69,  0};
```

● "ABCDE" は **文字列リテラル** ･･･ メモリ上のどこかに定数として配置される

➢文字列リテラルは、数値定数(int a = 10; としたときの「10」)と同様に、書き換え不可

➢char str[] = "ABCDE";　　･･･ 配列として確保した領域に、文字列リテラルの中身がコピーされる　➡　str は配列なので、書き換え可能

char *strP = "ABCDE";　　･･･ 文字列リテラルの先頭要素のアドレスが代入される
➡　strP は書き換え禁止領域を指している

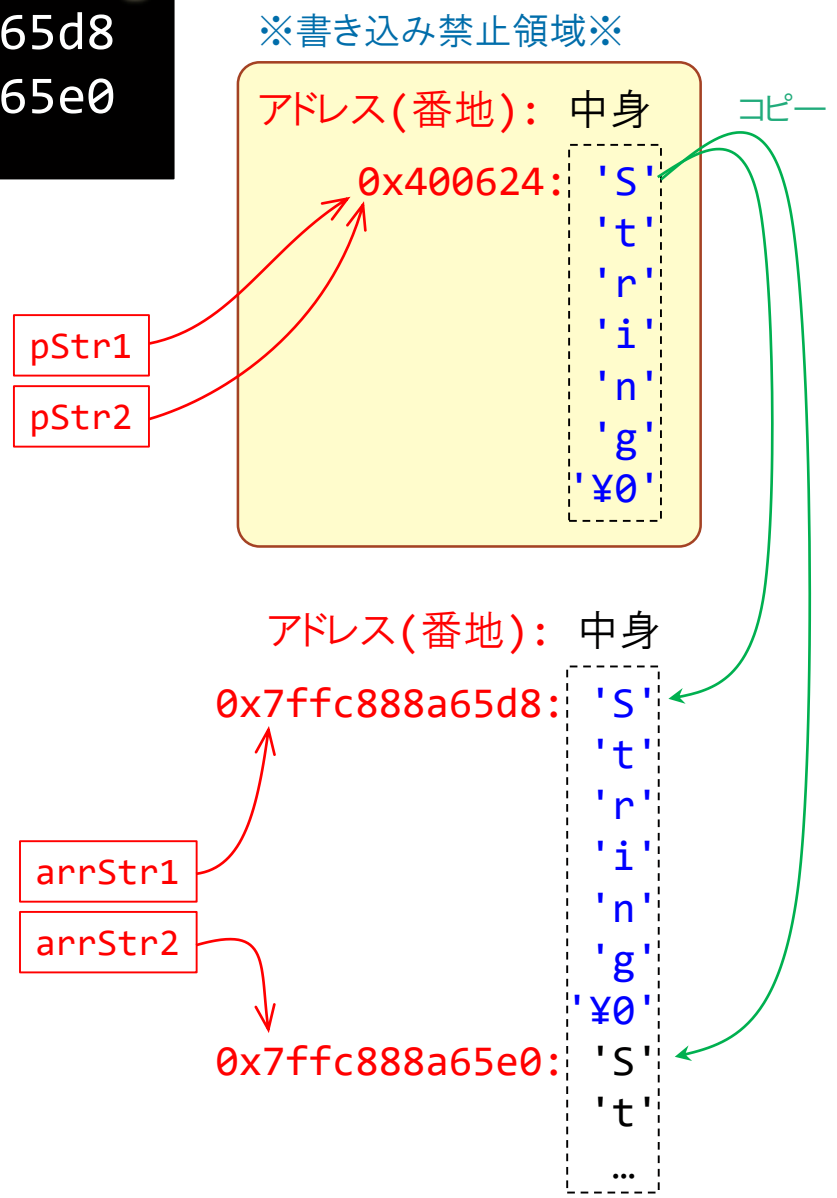
実行例:

```
pStr1   = String : 0x400624
pStr2   = String : 0x400624
arrStr1 = string : 0x7ffc888a65d8
arrStr2 = String : 0x7ffc888a65e0
```

1つだけ生成されている
(同じアドレス)

```c
// Example 05-03
#include <stdio.h>

int main(void) {
    char *pStr1    = "String";  // 全く同じ文字列リテラルは、
    char *pStr2    = "String";  // 1つだけしか生成されない場合がある
    char arrStr1[] = "String";  // (読み出ししかされないので1つあれば
    char arrStr2[] = "String";  //  問題がない)


    // pStr1[0] = 's';          // 書き込み禁止領域に書き込もうとするので、
                                // 実行時エラーとなる
    arrStr1[0] = 's';       // 配列の中身を書き換えるだけなので、
                            // 実行時エラーにはならない

    printf("pStr1   = %s : %p¥n", pStr1, pStr1);
    printf("pStr2   = %s : %p¥n", pStr2, pStr2);
    printf("arrStr1 = %s : %p¥n", arrStr1, arrStr1);
    printf("arrStr2 = %s : %p¥n", arrStr2, arrStr2);

    return 0;
}
```

https://paiza.io/projects/wdlQVni4jKu97DBDaLrm1w



演習(3)

- Example 05-01, Example 05-03 を実行(+改造等)して、動作について実際に確認してみましょう。

これが分かればOK

10

