Moodleの
出席確認を
提出しておいて
下さい。

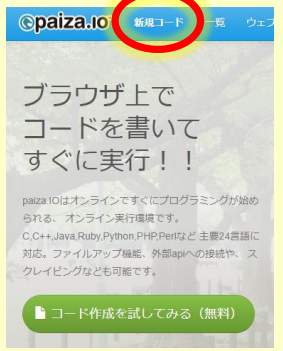
各自のC言語開発環境の起動、あるいは、

Web上のC言語実行環境 paiza.io を開いておいて下さい!

https://paiza.io/ja/projects/new?language=c

または、「https://paiza.io」にアクセス

➡ 新規コード ➡ C言語を選択

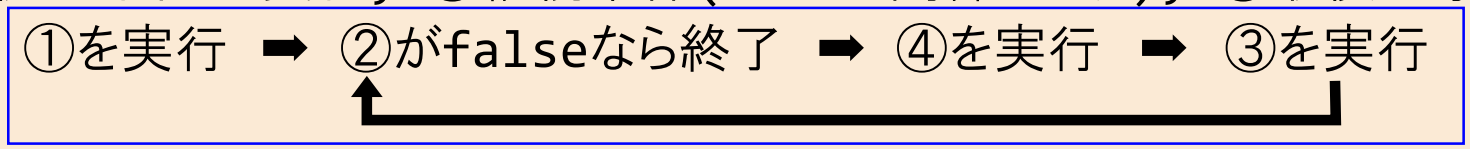


画像処理 (4J)

第05回

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()` (`stdlib.h`), `time()` (`time.h`) ･･･実行の度に異なる乱数列の発生、任意の範囲の乱数への変換方法

