コンピュータとプログラミング C言語編

4. 構造体、ポインタ

阿部洋丈 / ABE Hirotake
habe@cs.tsukuba.ac.jp

構造体とは

- 複数の変数（メンバー）をまとめて扱うための仕組み。
- よくある例：
  - 複素数： 実部と虚部
  - 空間上の点： x座標、y座標、z座標、…
  - 多角形： 点1、点2、点3、…
  - 名簿情報： 学籍番号、姓、名、学類、入学年、クラス、…
  - etc etc…

構造体の使い方

構造体の定義。
定義のある場所によって
定義の有効範囲が変わる

```
struct point {
  float x;
  float y;
};

struct point a;
a.x = 1.0;
a.y = 2.0;
struct point b = {3.0, 4.0};
```

「.」を使って
要素にアクセス

配列のような
初期化も可能

```
struct segment {
  struct point p1;
  struct point p2;
} s[10];

s[0].p1.x = a.x;
s[0].p1.y = a.y;
s[0].p2 = b;
```

構造体の中に
構造体

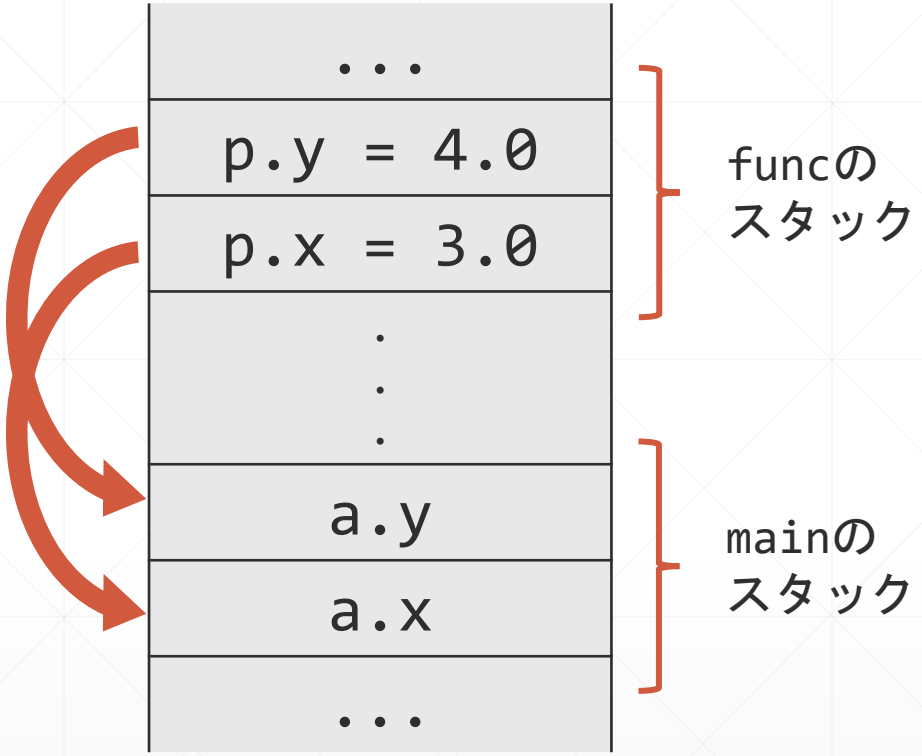
構造体を定義し、同時に
その配列を宣言

全体を
コピーする
こともできる

関数間の構造体受け渡し

- 構造体全体がコピーされる。

```
struct point { float x, y; };

struct point func(void) {
  struct point p = {3.0, 4.0};
  return p;
}

int main(int argc, char *argv[]) {
  struct point a = func();
}
```



注意：
（左の例とは逆に）構造体を関数に渡す場合、呼ばれた側で値を変更しても、呼び出し側には反映されない。

共用体

- 見た目は構造体と似ているが、メンバーのためのメモリ領域が共用化される点が異なる
  - 同じメモリ領域に異なる型としてアクセスできる

```
union {
  int i;
  double d;
} x;

x.i = 0;
x.d = 1.234567;
printf("%f¥n", x.d);    // 1.234567
printf("%d¥n", x.i);    // 1402701959 <- 無理やり int として解釈した結果
                        // (結果は実行環境によって異なる)
```