```
// Example 05-01
#include <stdio.h>
int main(void){
    char str[] = "NMLKJIHGFEDCBA";
    char *p1 = str;
    int *p2 = str;

    printf("str = %s¥n", str);
    printf("str    : %p¥n", str);
    printf("&str[0]: %p¥n", &str[0]);
    printf("p1 = %p¥n", p1);
    printf("p2 = %p¥n", p2);
    printf("sizeof(char) = %d, ", sizeof(char));
    printf("sizeof(int) = %d¥n¥n", sizeof(int));

    printf("str = %s¥n", str);
    printf("str[3] = %c¥n", str[3]);
    printf("*str+3 = %c¥n", *str+3);
    printf("*(str+3) = %c¥n¥n", *(str+3));

    printf("p1 = %s¥n", p1);
    printf("p1[3] = %c¥n", p1[3]);
    printf("*p1+3 = %c¥n", *p1+3);
    printf("*(p1+3) = %c¥n¥n", *(p1+3));

    printf("p2 = %s¥n", p2);
    printf("p2[3] = %c¥n", p2[3]);
    printf("*p2+3 = %c¥n", *p2+3);
    printf("*(p2+3) = %c¥n", *(p2+3));
    return 0;
}
```

https://paiza.io/projects/StgEvUtk-9su5KR4WYjMng

実行例:

```
str = NMLKJIHGFEDCBA
str    : 0x7ffcc758cb00
&str[0]: 0x7ffcc758cb00
p1 = 0x7ffcc758cb00
p2 = 0x7ffcc758cb00
sizeof(char) = 1, sizeof(int) = 4

str = NMLKJIHGFEDCBA
str[3] = K
*str+3 = Q
*(str+3) = K

p1 = NMLKJIHGFEDCBA
p1[3] = K
*p1+3 = Q
*(p1+3) = K

p2 = NMLKJIHGFEDCBA
p2[3] = B
*p2+3 = Q
*(p2+3) = B
```

21

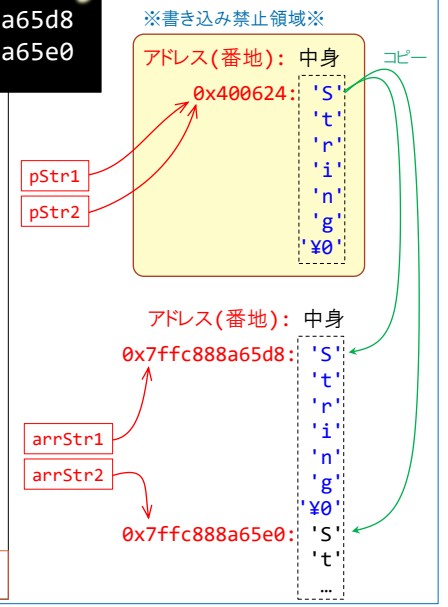
実行例:

```
pStr1   = String : 0x400624
pStr2   = String : 0x400624
arrStr1 = string : 0x7ffc888a65d8
arrStr2 = String : 0x7ffc888a65e0
```

1つだけ生成されている(同じアドレス)

```
// Example 05-03
#include <stdio.h>

int main(void) {
    char *pStr1   = "String";  // 全く同じ文字列リテラルは、
    char *pStr2   = "String";  // 1つだけしか生成されない場合がある
    char arrStr1[] = "String";  // (読み出ししかされないので1つあれば
    char arrStr2[] = "String";  //  問題がない)


    // pStr1[0] = 's';      // 書き込み禁止領域に書き込もうとするので、
                            // 実行時エラーとなる
    arrStr1[0] = 's';    // 配列の中身を書き換えるだけなので、
                         // 実行時エラーにはならない

    printf("pStr1   = %s : %p¥n", pStr1, pStr1);
    printf("pStr2   = %s : %p¥n", pStr2, pStr2);
    printf("arrStr1 = %s : %p¥n", arrStr1, arrStr1);
    printf("arrStr2 = %s : %p¥n", arrStr2, arrStr2);

    return 0;
}
```

https://paiza.io/projects/wdlQVni4jKu97DBDaLrm1w



第一部まとめ

今回のまとめ①

- 変数に割り当てられている "メモリ内の場所=アドレス" は、
  変数名の前に「&」を付けると得られる。
- ポインタ変数は、変数名の前に「*」を付けて宣言する。
- ポインタ変数の
  変数名の前に「*」を付けてアクセスすると、
  ポインタ変数に入っている"アドレス値"の
  先にある中身にアクセスできる。