第4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。

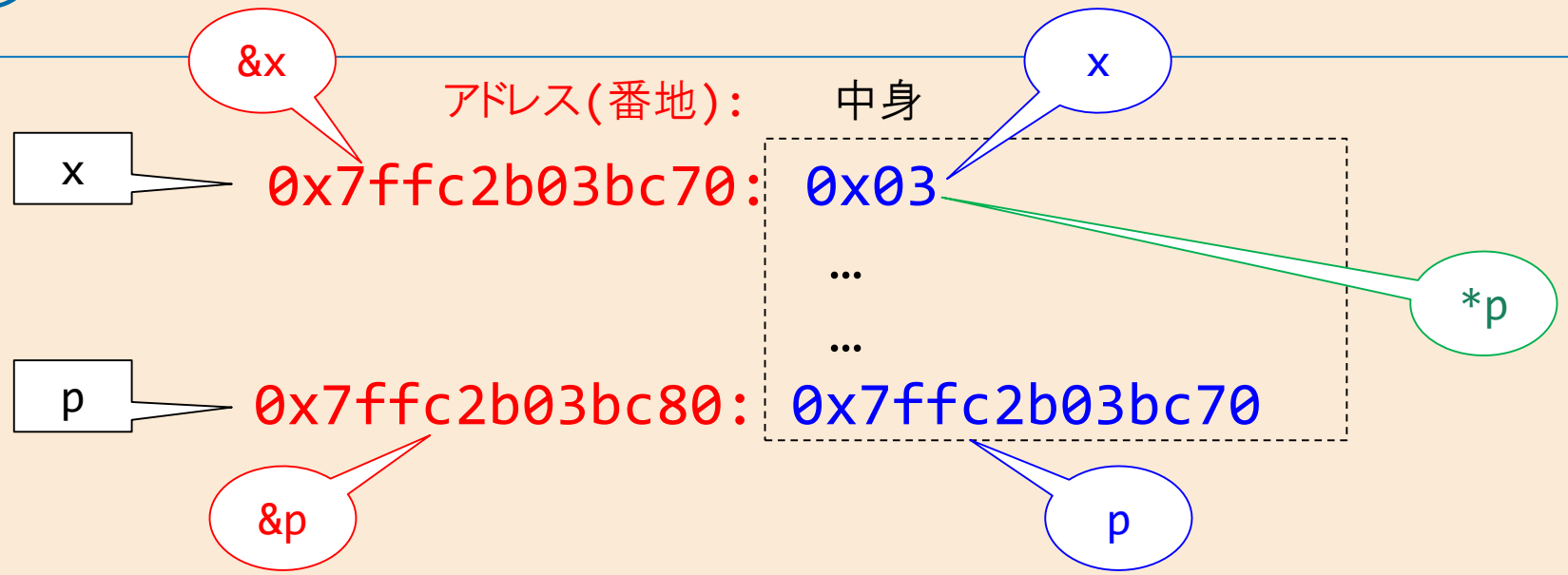
```
void func(int *x, int *y);
```

- 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。

第4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4回のまとめ③


```
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
中身	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。)

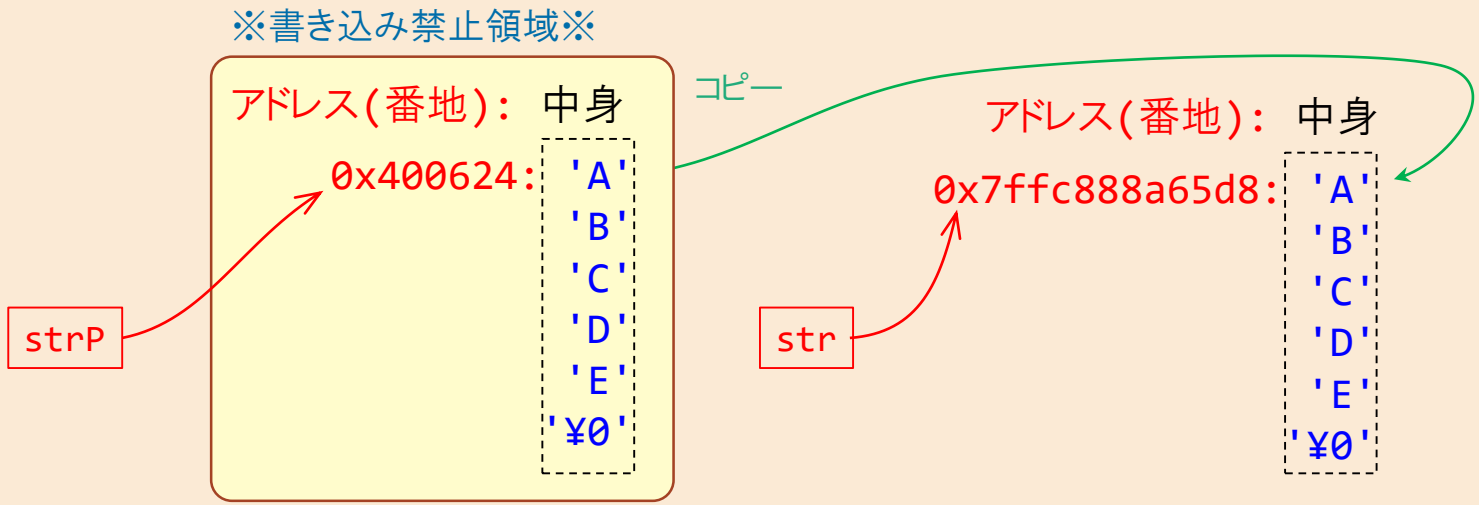
第4回のまとめ④

●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



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 05-01
#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… };**

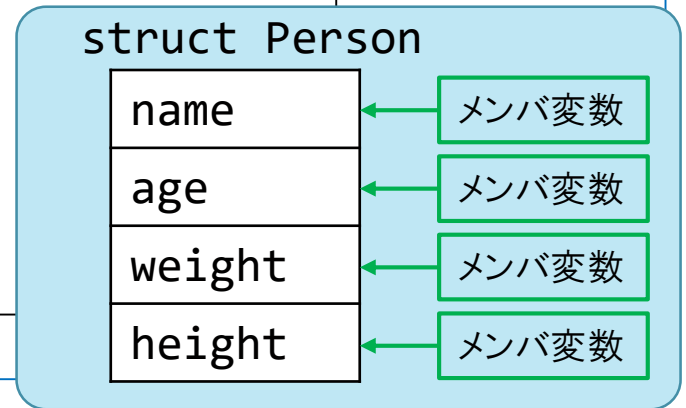
これが新しい「型名」のような扱いになる

●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/Im_7hFsCVjP0UAxMa2QNew

```
    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 05-01 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 05-01
#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