無名の union を定義し、
その方の変数を1つ宣言
（構造体でも同じことが可能）

typedef

- すでに存在する型に別名を与える機能
  - struct や union、unsigned と毎回書くのが面倒な場合
  - 特定の用途で使っていることを敢えて意識させたい場合

```
typedef struct point {   // ここに point と書かなくても良い
  float x;
  float y;
} point_t;   // 型名の末尾に "_t" と付けることが多い

typedef unsigned int size_t;   // 序数でなく基数が入ることを示す
```

ポインタとは

- アセンブラを既に学んでいる皆さん向けの説明：

  ポインタとは、アドレスを格納するための変数である。ただし、そのアドレスの指す先の型が予め決められている。

- アセンブラを学んでいない人に説明するのは結構大変
  - 教科書によれば：「ポインタは左辺値を扱うための概念であり、あるメモリ領域を扱うための手段となる。」（この説明の前に「左辺値」の説明が来る…）

ポインタの表記方法


```
int i = 0;    // ふつうの int 型変数の宣言
int *p;       // int 型領域を指すためのポインタ変数 p を宣言
              // (この表記には少し慣れが必要)

p = &i;       // i のアドレスを p に代入
*p = 1;       // p が指す先のメモリ（つまり i）に 1 を代入
printf("%d¥n", i);   // 1 と表示される

int *q, j;    // q はポインタ。j は int 変数（ポインタではない）
q = p;        // q も i を指すようになる
printf("%d¥n", *q + 1);  // 1 + 1 で 2 と表示される
```

これまでの種明かし (その１)

- scanf で引数に & を付けるのは、その変数の値ではなくアドレスを与えるため
- 第2週の課題の *((unsigned *) &val) は、暗黙の型変換を防ぐために、一旦アドレスに変換した上で unsigned に戻すという小細工
- 配列名は実はポインタ（より正確には定数）。"int a[10]" と定義すると、a の型は int * 型。なので & を付けなくて良かった

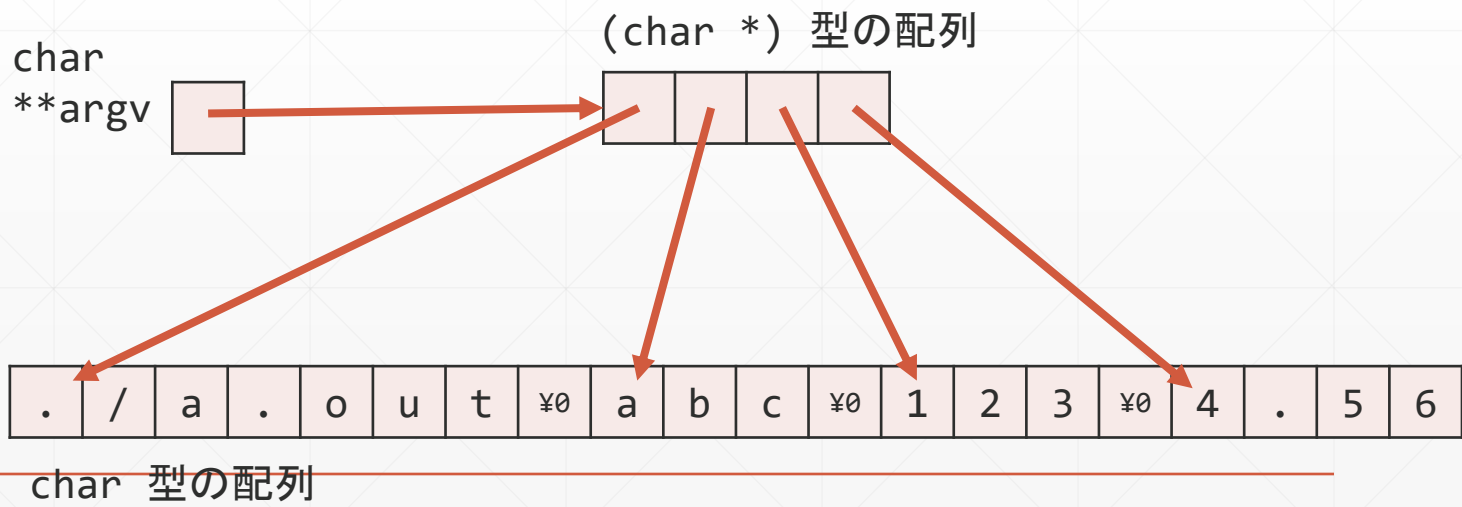
```
int a[10];
int *p;

p = a;    // 型が同じなのでエラーにならない
a = p;    // a 自体は const なのでコンパイルエラーになる
```

