

2019年度 計算機システム演習
第2回
2019.12.06

遠藤 敏夫(学術国際情報センター/数理・計算科学系 教授)
野村 哲弘(学術国際情報センター/数理・計算科学系 助教)

復習： ポインターと文法

	d	&d	*d
int d (intを保持)	dの値 (つまりintの値)	dのアドレス	
int *d (intの変数のアドレスを保持)	dの値 (dが指す変数のアドレス)	dのアドレス	dが指す変数の値

※ *：アドレスの参照先の値を出力



構造体とポインタ

(id, height)のペアをバブルソートするプログラム

sample19.c

```
#include <stdio.h>

void swap(int *x, int *y){
    int temp = *x;
    *x = *y;
    *y = temp;
}

void sort(int id[], int data[], int n){
    int k = n - 1;
    while ( k >= 1){
        int i;
        for (i = 1; i <= k; i++){
            if (data[i-1] > data[i]) {
                swap(&id[i-1], &id[i]);
                swap(&data[i-1], &data[i]);
            }
        }
        k -= 1;
    }
}
```

```
int main(){
    int i;
    int len = 5;
    int id[] = {100, 101, 102, 103, 104};
    int height[] = {184, 164, 175, 171, 179};
    sort(id, height, len);
    for (i = 0; i < len; i++){
        printf("%d: %d(id=%d)¥n", i, height[i], id[i]);
    }
    return 0;
}
```

- ▶ 問題点
 - ▶ idとheightの関連性がわかりづらい
 - ▶ 例えばweightを追加したいときswapも追加
- ▶ javaならStudentクラスの配列を作成
 - ▶ クラス名 : Student
 - ▶ フィールド : int id, int height

構造体

- ▶ 関連する個々のデータ (メンバ)をまとめて1つのデータとして扱える
 - ▶ Javaにおけるフィールドだけのクラス

- ▶ 定義方法

```
struct data {  
    char *name;  
    int val;  
};
```

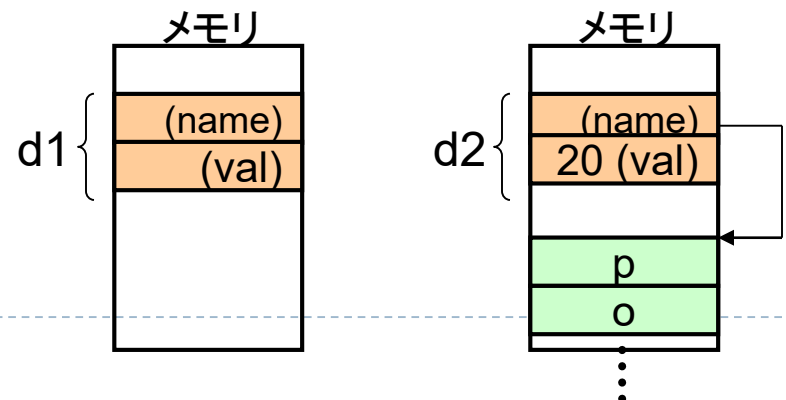
必要

- ▶ 関数の外でも内でも定義可能
 - ▶ 外: 全ての関数で利用できる
 - ▶ 内: その関数のみで使用

- ▶ 宣言&使用方法

- ▶ 各メンバにはドット演算子(.)を用いてアクセス

```
struct data d1;  
struct data d2 = { "poteto", 20 };  
d1.name = "tomato";  
d1.val = 10;  
printf("%s %d¥n",  
        d1.name, d1.val);
```



サンプル：構造体の使用方法

sample18.c

```
#include <stdio.h>
struct student {
    char *name;
    int height;
    double weight;
};
int main(){
    struct student sdt1 = {"ichiro", 180, 68.5};
    struct student sdt2;
    sdt2.name = "hanako";
    sdt2.height = 170;
    sdt2.weight = 60.2;

    printf("std1=%s, %d, %f¥n", sdt1.name, sdt1.height, sdt1.weight);
    printf("std2=%s, %d, %f¥n", sdt2.name, sdt2.height, sdt2.weight);
}
```