```
char x = 3;
char *p = &x;
printf("x = %d¥n", (*p));
(*p) = 5;
printf("x = %d¥n", x);
```

- ポインタを使うと、別の変数の中身を
  書き換えることができる。

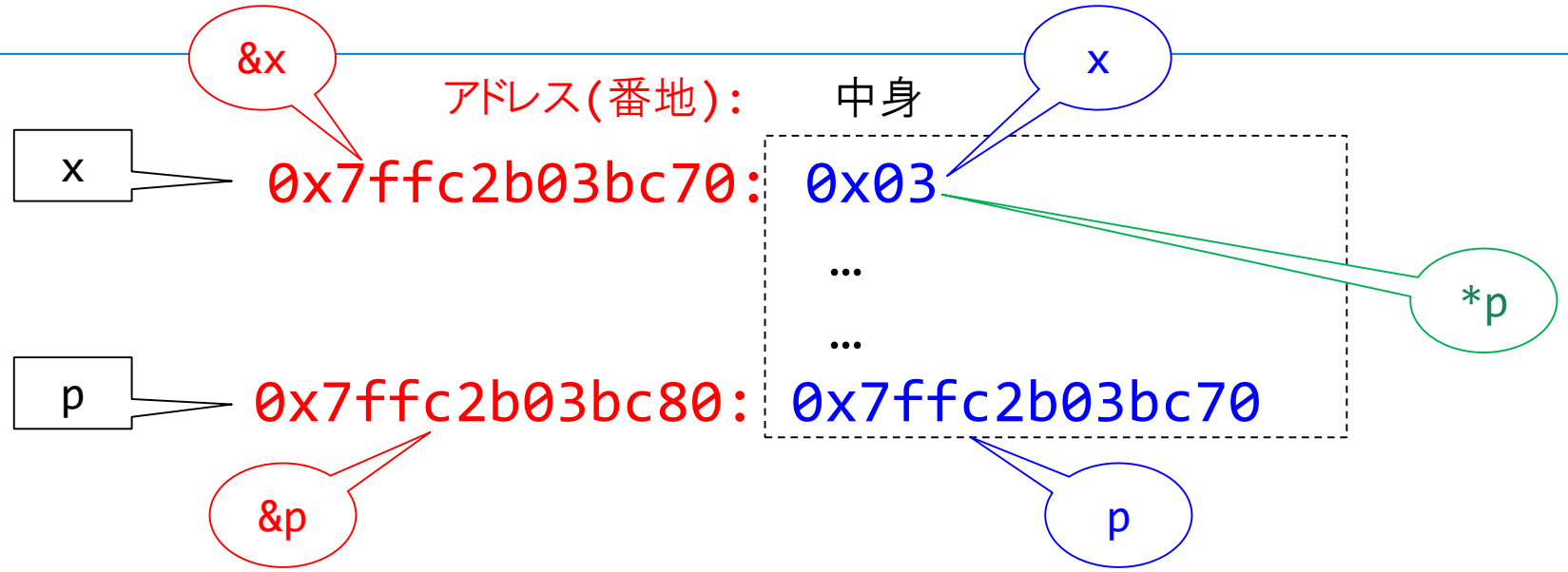
```
void func(int *x, int *y);
```

- 関数の引数をポインタ変数にすることで、「参照渡し」となり、呼び出した関数内で、
  呼び出し元の(複数の)変数の"中身"を直接書き換えることが可能になる。

今回のまとめ②


```
char x = 3;
char *p = &x;
```

・・・とした場合。



x	・・・ 3	「値(char値)」(中身)
&x	・・・ 0x7ffc2b03bc70	xが入っている「アドレス」
p	・・・ 0x7ffc2b03bc70	「値(アドレス値)」(中身)
&p	・・・ 0x7ffc2b03bc80	pが入っている「アドレス」
*p	・・・ 3	「アドレス値が指す番地の"中身"」

第二部①②まとめ

今回のまとめ①