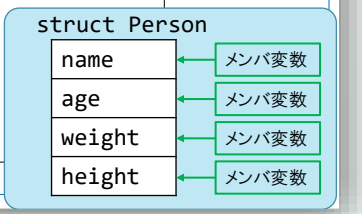
➢宣言：**struct 構造体タグ名** 変数名;

➢メンバ変数へのアクセス：

変数名.メンバ変数名

https://paiza.io/projects/Im_7hFsCVjP0UAxMa2QNew

```
    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/7d_35zHHUDPa6mKA7NVz6A

```
// Example 05-02
#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)

BMI=22 の標準体重が、最も病気になりにくいとされています。
BMI>25 の場合、生活習慣病のリスクが2倍以上になるとのこと。
また、極端な低体重も、(もともとの体質等を除いて)健康上のリスク高くなります。
近年では BMI<18 のモデルは痩せすぎとして
欧州ではショーに出られない等の規制もあるほどです。
(つまりここで挙げた「モデル体重」だとモデル業が出来ない・・・)

1. Example 05-02 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/7d_35zHHUDPa6mKA7NVz6A

```
// Example 05-02
#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 05-03
#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/5k-FvIH6y_yzJ3qjdCUuAQ

【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 05-03 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)

●他の型と同様、**配列**や**ポインタ**も使える。

```
// Example 05-03
#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/5k-FvIH6y_yzJ3qjdCUuAQ

【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。
```

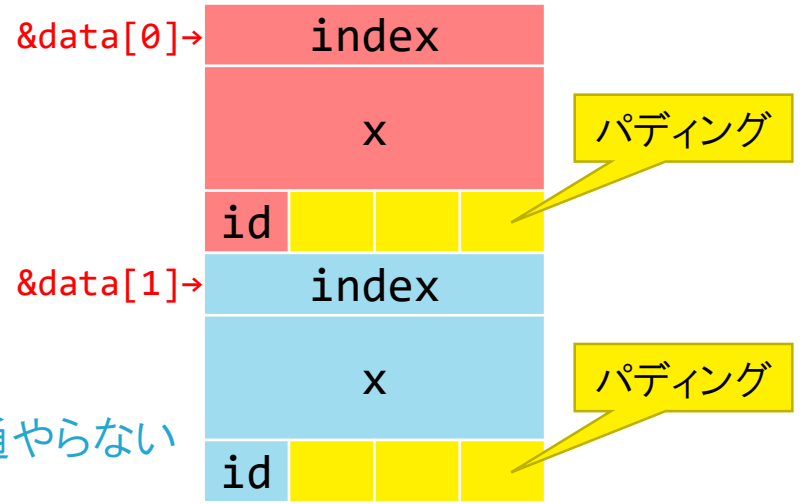
●パディングの話

➢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/Q1iNF7C3RHp4EgWPA7e_1w

```
// Example 05-04
#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 05-05
#include <stdio.h>
typedef unsigned int UINT;

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

    return 0;
}
```

https://paiza.io/projects/DGK0m2_h-5KCe105wK1grg

⬆ 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 05-06
#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/B7tKfKrEVaDrxgsOxAof5g

⬆ 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。

動的配列

動的配列とは･･･？

●通常の配列(静的配列)は、コンパイル時にサイズが確定している必要がある。

- 例えば、　`int data[1000];`　と宣言したのに、実際は　10個 しか入力がないとか
- 逆に、　`int data[100];`　と宣言したら、データ 200個 入力があったりとか

```
int n;
scanf("%d", &n);
int data[n];
```

- みたいに出来たら良いよね？　~~➡ 残念ながら、出来ません。~~

Variable Length Array

➡ C99 では、できるようになってました。 ･･･“可変長配列(VLA)”

➡ ですが、(より新しい)C11ではオプション扱いになってるらしい。

- ぐぬぬ･･･。malloc() でのメモリ領域確保については、「動的配列の実現のため」が、分かりやすいモチベーションになるのに･･･。

  - 可変長配列は“スタック領域”に確保されるので、関数が終わると消えます。
    また、一度宣言した可変長配列は、あとから長さを変更することは出来ません。
    ⇔ malloc()では“ヒープ領域”に確保されるので、明示的に free() で開放するまで使えます。
    また、realloc()で後から長さを変更することもできます。

(気を取り直して、)まずは動的配列から

#include <stdlib.h>　が必要です。

●大きさを指定して、メモリ領域を確保する