これまでの種明かし（その２）

- char **argv
  - 実は、2次元配列ではなく、1次元配列をポインタで区切って使っている。（本当に2次元配列だと無駄が出る（左図））
  - char ** は、char へのポインタへのポインタという意味

.	/	a	.	o	u	t	¥0	…	…
a	b	c	¥0	…	…	…	…	…	…
1	2	3	¥0	…	…	…	…	…	…
4	.	5	6	¥0	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…
…	…	…	…	…	…	…	…	…	…

もし二次元配列だったら…



ポインタに対する演算

- 下記の演算のみ可能：
  - ポインタと整数の加減算： 型のサイズを考慮してポインタを整数個分だけ前後にずらす
  - ポインタとポインタの減算： 型のサイズを考慮してその2つのポインタの間にいくつ変数が入るかを返す（整数値）
  - ポインタの比較： 中身のアドレスの大小比較
- それ以外の演算はコンパイルエラーになる

ポインタ演算の例

```
int a[10] = {10,20,30,40,50,60,70,80,90,100};
int *p, *q; p = a; q = &a[9];

printf("%d¥n", *a);         // 10
printf("%d¥n", *a+1);       // 11
printf("%d¥n", *(a+1));     // 20
printf("%d¥n", q - p);      // 9  (= 9 - 0)
if (p < q) {puts("p < q is true");}   // true

double *d1, *d2;
d1 = (double *) p;
d2 = (double *) q;
printf("%d¥n", d2 - d1);  // 4
```

*演算子の方が優先順位が高いので (*a)+1 と解釈される（よくある間違いなので注意）

ポインタの型キャスト（無いと warning が出るがそれでも一応動く）

double 型として考えた場合（小数点以下切り捨て）

ポインタと配列

- 前述の通り、int a[10] の a はポインタ
- 実は、a[...] という表記はポインタの syntax sugar
  - 読み書きを簡単にするために導入された(本質的には必要ない)構文

```
int a[10] = {10,20,30,40,50,60,70,80,90,100};

print("d¥n", a[0]);      // 10
print("d¥n", *a);        // 10
print("d¥n", *&a[0]);    // 10      (冗長だが...)
print("d¥n", a[9]);        // 100
print("d¥n", *(a + 9));    // 100
print("d¥n", 9[a]);        // 100  (実はこれでも動く(!))
```

ポインタと関数

- ポインタを渡すことで、呼ばれた側の関数に、呼び出し側の関数のメモリへのアクセスを許すことが出来る

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

int main() {
  char s[] = "hello world";

  puts(s); // hello world
  func(s, strlen(s));
  puts(s); // Hello World
}
```

main 関数では何もしていないのに値が書き換わっている

```
int func(char *str, int length) {
  int i;
  int status = 1;

  for (i = 0; i < length; i++) {
    if (status == 1) {
      str[i] =  toupper(str[i]);
      status = 0;
    }
    if (isspace(str[i])) {
      status = 1;
    }
  }
  return i;
}
```

文字列の最初の文字および空白の次の文字を大文字に変更する関数

main の中の配列を直接読み書きしている。func から戻ってもその結果は消えない

NULL

- 「どこも指していない」状態のポインタを作るための特別なアドレス。すべてのポインタに型キャストなしで代入可能。(null 文字とは別の概念)
  - Python における None と大体同じ。
  - これを参照するとプログラム実行が止まってしまうので注意。

```
#include <stddef.h>
char c;
char *p;

p = &c;      // 一旦 c を指した上で...
p = NULL;    // どこも指していない状態にする
if (p == NULL) puts("p is NULL");   // ポインタは比較可能
*p = 'a';    // Segmentation Fault でプログラム実行が停止
```

NULL はマクロで定義されているので include が無いとコンパイルエラーになる (stdio.h 等からも include されているのでそれらでも代用可)

NULL の実体

- stddef.h では以下のように定義されている：