```
int data[10]={0,2,4,6,8,10,12,14,16,18};
int *p = data;
```

・・・と宣言した場合の例。

※ sizeof(int) = 4 byte

●配列とポインタの対応

	配列での書き方	ポインタでの書き方
値	data[0]	*p
	data[1]	*(p+1)
	data[2]	*(p+2)
	…	…
アドレス	data	p
	&data[0]	p
	&data[1]	p+1
	&data[2]	p+2
	…	…
	data には代入は出来ない(参照のみ)	p には代入ができる(ポインタ変数)

p+1 は、
1つ先のアドレスではなく、
【要素1つ分先のアドレス】
であることに注意！

この例では、「int *p」と宣言されたポインタなので、
+1 すると、sizeof(int) の分先のメモリを指すことになる。

※完全に同じ意味になる書き方が2種類あると思って(ほぼ)良く、
上記の宣言の仕方で、 *(data+1) や p[1] のように
アクセスしても、正しく動作する。
(このことは、 data と p が等価であることからもわかる)

ただし、 data は、既にメモリのどこかに確保された配列の先頭アドレスを意味するので、参照しか出来ない。(つまり data = … のように代入は不可能。一方、p は p = … のように代入が可能。)

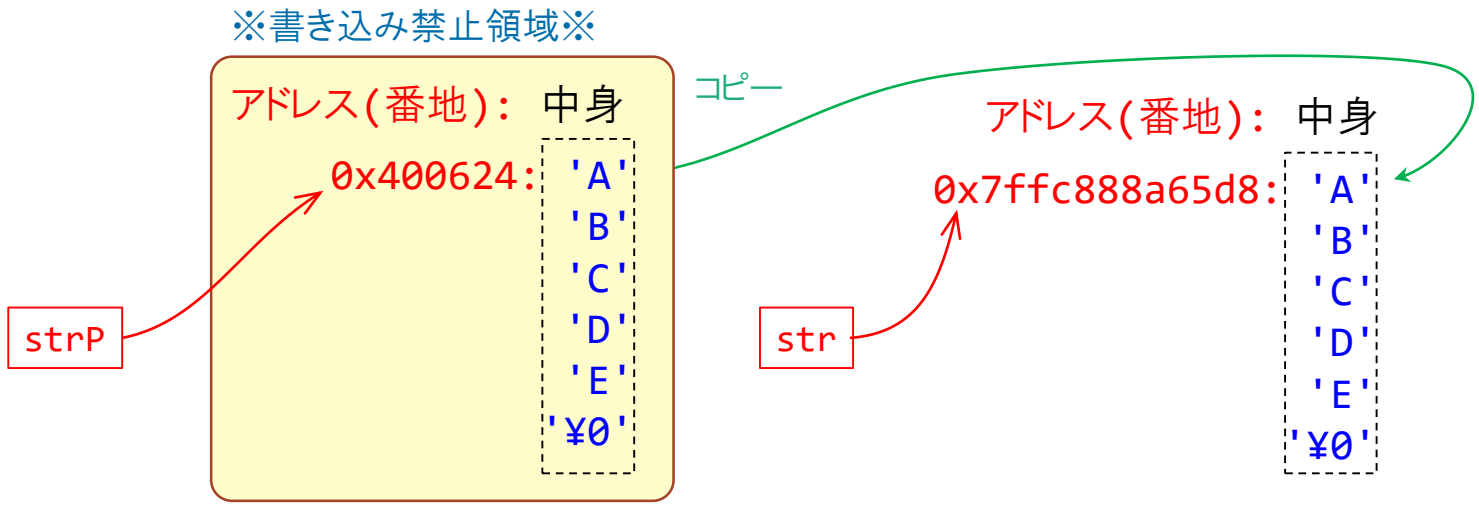
今回のまとめ②

●文字列

➢"ABCDE" は **文字列リテラル** ･･･ メモリ上のどこかに定数として配置される

- 文字列リテラルは、数値定数(`int a = 10;` としたときの「`10`」)と同様に、書き換え不可
- `char str[] = "ABCDE";` ･･･ 配列として確保した領域に、文字列リテラルの中身がコピーされる ➡ `str` は配列なので、書き換え可能

`char *strP = "ABCDE";` ･･･ 文字列リテラルの先頭要素のアドレスが代入される ➡ `strP` は書き換え禁止領域を指している



用意してある main() 関数内は、
（「???」部分意外は）変更しないこと。

課題No.04

~~提出課題としますので、授業中にできなかった分は宿題として、~~
~~（実行結果の例も添えて、）次週までに提出のこと。~~

➡ Mooodleから直接提出して下さい。

課題No.04-A

●void wasa(int *wa, int *sa, int x, int y);
wa を x+y の値に、 sa を x-y の値に書き換える関数を作りなさい。

https://paiza.io/projects/fcrkFfJ9AO1CgNwSdSjPdQ

```
// No.04-A
#include <stdio.h>
void wasa(int *wa, int *sa, int x, int y); // プロトタイプ宣言

int main(void){
    int x, y, wa, sa;
    scanf("%d", &x);
    scanf("%d", &y);
    printf("x = %d, y = %d¥n", x, y);
    wasa(????????????????); // 適切な引数の指定をする
    printf("wa = x+y = %d¥nsa = x-y = %d¥n", wa, sa);
    return 0;
}

// この後に wasa() 関数を書く
```

実行例1:
※標準入力から、10 と 20 を入力

```
x = 10, y = 20
wa = x+y = 30
sa = x-y = -10
```

実行例2:
※標準入力から、-30 と 50 を入力

```
x = -30, y = -50
wa = x+y = -80
sa = x-y = 20
```

課題No.04-B

●引数に指定した2つのint型変数の値を入れ替える関数
void my_swap(??(型名)?? a, ??(型名)?? b); を作りなさい。

```
// No.04-B                          https://paiza.io/projects/
#include <stdio.h>                  qq6WhBRIZTq3T-MOHNDTqg

void my_swap(??? a, ??? b); // プロトタイプ宣言(???を埋める)

int main(void){
    int a, b;
    scanf("%d", &a);
    scanf("%d", &b);
    printf("a=%d, b=%d¥n", a, b); // 入れ替え前
    my_swap(???a, ???b); // ???に適切な指定をして呼び出す
    printf("a=%d, b=%d¥n", a, b); // 入れ替え後
    return 0;
}

// この後に my_swap() 関数を書く
```

実行例1:
※標準入力から、10 と 30 を入力

```
a=10, b=30
a=30, b=10
```

実行例2:
※標準入力から、12 と -30 を入力

```
a=12, b=-30
a=-30, b=12
```

課題No.04-C

●引数に指定したint型変数に、標準入力から読み込んだ整数値を入力する
`void scanf_d(??(型名)?? a);` を作りなさい。

➢ 作った関数内で scanf() を用いるものとする ※main()内ではscanf()を使用しない!!

https://paiza.io/projects/9JMYwfzLxjLfOX6QCl3f8w

```
// No.04-C
#include <stdio.h>

void scanf_d(??? a); // プロトタイプ宣言(???を埋める)

int main(void){
    int x;
    scanf_d(???x);    // ???に適切な指定をして呼び出す
    printf("a=%d¥n", x);
    return 0;
}

// この後に scanf_d() 関数を書く
```

実行例1:
※標準入力から、100 を入力

```
a=100
```

実行例2:
※標準入力から、 -33 を入力

```
a=-33
```

以下は、たぶん次回やります。
(課題No.06も)

構造体 – Structures –

構造体とは？ (1)

●1つ以上の変数を、ひとまとめにしたもの　---　自分で作る「型」