std1=ichiro, 180, 68.500000
std2=hanako, 170, 60.200000

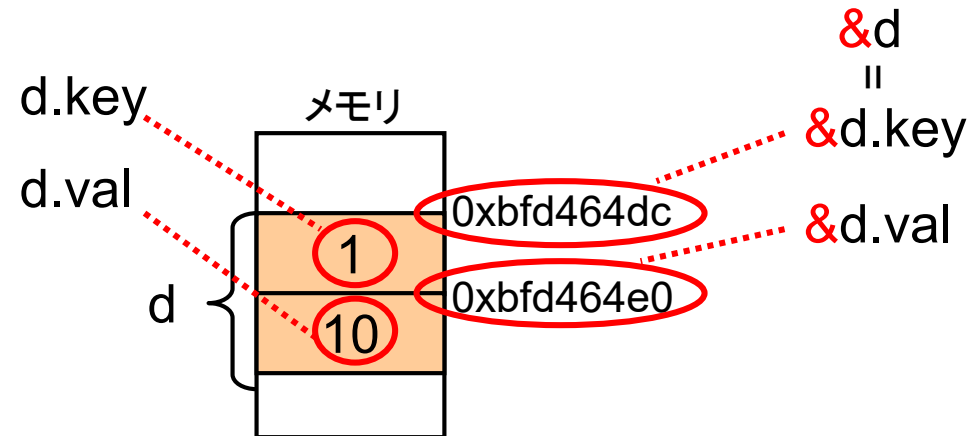
構造体のメモリ配置

- ▶ 各メンバは宣言された順にメモリ上に配置
 - ▶ 各々アドレスを持つ

```
struct student {  
    int key;  
    int val;  
};  
  
int main() {  
    struct data d = { 1, 10 };  
    printf("&d.key=%p¥n", &d.key);  
    printf("&d  =%p¥n", &d);  
    printf("&d.val=%p¥n", &d.val);  
    return 0;  
}
```

Output

```
&d.key=0xbfd464dc  
&d  =0xbfd464dc  
&d.val=0xbfd464e0
```



注) &d.key = &(d.key)

結合度: . > &

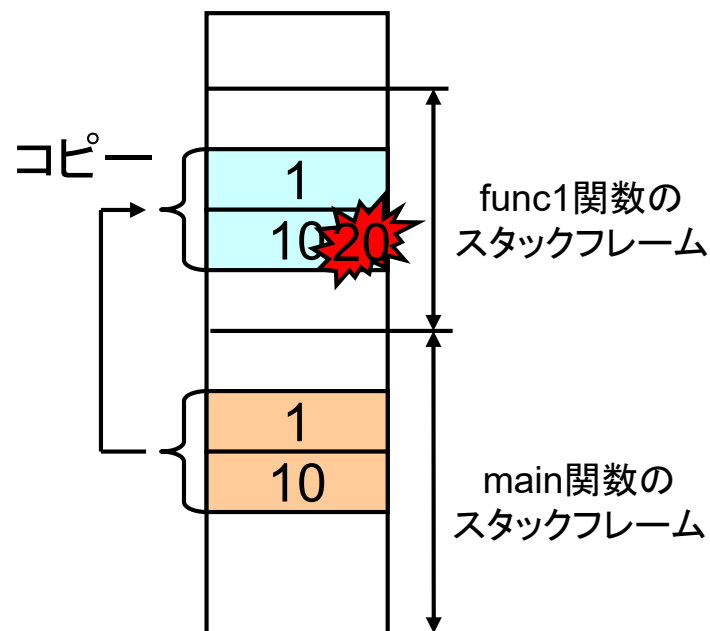
構造体と関数

▶ 構造体は値渡し

- ▶ 関数内で値を変更しても、呼び出し元は変更されない

sample21.c

```
#include <stdio.h>
struct data {
    int key;
    int val;
};
int func1(struct data d) {
    d.val = 20;
}
int main() {
    struct data d = {1, 10};
    printf("d.val = %d¥n", d.val);
    func1(d);
    printf("d.val = %d¥n", d.val);
    return 0;
}
```



