

# Zoom URL

- **第6回講義は下記URLで実施します。**
- **5/27（木） 13:30 までにアクセスしてください。**  
**（10分程前からアクセス可能になります）**

<https://us02web.zoom.us/j/89923692776?pwd=YVFhUFZ3bExhVWVNMcm93TUVXbFF0QT09>

ミーティングID: 899 2369 2776

パスワード: 550262

# コンピュータサイエンスとプログラミング

## 第6回演習 2021年05月27日

構造体(2) (structure)  
動的メモリ割り付け

**オブジェクトを思うままに定義できるようになる！**  
**(先週の続き)**

# 構造体のtypedef宣言

```
typedef struct 構造体タグ 新しい型名;
```

- 構造体型pointに対応するPOINTという型の宣言

```
struct point{  
    int x; /* x座標 */  
    int y; /* y座標 */  
};
```

```
typedef struct point POINT;
```

```
typedef struct point{  
    int x; /* x座標 */  
    int y; /* y座標 */  
}POINT;
```

構造体型の宣言と  
typedef宣言を  
まとめた書き方

- POINT型の変数p1の定義

```
POINT p1;
```

変数p1はPOINT型の変数と呼ばれる

# 構造体の引数

- 例題5-1\*の原点から点p1までの距離を計算する処理をdistance関数として定義

- 構造体型pointを引数に指定してdistance関数を宣言

```
double distance(struct point p1);
```

いちいち構造体型pointを使うと関数の宣言が長くなる

- typedef名のPOINT型を使ってdistance関数を宣言

```
double distance(POINT p1);
```

- distance関数の定義

```
double distance(POINT p1){  
    return sqrt(p1.x * p1.x + p1.y * p1.y);  
}
```

- distance関数の呼び出し

```
POINT pt1 = {2, 4};  
double d;
```

```
d = distance(pt1);
```

実引数にPOINT型の構造体を指定する

# 構造体を返す関数 (例)

- 点p1を(dx, dy)だけ移動した点を返すmovePoint関数の宣言
  - 返却値の型にPOINT型を指定

```
POINT movepoint(POINT p1, int dx, int dy);
```

- movePoint関数の定義

```
POINT movePoint(POINT p1, int dx, int dy){  
    POINT p2;  
  
    p2.x = p1.x + dx;  
    p2.y = p1.y + dy;  
    return p2;  
}
```

- 引数と同じように、関数の返却値にも構造体が使える
- 点p1を(dx, dy)移動した点を返す関数
- POINT型の局所変数p2を定義
- 移動後の点をp2に格納

- 関数の呼び出し

```
POINT pt1 = {3, 7};  
POINT pt2;  
  
pt2 = movePoint(pt1, 2, 5);
```

- POINT型の変数pt2をさらに定義
- movePoint関数の返却値を変数pt2にコピー

# 【例題6-1】

## 構造体を返す関数のプログラム例 (1/2)

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

```
typedef struct point{  
    int x;    /* x座標 */  
    int y;    /* y座標 */  
} POINT;
```

```
POINT movePoint(POINT p1, int dx, int dy){  
    POINT p2;  
  
    p2.x = p1.x + dx;  
    p2.y = p1.y + dy;  
    return p2;  
}
```

- 入力された点の座標を移動するプログラム

次ページに続く

# 【例題6-1】

## 構造体を返す関数のプログラム例 (2/2)

```
int main(void) {
    POINT pt1;          /* 移動前の座標 */
    POINT pt2;          /* 移動後の座標 */
    int dx, dy;

    printf("> (x1, y1) ");
    scanf("%d %d", &pt1.x, &pt1.y); /* pt1を入力 */

    printf("> (dx, dy) ");
    while(scanf("%d %d", &dx, &dy) != EOF) {
        pt2 = movePoint(pt1, dx, dy);
        printf("Original: (%d,%d), Shift: (%d,%d), Result: (%d,%d)¥n",
            pt1.x, pt1.y, dx, dy, pt2.x, pt2.y);

        pt1 = pt2;          /* 構造体pt2をpt1にコピー */
        printf("> (dx, dy) ");
    }
    return 0;
}
```