```
// Example 06-01
#include <stdio.h>

struct Person {
    char name[20];
    char age;
    double weight;   // kg
    double height;   // cm
};

int main(void){
    struct Person a;
    double bmi;

    scanf("%s", a.name);
    scanf("%d", &a.age);
    scanf("%lf", &a.weight);
    scanf("%lf", &a.height);
```

【構造体の基本①】

●作る時：**struct 構造体タグ名 { メンバ変数のリスト… };**

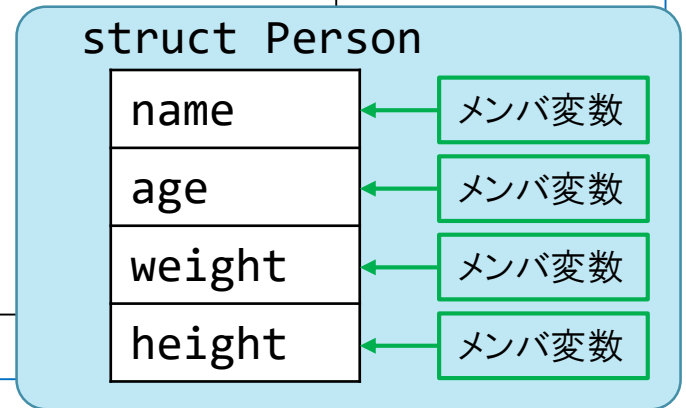
これが新しい「型名」のような扱いになる

●使う時

- ➢宣言：**struct 構造体タグ名** 変数名;
- ➢メンバ変数へのアクセス：
  変数名.メンバ変数名

https://paiza.io/projects/DUbqCIr31TgNUDQhDsTwhA

```
    bmi = a.weight / ((a.height/100) * (a.height/100));

    printf("Name=%s¥n", a.name);
    printf("Age=%d¥n", a.age);
    printf("BMI=%f¥n", bmi);

    return 0;
}
```



演習(1)

1. Example 06-01 を実行して、丁寧に動作を追ってみる
2. 出席番号として"int型の「id」というメンバ変数"を構造体に追加し、
   その入力(scanf())と出力(printf())を追加して実行してみる

構造体とは？ (1)

●1つ以上の変数を、ひとまとめにしたもの　--- 自分で作る「型」

```
// Example 06-01
#include <stdio.h>

struct Person {
    char name[20];
    char age;
    double weight;  // kg
    double height;  // cm
};

int main(void){
    struct Person a;
    double bmi;

    scanf("%s", a.name);
    scanf("%d", &a.age);
    scanf("%lf", &a.weight);
    scanf("%lf", &a.height);

    bmi = a.weight / ((a.height/100) * (a.height/100));

    printf("Name=%s¥n", a.name);
    printf("Age=%d¥n", a.age);
    printf("BMI=%f¥n", bmi);

    return 0;
}
```

【構造体の基本①】

●作る時：`struct 構造体タグ名 { メンバ変数のリスト… };`

これが新しい「型名」のような扱いになる

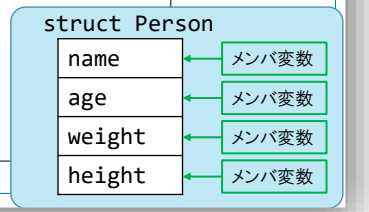
●使う時

➢宣言：`struct 構造体タグ名 変数名;`

➢メンバ変数へのアクセス：

変数名.メンバ変数名

https://paiza.io/projects/DUbqCIr31TgNUDQhDsTwhA



構造体とは？ (2)

●関数の引数や返り値の型にも使える。 --- 複数の値を渡す手段にも使える。

https://paiza.io/projects/BhYF-GfcGjo0Kupkv7cwQQ

```
// Example 06-02
#include <stdio.h>

struct Person {
    char name[20];
    char age;
    double weight;    // kg
    double height;    // cm
};

struct Index {
    double bmi;
    double standard;      // BMI=22
    double biyou;         // BMI=20
    double cinderella;    // BMI=18
};

struct Index calc(struct Person x);
void print(struct Person p, struct Index x);

int main(void){
    struct Person b = {"Hanako", 18, 55, 160};
    struct Index r = calc(b)  ;
    print(b, r);
    return 0;
}
```

【構造体の基本②】

●配列と同様に、宣言と同時であれば初期化可能。

➢ メンバ変数の並び順に列挙する

```
struct Index calc(struct Person x) {
    struct Index ind;
    ind.bmi = x.weight / ((x.height/100) * (x.height/100));
    ind.standard   = x.height/100 * x.height/100 * 22;
    ind.biyou      = x.height/100 * x.height/100 * 20;
    ind.cinderella = x.height/100 * x.height/100 * 18;

    return ind;
}

void print(struct Person p, struct Index x) {
    printf("[%s]¥n", p.name);
    printf("  age: %d¥n", p.age);
    printf("  weight: %.1f kg¥n", p.weight);
    printf("  height: %.1f cm¥n¥n", p.height);

    printf("BMI = %.2f¥n", x.bmi);
    printf("-----------------------------------¥n");
    printf("  standard weight(BMI=22) = %.1f kg¥n", x.standard);
    printf("     biyou weight(BMI=20) = %.1f kg¥n", x.biyou);
    printf("cinderella weight(BMI=18) = %.1f kg¥n", x.cinderella);
    //printf("     model weight(BMI=17) = %.1f kg¥n", x.model);