```
#define NULL ((void *)0)
```

- void 型：「何型でもない」ことを表すための特別な型
  - 関数の引数や返戻値が無い場合は void 型を指定すれば良い。
  - void 型の変数は定義できないが、ポインタは定義できる。
  - 型が無いのでポインタの演算はできない。つまり単なるアドレス。
- NULL の代わりに (void *) 0 と直接書いても動くが、 読みやすさの為には NULL を使うことを推奨。

ポインタと構造体

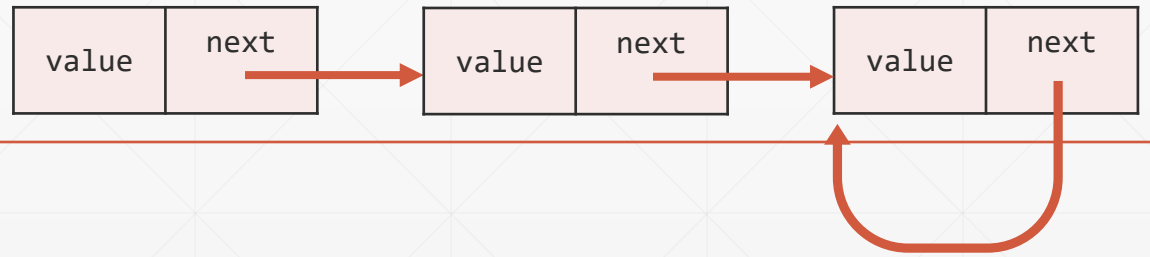
- ポインタで構造体を指す場合、* や . を使った表記の代わりにアロー演算子を使うことができる

```
typedef struct {
  double re;
  double im;
} complex_t;

complex_t a[10];
complex_t *p = a[0];
*(p).re = 0.0;
p->re = 0.0;     // 上と同じ意味になる（これも syntax sugar）
```

自己参照構造体

- 構造体の中に、自分自身と同じ型の構造体へのポインタを含むことができる
  - リストやツリーなどのデータ構造を構築する時によく使う。具体的な使い方については後日詳しく説明

```
struct list {
  int value;
  struct list *next;  // 自分と同じ型の構造体を参照可能（自分自身も可）
};
```



ポインタと文字列

- 文字列操作ではポインタは多用される（argv のように）
  - いまどの文字を処理しているのかを記憶
  - 部分文字列の生成
- 文字列を比較する際には注意が必要

```
char s[] = "Hello";
char t[] = "Hello";  // s とは異なるメモリ領域だが、中身が一緒

if (s == t) { ...; }   // これは成り立たない。アドレスが異なる
if (strcmp(s, t) == 0) { ...; }  // 中身が一緒なので成り立つ
if (strcmp(s, s) == 0) { ...; }  // 常に成り立つ
```

文字列操作の例： string tokenizer

- スペースで区切られた文字列を単語ごとに切り出す
  - より汎用的なものは string.h で strtok として定義されている

```
#include <string.h>

int main(void) {
  char s[] = "This is a pen.";
  char *p;