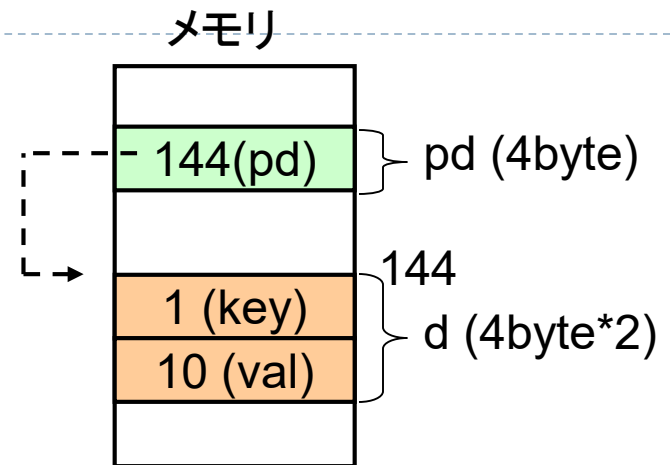
出力

```
d.val = 10
d.val = 10
```


構造体とポインタ

ポインタ変数の宣言、初期化

```
struct data d = { 1, 10 };  
struct data *dp;  
dp = &d;
```



ポインタを介したメンバへのアクセス

アロー演算子(→)を使用

dp->val と (*dp).val は同じ (記述の簡略化のため)

```
printf("( *dp ).val = %d¥n", (*dp).val); //10  
printf("dp->val  = %d¥n", dp->val); //10  
dp->val = 100;  
printf("dp->val = %d¥n", dp->val); //100  
printf("d.val  = %d¥n", d.val); //100
```

← 結合度: . > *

サンプル：構造体とポインタ

sample22.c

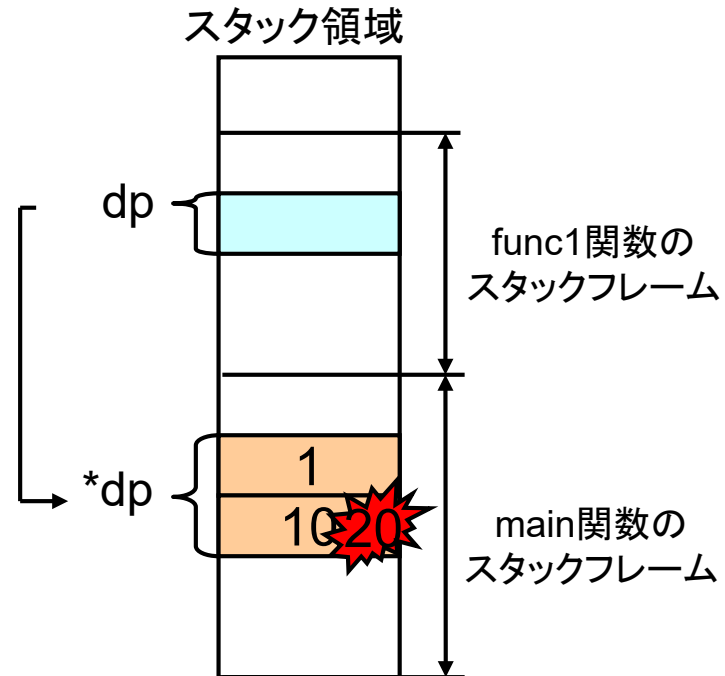
```
#include <stdio.h>

struct data {
    int key;
    int val;
};

int func1(struct data *d) {
    d->val = 20; // (*d).val=20
}

int main() {
    struct data d = {1, 10};
    printf("d.val = %d¥n", d.val);
    func1(&d);
    printf("d.val = %d¥n", d.val);
    return 0;
}
```

- ▶ ポインタを用いて、呼び出しもとの値を変更



d.val = 10
d.val = 20

(id, height)のペアをバブルソートするプログラム2

sample20.c

```
#include <stdio.h>
struct data {
    int id;
    int height;
};
void swap(struct data *dx, struct data *dy)
{
    struct data d = *dx;
    *dx = *dy;
    *dy = d;
}
void sort(struct data d[], int n)
{
    int k = n - 1;
    while (k >= 1){
        int i;
        for (i = 1; i <= k; i++){
            if (d[i-1].height > d[i].height) {
                swap(&d[i-1], &d[i]);
            }
        }
        k -= 1;
    }
}
```

```
int main()
{
    int i;
    int len = 5;
    struct data d[] = {
        {100, 184},
        {101, 164},
        {102, 175},
        {103, 171},
        {104, 179}
    };
    sort(d, len);
    for (i = 0; i < len; i++)
        printf("%d:%d(id=%d)¥n", i, d[i].height, d[i].id);
    return 0;
}
```

構造体とポインタ (まとめ)