```
void *malloc(size_t size);
```

size で指定した byte 数分のメモリ領域を確保し、
確保した領域の先頭のアドレスを返す。
(※確保に失敗した場合は NULL が返る。)

size [byte]
の領域
‼ 中身は不定 ‼

どっか(連続して)
空いている
領域を探してきて
割り当てる
(ヒープ領域内)

●(malloc()等で確保した)メモリ領域を解放する

```
void free(void *ptr);
```

ptr には、確保した際のmalloc()の返り値を指定する。
確保した領域は、使わなくなったら必ず解放する!
解放するまでは、malloc()の返り値を保持しておく必要がある。

●malloc() と free() は必ずセットで使う!! ...メモリリークの主要な原因のひとつ(free()忘れ)
また、free()後に、ptrの値を誤って使用しないように、すぐに NULL を代入する習慣を。

動的配列としての典型的な使い方

●配列として利用するには、キャストと、確保する領域の大きさの計算が必要。

```
n = 100;    // 要素数
int *arri = (int *)malloc(sizeof(int) * n);
double *arrd = (double *)malloc(sizeof(double) * n);
```

- sizeof(型名)　　･･･ 要素1つ分の byteサイズ を求めて、
- sizeof(型名) * n　　･･･ n要素分の byteサイズ を計算して領域確保。
- malloc()の返り値は (void *)型なので、(型名 *) のポインタにキャスト
- それを、 ポインタ変数として受け取る。

各要素へは、 arri[0], arri[1], arri[2], …
または *(arri+0), *(arri+1), *(arri+2), … としてアクセス

使い終わったら速やかに **free(arri);**
arri = NULL;　　// 安全のためNULLを代入

malloc()による動的配列

https://paiza.io/projects/RHce0P6Jl9iqf-vOvRm8Zg

```
// Example 05-07: 動的配列
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int n;
    scanf("%d", &n);  // 配列の要素数を入力

    // intのサイズ×要素数分の領域を確保して、intポインタ型にキャストして返す
    int *arr = (int *)malloc(sizeof(int) * n);

    for(int i=0; i < n; i++)
        arr[i] = i * i; // *(arr+i) = i * i; でも同じこと

    for(int i=0; i < n; i++)
        printf("arr[%d] = %d¥n", i, arr[i]);

    free(arr);     // 実際はプログラムが終了すると解放されるので不要だが、習慣として。
    arr = NULL;    // (さらにNULLを代入しておくと良い)

    return 0;
}
```

演習(4)

1. Example 05-07 を実行して、丁寧に動作を追ってみる
2. int型ではなく、double型の要素の配列を扱うように修正してみる

malloc()による動的配列

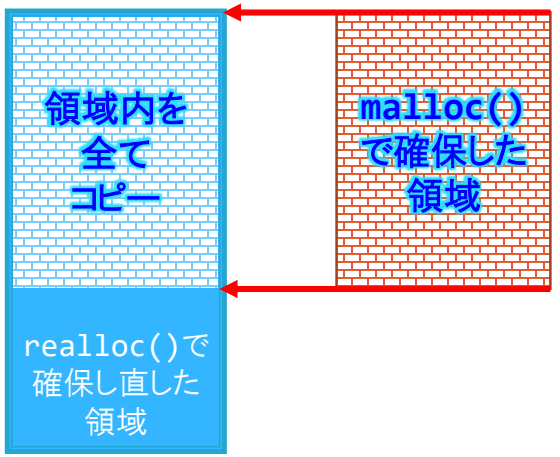
https://paiza.io/projects/RHce0P6Jl9iqf-vOvRm8Zg

```
// Example 05-07: 動的配列
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int n;
    scanf("%d", &n);  // 配列の要素数を入力