  p = tokenize(s);
  while (p != NULL) {
    puts(p);
    p = tokenize(NULL);
  }
}
```

tokenize に渡す文字列。
内容が破壊されるので注意

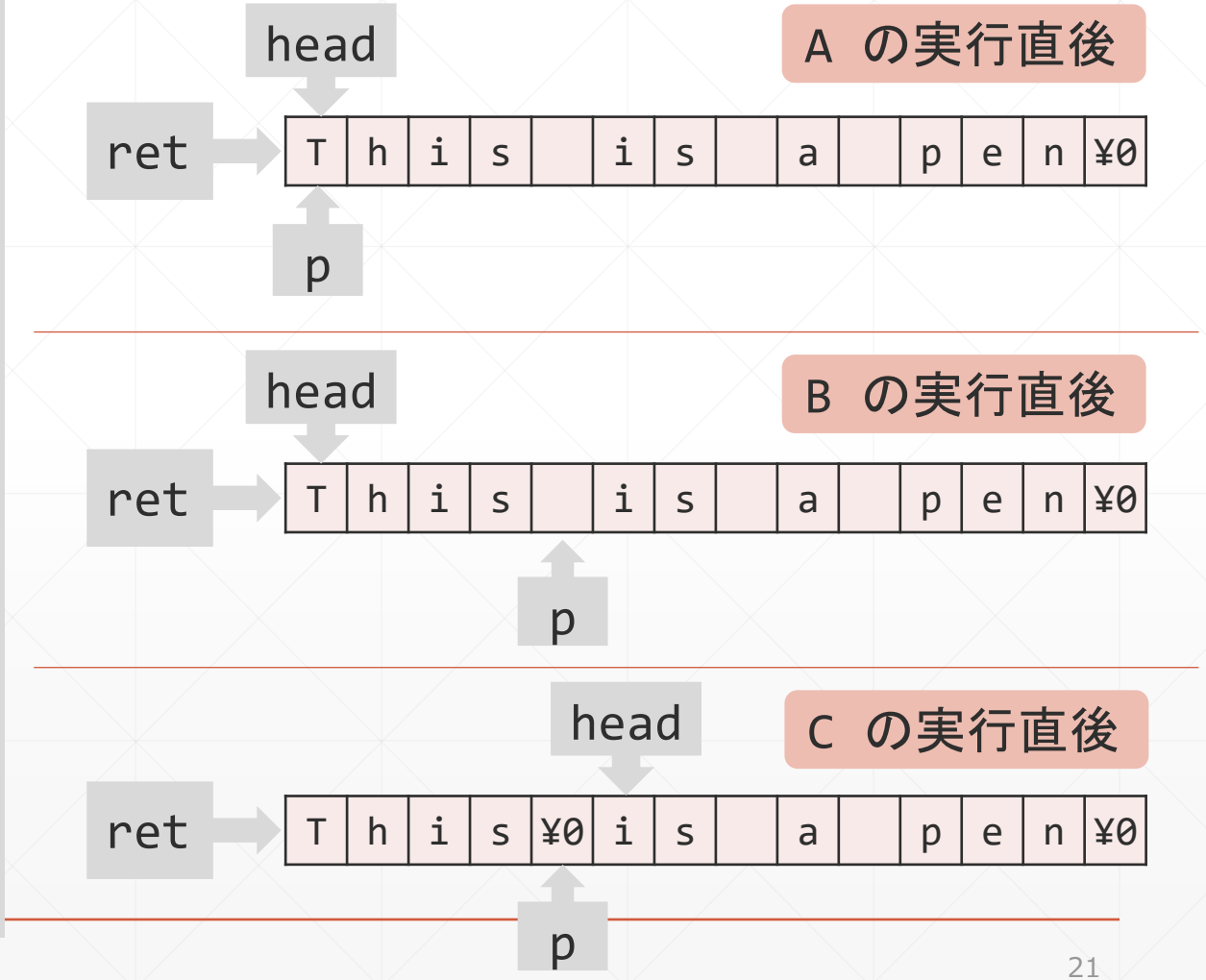
tokenize を初期化。
同時に最初の単語が返される

引数が NULL だと次の単語を返す

tokenize 関数の実装の一例

static で状態を保存している
(詳しい人向けの説明: thread safe ではない)

```
char *tokenize(char *init) {
  char *p, *ret;
  static char *head = NULL;

  if (head == NULL && init == NULL) return NULL;
  if (init != NULL) head = init;
  if (*head == '¥0') return NULL;

  p = head; ret = head; // A
  while (*p != '¥0' && *p != ' ') { p++; } // B
  if (*p == '¥0') {
    head = p;
  } else {
    *p = '¥0';
    head = p + 1;
  } // C

  return ret;
}
```



今日の演習

- tokenize 関数の中身を変更し、以下のうち最低２つの機能を実現せよ。ただし、配列表記(p[i])は使わずに、ポインタ表記(*p)を使うこと。
  - 空白だけでなく、複数種類の区切り文字（ピリオド、カンマ、セミコロン等）に対応できるようにする
  - 区切り文字が複数回連続していても一つの区切りと見なすようにする
  - ダブルクオート (") で囲まれた部分は、その中に区切り文字があっても区切らないようにする
  - 全トークンの切り出し終了後にinitで与えられた文字列が元の状態に戻っているようにする
  - thread safe にする（詳細は自分で調べること）
- 実現した機能が正しく動作していることを確認するmain関数も作成すること