# 【課題6-1】

- POINT型（例題6-1参照）の2つの引数の中点を返すgetMiddlePoint関数を次のように定義し、入力した2点の中点を出力するプログラムを作成せよ。

ただし、下のような実行例を想定している。

```
POINT getMiddlePoint(POINT p1, POINT p2);
```

- 実行例

```
% ./a.out  
> (x1, y1) 1 2  
> (x2, y2) 5 8  
Midpoint between (1, 2) and (5, 8) is (3, 5).
```



# 値渡しと参照渡し

- 値渡し
  - 変数の**値をコピーして渡す**
- 参照渡し
  - 変数の値が格納されている**アドレスをコピーして渡す**
- 参照渡しは配列や構造体の先頭アドレスを渡すだけで良いので、値渡しに比べて格段に処理が速く、メモリ消費も少ない

# 構造体へのポインタ引数 (1/3)

- 「ポインタ」で学んだように，関数が呼び出された時，実引数の値が**仮引数にコピー**されてから関数が実行される（**値渡し**）。
- **引数が構造体の場合も同様.**
- **関数内で仮引数の構造体のメンバの値を変更しても，実引数のメンバの値は変化しない.**

# 構造体へのポインタ引数 (1/3)

(例)

```
void moveVpoint(POINT p1, int dx, int dy){  
    p1.x = p1.x + dx;  
    p1.y = p1.y + dy;  
}
```



コピーが作られている

```
int main(){  
    POINT p1 = {1,1};  
    moveVpoint(p1, 3, 3);  
}
```



関数内で仮引数p1のメンバの値を変更しているが、  
実引数のメンバの値は変化しない。

# 構造体へのポインタ引数 (2/3)

- 実引数の構造体のメンバの値を関数で変更したい時は、構造体へのポインタを引数に指定

```
void movePpoint(POINT *p1, int dx, int dy);
```

- 構造体へのポインタを使用した時、そのメンバを参照するには、メンバ参照演算子「.」ではなく、**間接メンバ演算子（アロー演算子）「->」**を用いる。

構造体へのポインタ->メンバ名

(例) 構造体へのポインタを使ってmoveVpointを書き換えると

```
void movePpoint(POINT *p1, int dx, int dy){  
    p1->x = p1->x + dx;  
    p1->y = p1->y + dy;  
}
```

# 構造体へのポインタ引数 (3/3)

- 構造体のアドレスを実引数に指定して、次のように movePpoint関数を呼ぶと、構造体pt1の値は (5, 12) に変更される.

```
POINT pt1 = {3, 7};  
  
movePPoint(&pt1, 2, 5);
```

- 変数p1が構造体へのポインタの時、間接演算子「\*」を使うと、\*p1で構造体自身を参照できるので、メンバxは、(\*p1).xでも参照できる.
- しかし、括弧を使ったりして複雑なので、一般には、p1->xと書く.

メンバ参照演算子「.»の優先順位は、間接参照演算子「\*」より高いため、\*p1.xは\*(p1.x)の意味.

## 【例題 6-2】

### 構造体へのポインタ引数をもつ関数の例 (1/2)

```
#include <stdio.h>

typedef struct point{
    int x;    /* x座標 */
    int y;    /* y座標 */
} POINT;

POINT movePPoint(POINT *p1, int dx, int dy){
    p1->x = p1->x + dx;
    p1->y = p1->y + dy;
}
```

次ページに続く

## 【例題6-2】

## 構造体へのポインタ引数をもつ関数の例 (2/2)

```
int main(void) {
    POINT pt1;          /* 移動後の座標 */
    POINT pt2;          /* 移動前の座標 */
    int dx, dy;

    printf("> (x1, y1) ");
    scanf("%d %d", &pt1.x, &pt1.y); /* pt1を入力 */

    printf("> (dx, dy) ");
    while(scanf("%d %d", &dx, &dy) != EOF) {
        pt2 = pt1;

        movePPoint(&pt1, dx, dy);

        printf("Original: (%d, %d), Shift: (%d, %d), Result: (%d, %d).¥n",
               pt2.x, pt2.y, dx, dy, pt1.x, pt1.y);

        printf("> (dx, dy) ");
    }
    return 0;
}
```