    // intのサイズ×要素数分の領域を確保して、intポインタ型にキャストして返す
    int *arr = (int *)malloc(sizeof(int) * n);

    for(int i=0; i < n; i++)
        arr[i] = i * i; // *(arr+i) = i * i; でも同じこと

    for(int i=0; i < n; i++)
        printf("arr[%d] = %d¥n", i, arr[i]);

    free(arr);     // 実際はプログラムが終了すると解放されるので不要だが、習慣として。
    arr = NULL;    // (さらにNULLを代入しておくと良い)

    return 0;
}
```

動的配列の大きさを変更する、0初期化

- `void *realloc(void *ptr, size_t size);`

  ptrには malloc()の返り値を入れる。
  (※ptr が NULL の場合は malloc() と全く同じ動作になる)
  size には、変更後の大きさを指定する。
  返り値の使い方は、 malloc() と同じ。



※realloc()では、改めて領域を確保し、これまでの領域の内容を
全てコピーする動作が入るので、あまり頻発するのは良くない。

- `void *calloc(size_t n, size_t size);`

  malloc()との違いは、領域全体が 0 で初期化されていること。
  また、引数が2つあり、 n には要素数、size には1要素の大きさを指定する。

利用例:

```
double *d = (double *)calloc(n, sizeof(double));
```

realloc() による領域の大きさの変更

https://paiza.io/projects/MIjukvD1iu9ynAMcBBc2mg

```
// Example 05-08: 動的配列2
#include <stdio.h>
#include <stdlib.h>
int main(void){
    int n;
    scanf("%d", &n);  // 配列の要素数を入力

    // intのサイズ×要素数分の領域を確保して、
    // intポインタ型にキャストして返す
    int *arr = (int *)malloc(sizeof(int) * n);

    for(int i=0; i < n; i++)
        arr[i] = i * i; // *(arr+i) = i * i; でも同じこと

    for(int i=0; i < n; i++)
        printf("arr[%d] = %d¥n", i, arr[i]);

    printf("¥n¥nrealloc()¥n");
    arr = (int *)realloc(arr, sizeof(int) * n * 3); // 3倍に増やす

    for(int i=n; i < n*3; i++)
        arr[i] = -i * i; // 増やした部分にはマイナスの値を

    for(int i=0; i < n*3; i++)
        printf("arr[%d] = %d¥n", i, arr[i]);

    free(arr);
    arr = NULL;

    return 0;
}
```

その他、動的配列に関して

●静的配列のように sizeof() 等で
あとから全体のサイズを取得することは出来ないので注意

- 静的配列の場合は有効だった、

```
n = sizeof(data)/sizeof(data[0]);
```

のようなことが出来ない。

●なお、“必ず free() しなければいけない”、というのを解消(不要に)したのが、
Garbage Collection という仕組み。(JavaやC#に実装)

- 使われなくなった領域を自動的に開放してくれる
- プログラマが詳細にメモリ管理を意識しなくても良くなる
  ・・・ 解放忘れによるメモリリークを防ぐ
- ただ、結構重い処理だったりする

今回のまとめ

●構造体

- 1つ以上の変数(メンバ変数と呼ぶ)をまとめたもの
  - 右の例でいうと、 a.name とか a.age のように使う
  - ポインタ変数なら a->name とか a->age とする
  - a->age は (*a).age と等価
- 「struct 構造体タグ名」が型名となるが、通常は typedef を使って struct を省略。

```
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};
   …
```

●動的配列

- 典型的な使い方 `double *arrd = (double *)malloc(sizeof(double) * n);`
- 上記だと arrd[0], arrd[1],… または *(arrd+0), *(arrd+1), … としてアクセス
- 使い終わったら必ず `free(arrd);`
  free() 後のポインタは arrd = NULL; 等で、誤って使用しないよう対策する習慣を。

課題 No.05

課題 No.05-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/LOdRH2KSuX6NjFPtjUYtkQ

```
// No.05-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;
}
```

課題 No.05-B

●以下の関数を作成し、main()関数とともに提出しなさい。【main()内は変更しないこと!】

➢ **int* intArr(int n);** ･･･ n個のint型の動的配列を malloc() で作り、各要素に 0～n-1 の値(添字と同じ値)を設定し、その先頭のアドレス返す関数。

```
// No.05-B
#include <stdio.h>
#include <stdlib.h>

int* intArr(int n);

int main(void){
    int n;
    scanf("%d", &n);
    int *arr = intArr(n);
    for(int i = 0; i <n; i++)
        printf("%d¥n", arr[i]);
    free(arr);

    return 0;
}

// ここに intArr() 関数を書く
```

https://paiza.io/projects/H2mRnIImcONxz8r8yLHZHQ