    return;
}
```

※値渡しでは 全メンバ変数をコピー する必要があるので、巨大な構造体などではオーバーヘッドが大きくなる。

演習(2)

1．Example 06-02 を実行して、丁寧に動作を追ってみる

2．モデル体重を追加

① Index 構造体に、"モデル体重"(double型の「model」というメンバ変数)を追加

② calc()関数内で「BMI=17の場合の体重」として計算し、追加したメンバ変数に保存

③ print()関数内でコメントに
なっている行の「//」を外して、
結果を表示するように修正。

項目(メンバ変数)の増減があっても、
関数の定義の変更は不要であるなど
仕様変更への対応が比較的楽

構造体とは？ (2)

●関数の引数や返り値の型にも使える。 --- 複数の値を渡す手段にも使える。

https://paiza.io/projects/BhYF-GfcGjo0Kupkv7cwQQ

```
// Example 06-02
#include <stdio.h>

struct Person {
    char name[20];
    char age;
    double weight;   // kg
    double height;   // cm
};

struct Index {
    double bmi;
    double standard;      // BMI=22
    double biyou;         // BMI=20
    double cinderella;    // BMI=18
};

struct Index calc(struct Person x);
void print(struct Person p, struct Index x);

int main(void){
    struct Person b = {"Hanako", 18, 55, 160};
    struct Index r = calc(b)  ;
    print(b, r);
    return 0;
}
```

【構造体の基本②】
●配列と同様に、宣言と同時であれば初期化可能。
➢ メンバ変数の並び順に列挙する

```
struct Index calc(struct Person x) {
    struct Index ind;
    ind.bmi = x.weight / ((x.height/100) * (x.height/100));
    ind.standard   = x.height/100 * x.height/100 * 22;
    ind.biyou      = x.height/100 * x.height/100 * 20;
    ind.cinderella = x.height/100 * x.height/100 * 18;

    return ind;
}

void print(struct Person p, struct Index x) {
    printf("[%s]¥n", p.name);
    printf("  age: %d¥n", p.age);
    printf("  weight: %.1f kg¥n", p.weight);
    printf("  height: %.1f cm¥n¥n", p.height);

    printf("BMI = %.2f¥n", x.bmi);
    printf("-----------------------------------¥n");
    printf("  standard weight(BMI=22) = %.1f kg¥n", x.standard);
    printf("     biyou weight(BMI=20) = %.1f kg¥n", x.biyou);
    printf("cinderella weight(BMI=18) = %.1f kg¥n", x.cinderella);
    //printf("     model weight(BMI=17) = %.1f kg¥n", x.model);

    return;
}
```

※値渡しでは 全メンバ変数をコピー する必要があるので、巨大な構造体などではオーバーヘッドが大きくなる。

構造体とは？ (3)

●他の型と同様、**配列**や**ポインタ**も使える。

```
// Example 06-03
#include <stdio.h>
#define N 5
struct Person2 {
    char name[20];
    char age;
};
void func2(struct Person2 *x);

int main(void){
    struct Person2 arr[N]; // 配列で宣言

    for (int i=0; i<N; i++) {
        scanf("%s", arr[i].name); // 06-01と同様
        scanf("%d", &arr[i].age); // 06-01と同様
    }
    for (int i=0; i<N; i++)
        func2(&arr[i]);     // アドレスを渡している

    return 0;
}
```

https://paiza.io/projects/C8F788EpPpv3bYp6Eb1ubg

【構造体の基本③】

●構造体のポインタの場合、

`->` ･･･アロー演算子

でメンバ変数にアクセスできる。

●例えば、 `data->xy` と書くのと、

`(*data).xy` と書くのとは等価。

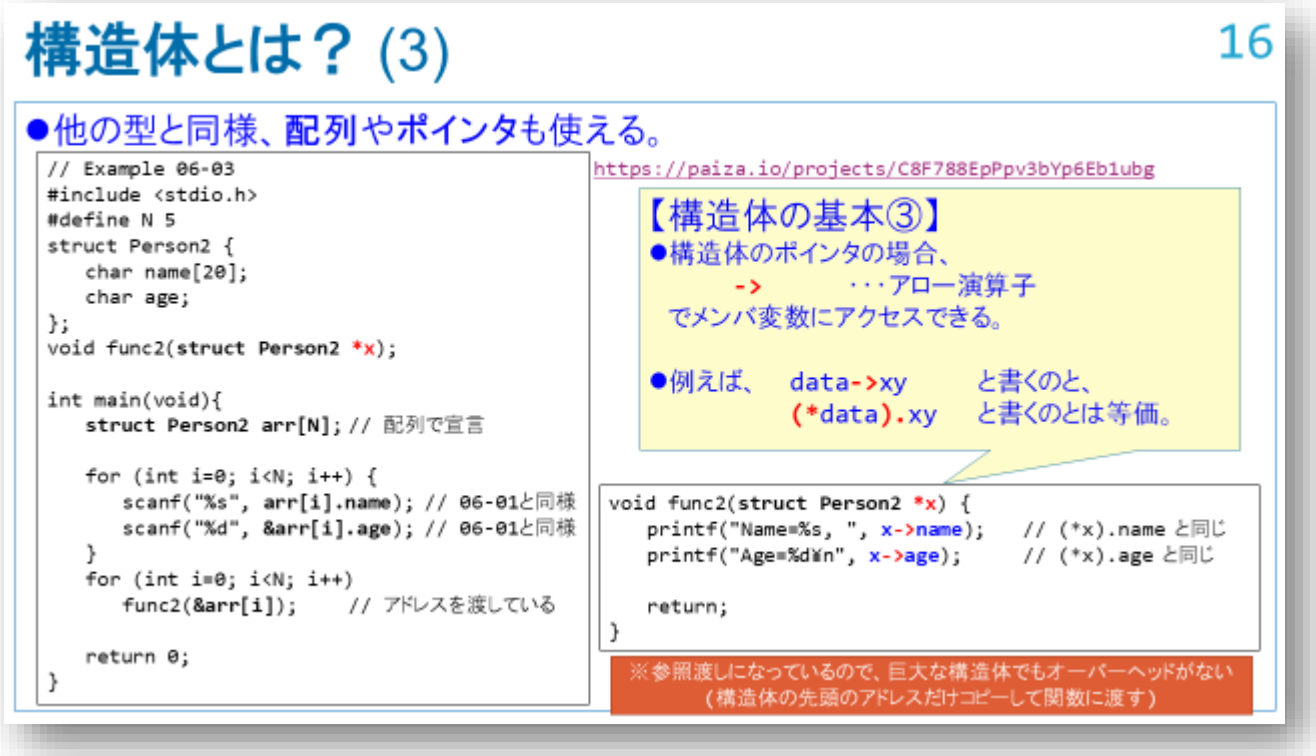
```
void func2(struct Person2 *x) {
    printf("Name=%s, ", x->name);    // (*x).name と同じ
    printf("Age=%d¥n", x->age);      // (*x).age と同じ