## 【課題6-2】

- 原点を中心として，点p1の位置を時計の針と逆方向にdt度回転するrotatePoint関数を次のように定義し，入力した点を順に回転させるプログラムを作成せよ。  
ただし，回転角度dtはラジアンではなく度とし，右のような実行例を想定するものとする。

```
void rotatePoint(POINT* p1, int dt);
```

```
% ./a.out  
> (x1, y1) 10 10  
> (dt) 45  
(0, 14)  
> (dt) 45  
(-10, 10)  
> (dt) 180  
(10, -10)  
> (dt) -30  
(4, -14)  
> (dt) -60  
(-10, -10)
```



# 構造体へのポインタを返す関数

- 例題6-1のmovePointを変更して，構造体へのポインタを返すようにmoveMPoint を定義する

```
POINT* moveMPoint(POINT p1, int dx, int dy){  
    POINT* p2;  
  
    p2.x = p1.x + dx;  
    p2.y = p1.y + dy;  
  
    return &p2;  
}
```

# 構造体へのポインタを返す関数


- 例題6-1のmovePointを変更して，構造体へのポインタを返すようにmoveMPoint を定義する

```
POINT* moveMPoint(POINT p1, int dx, int dy){  
    POINT* p2;  
  
    p2.x = p1.x + dx;  
    p2.y = p1.y + dy;  
  
    return &p2;  
}
```



# 構造体へのポインタを返す関数

- 例題6-1のmovePointを変更して，構造体へのポインタを返すようにmoveMPoint を定義する



```
POINT* moveMPoint(POINT p1, int dx, int dy){  
    POINT* p2;  
  
    p2.x = p1.x + dx;  
    p2.y = p1.y + dy;  
  
    return &p2;  
}
```

局所変数の有効範囲は  
この関数内

局所変数のアドレスを返している。  
局所変数のメモリ領域は関数を出るとき  
に開放される  
→他の変数で上書きされうる

# 動的メモリ割り付け

Dynamic memory allocation

- 任意の大きさのメモリ領域を，OSが管理するメモリ（**ヒープ**と呼ばれる）から確保する機能
- 次の2つの関数をペアで用いる（stdlib.hをインクルード）

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

- 大きさsizeバイトのメモリ領域を確保し，その先頭アドレスを返す。
- メモリ領域を確保できなかった時はNULLを返す。

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

- ポインタptrが指すメモリ領域を解放し，ヒープへ戻す。
- malloc関数の引数で指定するバイト数は，sizeof演算子を使って計算

```
size_t dsize = sizeof(double);
```

size\_t型は非負の整数を表す型 (unsigned int)

# 動的メモリ割り付け

- 動的メモリ割り付けで確保したメモリ領域は、その領域が開放される（freeされる）まで有効
- メモリ領域有効期間を**プログラムで制御できる**  
→関数やブロックを超えることが可能

# 動的メモリ割り付けの例

- **int型のデータに必要なメモリ領域の確保**

```
int* p1;  
  
p1 = malloc(sizeof(int));  
if (p1 == NULL) {  
    fprintf(stderr, "Memory Shortage¥n");  
    exit(1);  
}  
*p1 = 0;          /* p1が指すメモリ領域を初期化 */
```

- **int型のデータに必要なメモリ領域をmalloc関数を使ってヒープから確保**
- **その先頭アドレスをポインタp1に代入**
- **int型以外のメモリ領域を使いたい時は、赤文字の部分を書き換える**

# 【例題6-3】 動的メモリ割り付けの例(1)

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

```
int main(void){
    int x = 19;
    int* p1;
    int* p2;
```

```
p1 = &x;
```

```
p2 = malloc(sizeof(int));
if (p2 == NULL){
    fprintf(stderr, "Memory Shortage\n");
    exit(1);
}
```

```
*p2 = 77;
```

```
printf("HENSU ¥t VALUE ¥t INDIRECT ¥n");
printf("p1 ¥t %10p ¥t %d¥n", p1, *p1);
printf("p2 ¥t %10p ¥t %d¥n", p2, *p2);
free(p2);
return 0;
}
```