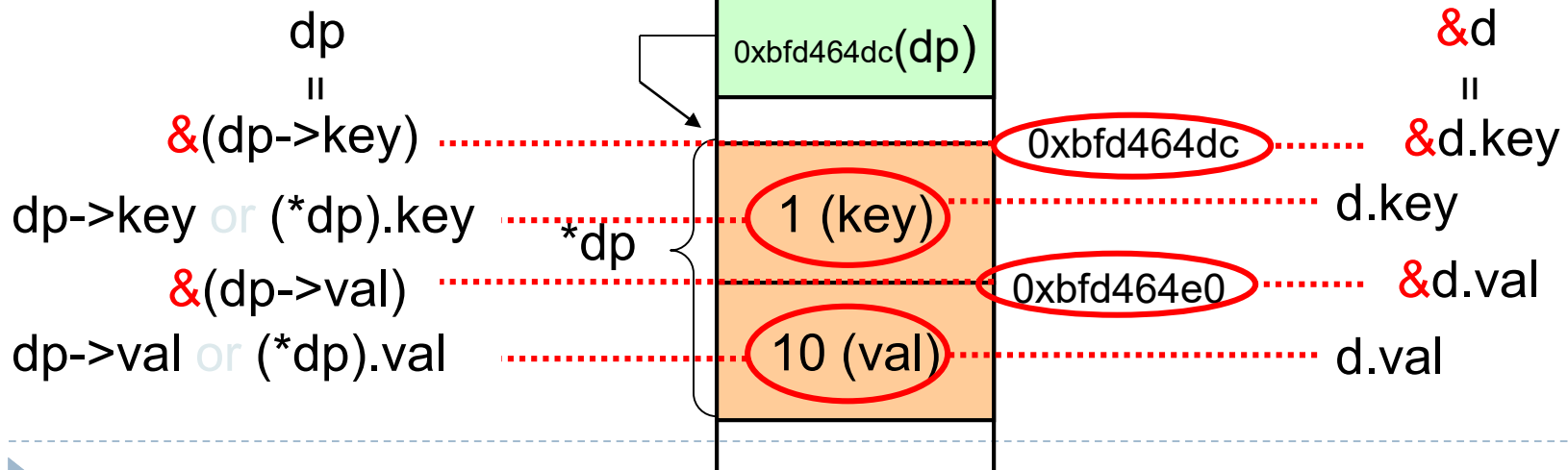
```
struct data {  
    int key;  
    int val;  
};
```

```
struct data d = {1, 10};  
struct data *dp;  
dp = &d;
```

構造体のポインタ
でアクセス

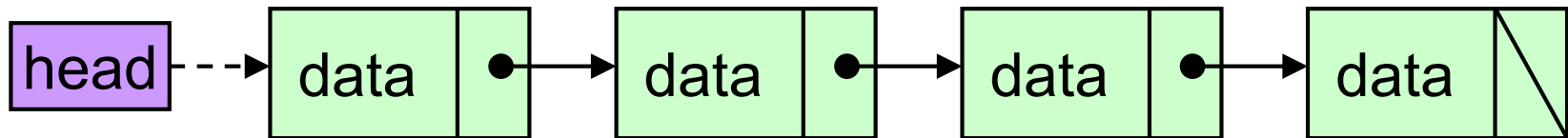
構造体
でアクセス

メモリ



構造体とポインタによる 連結リストの実装

構造体の活用： 連結リスト



- ▶ 構造体の使用例
 - ▶ 連結リスト、ツリー構造、キュー、スタック
- ▶ データをリンクで数珠繋ぎにしたリスト
 - ▶ 連結リストを構成する各要素をノードと言う

ノードのデータ構造

▶ 構造体で定義

- ▶ headは先頭ノードを指す特別なnode構造体型データ

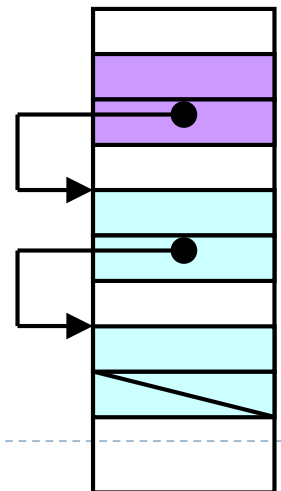
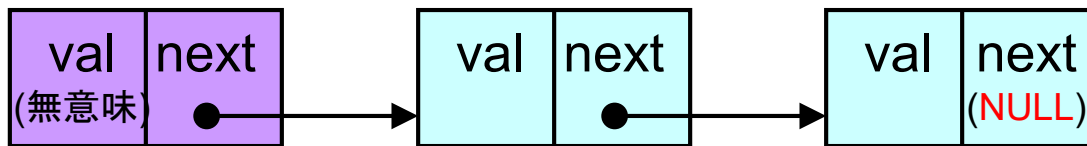
```
struct node {  
    int val;  
    struct node *next;  
};
```