    return;
}
```

※参照渡しになっているので、巨大な構造体でもオーバーヘッドがない
（構造体の先頭のアドレスだけコピーして関数に渡す）

演習(3)

1. Example 06-03 を実行して、丁寧に動作を追ってみる
2. func2()の中身を書き換え、「->」を使わない記述に変更してみる

3. 余裕がある人のみ
   - void func3(struct Person2 x);
     というfunc2()と同じ出力を行う関数を作り、
     main関数内でfunc2()を呼び出している行をfunc3()で置き換えてみる。
     ※値渡しで呼び出す



構造体とは？ (3) 16

●他の型と同様、配列やポインタも使える。

```
// Example 06-03
#include <stdio.h>
#define N 5
struct Person2 {
   char name[20];
   char age;
};
void func2(struct Person2 *x);

int main(void){
   struct Person2 arr[N]; // 配列で宣言

   for (int i=0; i<N; i++) {
      scanf("%s", arr[i].name); // 06-01と同様
      scanf("%d", &arr[i].age); // 06-01と同様
   }
   for (int i=0; i<N; i++)
      func2(&arr[i]);     // アドレスを渡している

   return 0;
}
```

https://paiza.io/projects/C8F788EpPpv3bYp6Eb1ubg

【構造体の基本③】
●構造体のポインタの場合、
-> ・・・アロー演算子
でメンバ変数にアクセスできる。

●例えば、 data->xy と書くのと、
(*data).xy と書くのとは等価。

```
void func2(struct Person2 *x) {
   printf("Name=%s, ", x->name);    // (*x).name と同じ
   printf("Age=%d¥n", x->age);      // (*x).age と同じ

   return;
}
```

※参照渡しになっているので、巨大な構造体でもオーバーヘッドがない
（構造体の先頭のアドレスだけコピーして関数に渡す）

その他、構造体に関して・・・

●構造体の名前には、大文字を使う事が多い(慣例的に)

➢基本型等と区別するため、全部大文字にしたり、先頭を大文字にしたりする。
→ PERSON とか Person とか。

●スコープについて

➢メンバ変数の変数名は、その構造体のみに所属する。(他の名前とかぶってもOK)

```
struct StA {
    int a;
    double b;
};

struct StB {
    char a;
    int b;
    double x;
};

int main(void) {
    struct StA a;
    struct StB x;
    a.a = …
    x.b = …
}
・・・・とか全く問題なし。
```

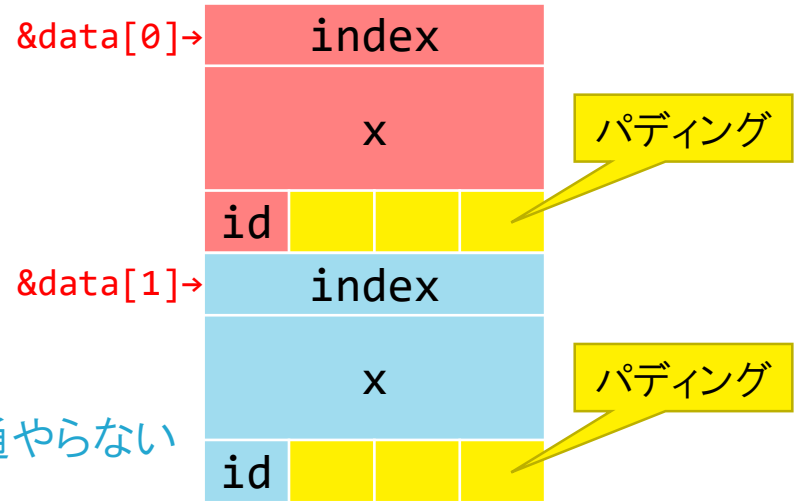
●パディングの話

➢構造体変数を宣言すると、メンバ変数を書いた順に、メモリの中に配置される。

➢ただし、中途半端な大きさの変数は
パディングが追加される場合がある。
(例えば4byte単位になるように追加)

➢コンパイラの設定(アラインメント)
などでも実際の配置が変わる場合が
あるので、アドレス計算(ポインタ演算)でアクセスするような場合は注意!←※普通やらない