- 各値と間接参照される値を出力する
  - 局所変数を指すポインタ (p1)
  - malloc関数によって確保したメモリ領域を指すポインタ (p2)

- p1には局所変数xのアドレスを代入

- p2にはmalloc関数で確保したメモリ領域を初期化

## 【例題6-4】 動的メモリ割り付けの例(2)

任意の個数のコマンドライン引数の平均を計算

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

int main(int argc, char *argv[]){
    int    i, num;
    double sum = 0.0;
    double* table;

    if (argc < 2){
        fprintf(stderr, "Arguments are not specified.¥n");
        exit(1);
    }

    num = argc - 1;
    table = malloc(sizeof(double) * num);
    if (table == NULL){
        fprintf(stderr, "Memory Shortage.¥n");
        exit(1);
    }
}
```

次ページに続く



## 【例題6-4】 動的メモリ割り付けの例(2)

```
for (i = 0; i < num; i++){  
    *(table + i) = atof(argv[i + 1]);  
}
```

```
for (i = 0; i < num; i++){  
    sum += table[i];  
}
```

```
printf("Average of %d doubles is %.2f.¥n", num, sum / num);  
free(table);  
return 0;
```

```
}
```

i+1番目のコマンドライン  
引数を実数に変換し、  
tableが指す配列のi番目  
の要素に格納

- 変数tableは配列の先頭要素を指すためのポインタ
- コマンドライン引数で指定した実数の個数をnumに代入
- malloc関数を使って確保した配列用のメモリ領域の先頭アドレスをポインタtableに格納
- ポインタtableが指す配列のi番目の要素は、\*(table + i) またはtable[i]で参照できる。

# 動的メモリ割り付け メリット&デメリット

## 【メリット】

- メモリ確保サイズをプログラム実行時に決められる。
- 初期化せずに変数を宣言できる。

## 【デメリット】

- 非常にデバッグしにくいバグを生む  
(freeのし忘れ)
- プログラム開発にかかる時間の大部分がメモリ関連のデバッグに充てられるとも言われる

**利用には注意が必要！**

# Appendix コマンドライン引数の利用

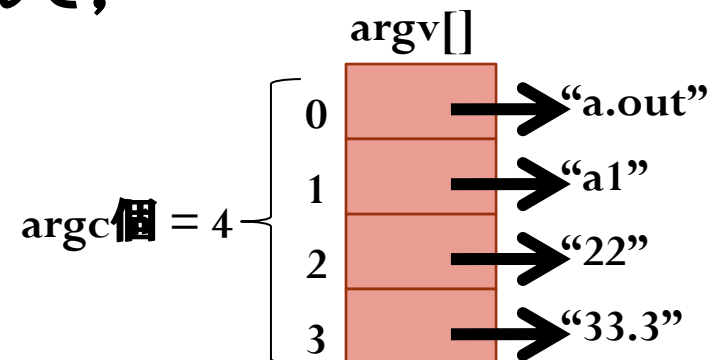
- データを実行時に指定できるようにする

```
% ./a.out a1 22 33.3
```

- main関数を次のように定義する

```
int main(int argc, char* argv[]);
```

- argc個のプログラム引数が  
argvが示す配列に格納されて、  
main関数が呼ばれる。



## 【例題 A1】

## コマンドライン引数を出力する例

```
#include <stdio.h>

int main(int argc, char* argv[]){
    int i;

    printf("argc = %d\n", argc);
    for (i=0; i < argc; i++)
        printf("argv[%d] %s\n", i, argv[i]);
    return 0;
}
```

実行例

```
% ./a.out a1 22 33.3
argc = 4
argv[0] ./a.out
argv[1] a1
argv[2] 22
argv[3] 33.3
```

printf関数

- 書式 %sは文字列を出力するため
- %s は二重引用符自身を出力するため

# コマンドライン引数を数値として利用