データ

次のノードを指すポインタ



head



連結リストの基本的な関数

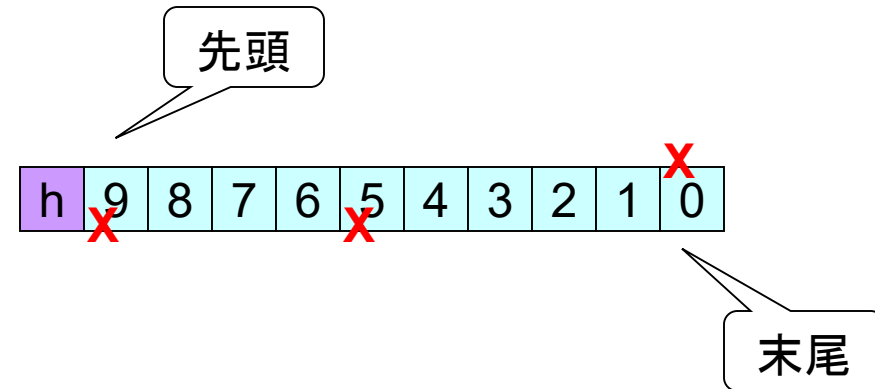
- ▶ `int add (struct node *head, int val);`
 - ▶ head が指すリストの先頭に val を追加
 - ▶ 追加に成功したら0を、失敗したら-1を返す
- ▶ `int deleteFirst (struct node *head);`
 - ▶ head が指すリストから先頭要素 (headの次のノード)を削除
 - ▶ リストが空なら-1を、空でなければ先頭要素の値を返す
- ▶ `int delete (struct node *head, int val);`
 - ▶ head が指すリストから要素 val を削除
 - ▶ val がリスト中にあれば val を、無ければ-1を返す
- ▶ `void display (struct node *head);`
 - ▶ head が指すリスト中の要素を全て表示



使い方

sample23.c

```
int main() {  
    struct node head = {-1, NULL};  
    int nums[] = {0,1,2,3,4,5,6,7,8,9};  
    int i;  
    for (i = 0; i < 10; i++){  
        int res = add(&head, nums[i]);  
        if (res != 0) return 1;  
    }  
    deleteFirst(&head); // 9を削除  
    delete(&head, 9); // -1を返す  
    delete(&head, 5);  
    delete(&head, 0);  
    display(&head);  
    return 0;  
}
```



出力

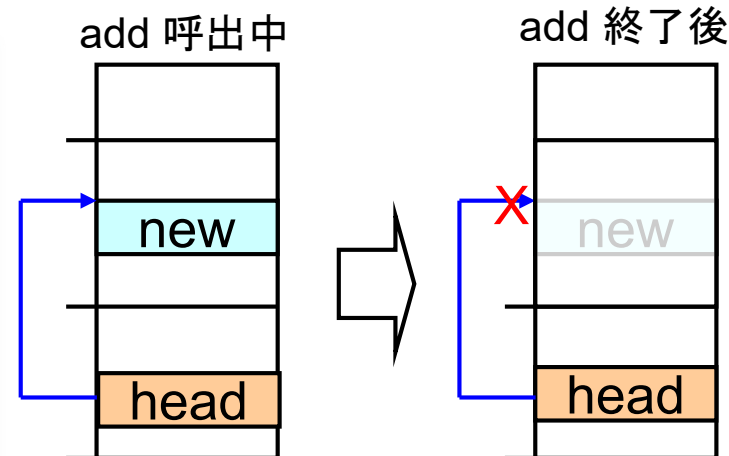
```
1: 1  
2: 2  
3: 3  
4: 4  
5: 6  
6: 7  
7: 8
```

add関数 (悪い例)

- ▶ ローカル変数 (struct node newnode) は使えない
 - ▶ newはスタック領域上のadd関数のスタックフレーム内に作られる
 - ▶ add終了時にはスタックフレームごと削除される

```
int add(struct node *head, int val) {  
    struct node newnode;  
  
    newnode.val = val;  
    newnode.next = head->next;  
    head->next = newnode;  
    return 0;  
}
```

ダメ



➡ 確保したメモリ領域がadd関数後も削除されないようにする必要がある

malloc, free 関数

- ▶ `void *malloc(size_t size)`
 - ▶ ヘッダファイル `stdlib.h` (`#include <stdlib.h>`)
 - ▶ `size` **バイト**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
 - ▶ **ヒープ領域**上に確保する⇒プログラム実行中消えることはない
 - ▶ `size_t`型は多くの処理系で「`unsigned long (int)`」型
 - ▶ ポインタさえ取得できれば全ての関数からアクセス可能
 - ▶ `void`ポインターはどのType型にも代入可能
 - ▶ 一般に代入される型にキャストする
 - ▶ javaの**new**に相当
 - ▶ `void free(void *ptr)`
 - ▶ ヘッダファイル `stdlib.h`
 - ▶ `ptr`が指すヒープ領域上のメモリ領域を解放
- ⇒ ヒープ領域上のデータは明示的に削除しないと、プログラム終了時までメモリを消費し続ける。
(開放し忘れを「メモリリーク」という)
- ▶ javaではGC(ガベージコレクター)が使用されなくなったメモリ領域を解放している。