```
struct StX {
    int index; // 4byte
    double x;  // 8byte
    char id;   // 1byte
};
…
struct StX data[2];
```



パディング

https://paiza.io/projects/-eCocLQNsHJPj5OPWaoupQ

```
// Example 06-04
#include <stdio.h>

struct StA {
    int a;
    char ch[4];
};

struct StB {
    int a;
    char ch[2]; // 2byte分、パディングが入る
};

int main(void){
    struct StA a;
    struct StB b;

    printf("sizeof(int)    = %d byte¥n", sizeof(int));
    printf("sizeof(char)   = %d byte¥n¥n", sizeof(char));

    printf("sizeof(a.a)    = %d byte¥n", sizeof(a.a));
    printf("sizeof(a.ch)   = %d byte¥n", sizeof(a.ch));
    printf("sizeof(a.a)+sizeof(b.ch)   = %d byte¥n", sizeof(a.a)+sizeof(a.ch));
    printf("sizeof(StA)   = %d byte¥n¥n", sizeof(struct StA)); // StAとStBは同じ大きさ

    printf("sizeof(b.a)    = %d byte¥n", sizeof(b.a));
    printf("sizeof(b.ch)   = %d byte¥n", sizeof(b.ch));
    printf("sizeof(b.a)+sizeof(b.ch)   = %d byte¥n", sizeof(b.a)+sizeof(b.ch));
    printf("sizeof(StB)   = %d byte¥n", sizeof(struct StB)); // StAとStBは同じ大きさ
    return 0;
}
```

変数型の“別名”の作成 - typedef -

●typedef

```
// Example 06-05
#include <stdio.h>
typedef unsigned int UINT;

int main(void){
    UINT i = 100;
    printf("%d", i);

    return 0;
}
```

https://paiza.io/projects/xH_7gmaQ13nh8UFT1O_sfw

⬆ unsigned int の代わりに uint を使うことができる。

●構造体を使う際、タグ名の省略とtypedefを組合せる場合がよくある

とても

```
// Example 06-06
#include <stdio.h>
typedef struct {
   char name[20];
   char age;
   double weight; // kg
   double height; // cm
} Person;
// 青字部分全体の別名としての Person

int main(void){
   Person a = {"Hanako", 18, 55, 160};
   …
```

https://paiza.io/projects/Ia6hmTFwP48HuU8ltwJx2Q

⬆ struct を毎回書くなくても良くなる

```
typedef struct tagName {
    char name[20];
    char age;
    double weight;  // kg
    double height;  // cm
} Person;
```

⬆のようにタグ名を省略せずに書いてもOK。
※上記の場合は
「struct tagName a;」
「Person a;」
の両方が使えるようになるが通常、あまり意味がない。

課題 No.06 (※No.05は欠番)

課題 No.06-A

●次スライドの?????部分を埋め、以下に示す仕様のプログラムを完成しなさい。

➢以下のメンバ変数を持つ構造体 SEISEKI を定義。

- int型　　id　　···id
- char型配列　　name[20]　　···名前
- int型　　kokugo, sansu, rika　　···国語、算数、理科の点数

➢標準入力から、最大100人分のデータを入力する。

- id, name, kokugo, sansu, rika の順で入力
- idが負の値だったら入力を終了する

➢以下を標準出力に出力する。(※平均点は小数点以下切り捨て。)

id: 名前 個人ごとの平均点

...

kokugo = 国語の平均点
sansu = 算数の平均点
rika = 理科の平均点

https://paiza.io/projects/vA3nBMmQ_hdEOZaPAIHp5g

```
// No.06-A
#include <stdio.h>
#define N 100

typedef ?????? {
   ????????
   …
   ????????
} SEISEKI;


int main(void){
    ????????? seiseki[N];
    for (int i=0; i < N; i++) {
        scanf("%d", ???????????.id);
        if (???????????.id ?? ???) //入力の終了条件
            break;
        scanf("%s", ???????????.name);
        scanf("%d", ???????????.kokugo);
        scanf("%d", ???????????.sansu);
        scanf("%d", ???????????.rika);
    }
    int n = 0;
    int kokugo = 0;
    int sansu = 0;
    int rika = 0;
    for (int i=0; seiseki[i].id ?? ??? &&
i ?? ?????; i++){ // 出力の終了条件(2条件のANDとして設定)
        printf("%d: %s %d¥n"
            , ?????????.id
            , ?????????.name
            , (?????????.kokugo
             + ?????????.sansu
             + ?????????.rika) / 3);
        n++;
        kokugo += ?????????;
        sansu += ?????????;
        rika += ?????????;
    }
    printf("kokugo = %d¥n", kokugo / n);
    printf("sansu = %d¥n", sansu / n);
    printf("rika = %d¥n", rika / n);

    return 0;
}
```