```
int atoi(const char* str);
```

- 文字列strをint型の数値に変換した値を返す

```
double atof(const char* str);
```

- 文字列strをdouble型の数値に変換した値を返す
- 注意
  - 関数や関数を使う時は、stdlib.hをインクルードする
  - キーワードconstは引数の値に関数内で変更しないことを示す

## 【例題 A2】

## コマンドライン引数を数値として利用する例

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

int main(int argc, char *argv[]){
    int x;
    double y;

    x = atoi(argv[2]);          /* int型に変換*/
    y = atof(argv[3]);          /* double型に変換*/

    printf("argv[2] %d\n", x);
    printf("argv[3] %f\n", y);

    return 0;
}
```

実行例

```
% ./a.out a1 22 33.3
argv[2] 22
argv[3] 33.300000
```

# 構造体へのポインタを返す関数

- 例題6-1のmovePointを変更して，構造体へのポインタを返すようにmoveMPoint を定義する
- malloc関数で確保したメモリ領域のアドレスをポインタnewPに代入
- 移動した点の座標をnewが指す構造体へ格納
- return文でnewPを返す

```
POINT* moveMPoint(POINT p1, int dx, int dy){
    POINT *newP;

    newP = malloc(sizeof(POINT));
    if (newP == NULL){
        fprintf(stderr, "Memory Shortage¥n");
        exit(1);
    }
    newP->x = p1.x + dx;
    newP->y = p1.y + dy;
    return newP;
}
```

## 【例題6-5】

例題6-1のプログラムを，moveMPoint関数を使うように変更する

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

typedef struct point{
    int x;    /* x座標 */
    int y;    /* y座標 */
} POINT;
```

moveMPoint の定義（前ページのとおり）

次ページに続く



## 【例題6-5】

```
int main(void) {
    POINT pt1;          /* 移動前の座標 */
    POINT *pt2;          /* 移動後の座標 */
    int dx, dy;

    printf("> (x1, y1) ");
    scanf("%d %d", &pt1.x, &pt1.y);    /* pt1を入力 */

    printf("> (dx, dy) ");
    while(scanf("%d %d", &dx, &dy) != EOF) {
        pt2 = moveMPoint(pt1, dx, dy);
        printf("Original: (%d,%d), Shift: (%d,%d), Result: (%d,%d) .Yn",
            pt1.x, pt1.y, dx, dy, pt2->x, pt2->y);

        pt1 = *pt2;
        free(pt2);
        printf("> (dx, dy) ");
    }
    return 0;
}
```

pt2を構造体へのポインタとして宣言

構造体pt1を(dx, dy)移動した点を生成し、その先頭アドレスをpt2に代入する

## 【課題6-3】

- 例題6-4のプログラムで、`malloc`関数で配列用のメモリ領域を確保し、コマンドライン引数の値を格納する処理を次の関数として定義し、プログラムを作成・実行して動作を確認せよ。ただし、`makeArray`関数は`int`型の配列の先頭を返すものとし、コマンドライン引数では整数を指定するように、`int`型の配列に変更する。

```
int* makeArray(int argc, char* argv[]);
```

なお、次のような実行例を想定する。

```
% ./a.out 13 22 36 41 55
Average of 5 doubles is 33.40.
```

# レポートに関して

- 課題6-1, 6-2, 6-3を実施しレポート課題として提出すること.
  - 提出期限：2021年06月03日 09:00
  - CLEで提出する際のファイル名（半角英数）は以下の通りとする.
    - 課題6-1の場合, XXXX-kadai6-1.c
    - 課題6-2の場合, XXXX-kadai6-2.c
    - 課題6-3の場合, XXXX-kadai6-3.c
- XXXXの部分は学籍番号下4桁である.

# 質問について

## ◇質問用メールアドレス

[csp-query@pn.comm.eng.osaka-u.ac.jp](mailto:csp-query@pn.comm.eng.osaka-u.ac.jp)

## ◇TAへの質問について:

- まず自分で調べること. エラーメッセージを検索すると情報が得られることが多い.
- 質問を明確にすること.
  - 単に「わかりません…」「できません…」「おかしいです…」という質問には答えません.
  - できるだけ具体的に状況を書くこと.
  - エラーメッセージを記載すると回答のヒントになる.
- プログラムリストの代筆はしない.
- バグ取りはしない.