型のサイズ

▶ sizeof演算子を用いて取得

▶ `size_t size = sizeof(Type)`

▶ Type型のサイズをバイト単位で返す

▶ size_t型は多くの処理系で「unsigned long (int)」型

```
struct data {  
    int key;  
    int val;  
};  
int main() {  
    size_t size = sizeof(struct data);  
    printf("%d¥n", size);  
    printf("%lu¥n", (unsigned long)size);  
    return 0;  
} // とともに8を出力
```

小さいサイズであれば
int型として表示もできる

正しい(安全)書き方

add 関数

sample23.c

```
int add(struct node *head, int val) {
```

```
    struct node *new;
```

```
    new = (struct node *)malloc(sizeof(struct node));
```

```
    if (new == NULL) return -1;
```

```
    new->val = val;
```

```
    // head->nextがNULLかチェック
```

```
    new->next = head->next;
```

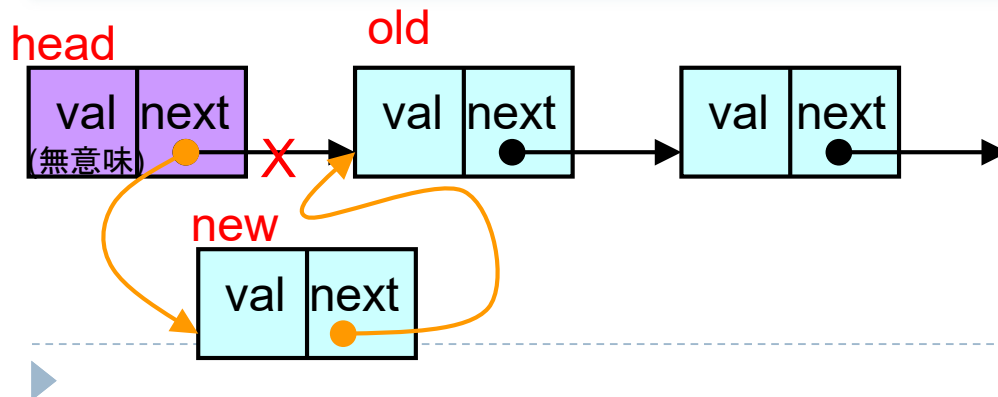
```
    head->next = new;
```

```
    return 0;
```

```
}
```

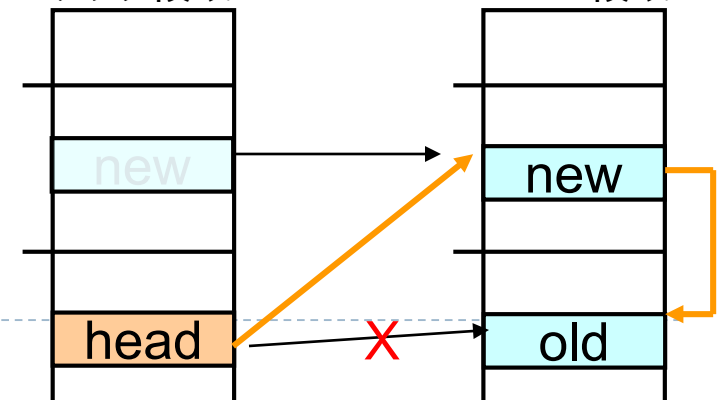
メモリ確保

確保されているかしっかりチェックすること



スタック領域

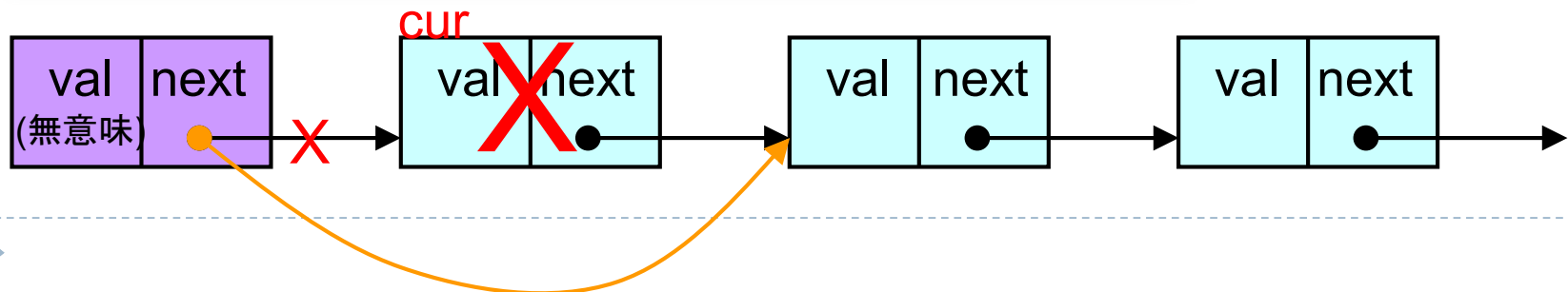
ヒープ領域



deleteFirst 関数

sample23.c

```
int deleteFirst(struct node *head) {  
    struct node *cur;  
    int result = -1;  
    if (head->next != NULL) {  
        cur = head->next;  
        head->next = cur->next;  
        result = cur->val;  
        free(cur);  
    }  
    return result;  
}
```



delete 関数

sample23.c

```
int delete(struct node *head, int val) {
    struct node *cur, *prev;
    int result = -1;
    // head->next == NULL check
    for (cur = head->next, prev = head; cur != NULL;
        cur = cur->next, prev = prev->next) {
        if (cur->val == val) {
            prev->next = cur->next;
            result = cur->val;
            free(cur);
            break;
        }
    }
    return result;
}
```

typedef

- ▶ 既存の型に対して同義語を与える宣言

```
typedef <既存型> <新規型>;
```

- ▶ 例) `typedef int NUMBER;` `typedef unsigned long size_t`
- ▶ メリット1: 読みやすさ・書きやすさ向上
 - ▶ 毎回 `struct data` と書く必要がない

```
struct _data {  
    int key;  
    int val;  
};  
typedef struct _data data;
```

同じ



```
typedef struct {  
    int key;  
    int val;  
} data;
```

- ▶ メリット2: コードに影響を与えず、既存型を置き換えられる
 - ▶ `typedef int NUMBER;` \Rightarrow `typedef double NUMBER;`

typedef (例)

sample31.c

```
#include <stdio.h>
struct _map1 {
    int key;
    int value;
};
typedef struct _map1 map1;

typedef struct {
    int key;
    int id;
    int value;
} map2;

int main() {
    map1 m1 = {1, 10};
    map2 m2 = {2, 2, 20};
    printf("m1={%d, %d}¥n", m1.key, m1.value);
    printf("m2={%d, %d, %d}¥n", m2.key, m2.id, m2.value);
}
```

```
m1={1, 10}
m2={2, 2, 20}
```

#define

sample32.c

```
#include <stdio.h>
#define LENGTH 5

int main()
{
    int i;
    int nums[LENGTH] = {0,1,2,3,4};

    for (i=0; i < LENGTH; i++){
        printf("%d¥n",nums[i]);
    }
    return 0;
}
```

- ▶ コンパイル前に指定した値に置換する

#define <記号定数名> <値>

- ▶ #define NUMBER 5
- ▶ 利点
 - ▶ 値の管理を1箇所に集約
 - ▶ プログラム中の数字(マジックナンバー)に名前を与えることで可読性向上

<C言語編>のまとめ

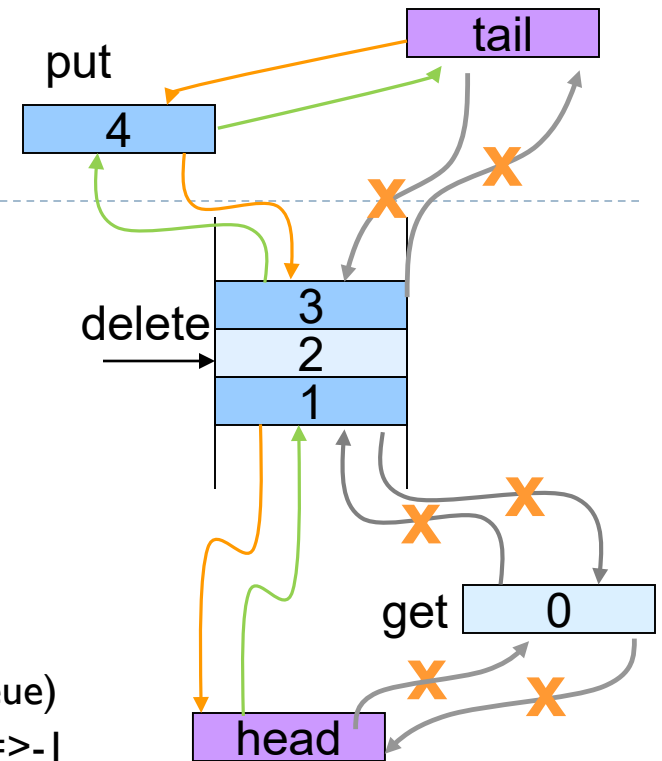
- ▶ Javaよりも低レベルなC言語を使うことにより、背後の計算機のシステムが少し見えた
- ▶ メモリ上の値はアドレスで参照することができ、C言語ではポインタを用いる
- ▶ 関数を呼び出すとコールスタックによって呼び出し関係が管理される。ローカル変数は、関数のスタックフレーム上に確保され、関数が終了すると破棄される
- ▶ 命令とデータはメモリ上に一緒に格納され(フォン・ノイマン型コンピュータ)、C言語の関数も単なるデータに過ぎない。C言語では関数へのポインタが利用可能
- ▶ データを関数として実行することもできる！



今日の課題

課題1: Queueの実装

- ▶ 連結リストを参考にQueueを作れ
 - ▶ Queue: First In, First Outなリスト
 - ▶ 実装する関数
 - ▶ **int put (struct node *tail, int val)**
 - Queue (tail 側) にデータを追加 (enqueue)
 - 戻り値: 成功=>0、失敗=>-1
 - ▶ **int get (struct node *head)**
 - Queue (head 側) からデータを取得 & 削除 (dequeue)
 - 戻り値: データが存在する=>val、存在しない=>-1
 - ▶ **int delete (struct node *head, int val)**
 - Queue から値がvalである要素を削除 (1個だけで良い)
 - 戻り値: データが存在する=>val、存在しない=>-1
 - ▶ **void display (struct node *head)**
 - Queueの内容を“head側”から順に表示
 - ▶ 動作を確認するためのmain関数も記述すること
 - ▶ put、getするデータの出力、displayによる出力を行い、実行結果もレポートにまとめること



```
struct node {  
    int val;  
    struct node *next;  
    struct node *prev;  
};
```

注:これが必要十分である保証はない

課題1の捕足: typedefの使い方

- ▶ 最終的に型名を“ node “として使いたい場合

例. 1

```
typedef struct _node
{
    int val;
    struct _node *next;
    struct _node *prev;
} node;
```

例. 2

```
struct _node
{
    int val;
    struct _node *next;
    struct _node *prev;
};
typedef struct _node node;
```

課題提出

- ▶ ✕切: **12/20 (金) 23:59**
 - ▶ OCW-iから提出すること
 - ▶ 遅れても(減点しますが)受け付けます。
- ▶ 提出物: 以下のファイルをzip形式で圧縮したもの
 - ▶ ドキュメント (pdf, txt 形式)
 - ▶ プログラムソースの簡単な説明、グラフ、工夫したところ
 - ▶ プログラムの実行結果
 - ▶ 感想、質問等
 - ▶ プログラムソース (課題I)
 - ▶ テスト用のmain関数も含む(コンパイルできて正しく実行できること)
 - ▶ 全てのファイル名は半角英数字でお願いします
 - ▶ 文字化け防止のため

第1回課題 フィードバック

- ▶ 12/6 10:30 までに提出された課題は確認済みです
 - ▶ 問題点がある方にはメールしていますので、現時点でメールが来ていない場合は合格点を取れています。
- ▶ 注意点
 - ▶ `main()` と `monte_carlo()`, `sort_int_array()` は別関数にして、他の人が `monte_carlo()` や `sort_int_array()` を呼び出す可能性を意識して書いてください。
 - ▶ もしかしたら間違った引数で呼び出すかもしれません。



おまけ: コマンドラインで問題サイズを取る

- ▶ こんな感じで書くことができます

- ▶

```
int main(int argc, char **argv) {  
    int problem_size = 100; // 規定値  
    if (argc > 1) { // 引数がある場合  
        problem_size = atoi(argv[1]); // 第1引数をintに変換  
        // ちなみにargv[0]はプログラム名  
    }  
}
```

- ▶ エラーチェックは適宜行う必要があります



課題締め切り

- ▶ 第01回 (11/29 出題)
 - ▶ 12/13 (金)
- ▶ 第02回 (本日出題)
 - ▶ 12/20 (金)
- ▶ 第03回 (12/13 出題)
 - ▶ 1/7 (火)
- ▶ 第04回 (12/20 出題)
 - ▶ 1/7 (火)

