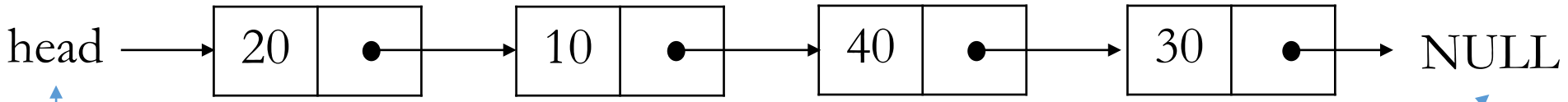
連結リスト

実装への準備

連結リスト(Linked list)とは

- 一連のノード(＝セル。以降ノードと呼ぶ)が、次または前のノードを指定するリンクを持つものが連結リスト
  - 線形リスト(片方向リスト・双方向リスト)・循環リストなどがある
  - 今回は線形リストの片方向リストを扱う
- 上記定義からすれば前回講義DATA-1Q-1-配列で紹介した「新しいデータ構造」は一種の連結リスト
- それの欠点をなくすのが、(C言語の場合)ポインタをリンクとして用いる方法
- 連結リストと言えばこのポインタを用いるものを指す
  - 実装にはある程度高度なプログラミングが必要

連結リストの例（イメージ）

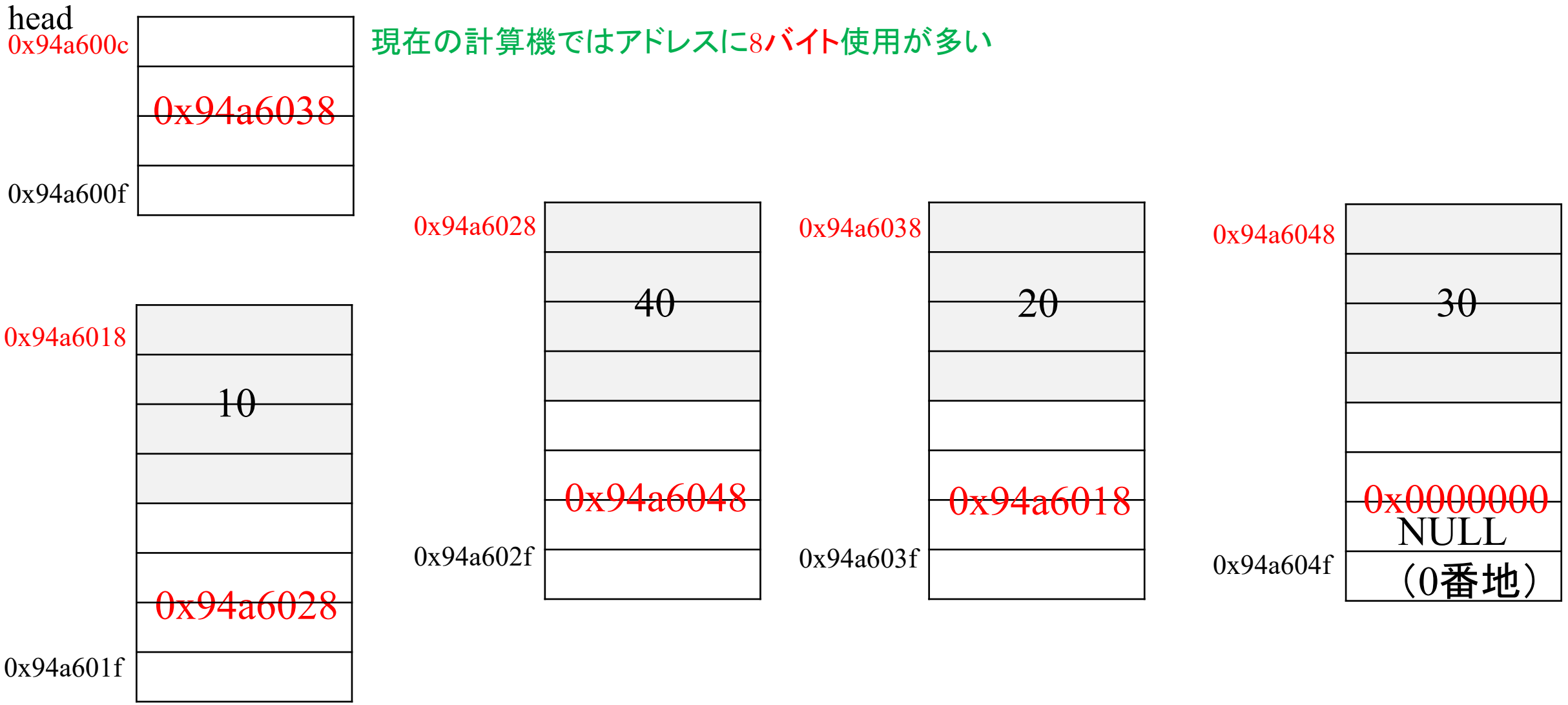


ノード
（データ部とポインタ部）

先頭ノードへのポインタ

NULLポインタ
「何もない」ことを表すマクロ定義の定数。
ここではそれ以上ノードがない。つまり、
連結リストの最後

メモリ上の表現（データがint型の場合）



実装への準備

- 連結リストの作成とそれへの操作（ノードの削除や挿入）の実装（プログラミング）には以下の知識（準備）が必要
  - ポインタ（復習）、ポインタのポインタ（発展）
  - メモリの動的確保と解放：malloc, free関数
  - 構造体（復習）、自己参照構造体

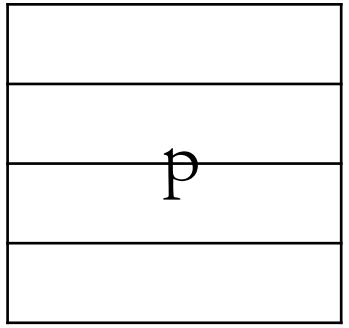
ポインタとは

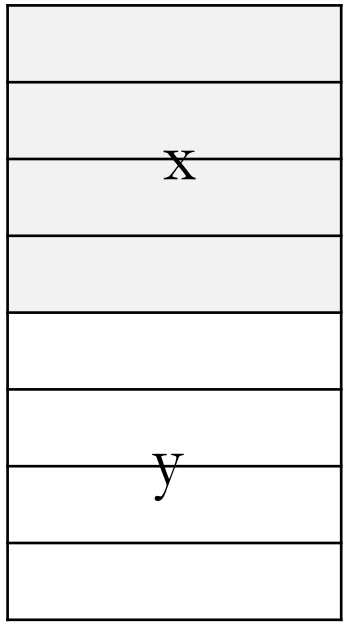
- ポインタはアドレスを扱う変数
- 名前は他の要素を指し示す（pointする）ということから由来

int *p, x;
の場合
p=&x で「pがxを指し示す」ことになる。つまり、
pの値がxの（メモリの）アドレスである

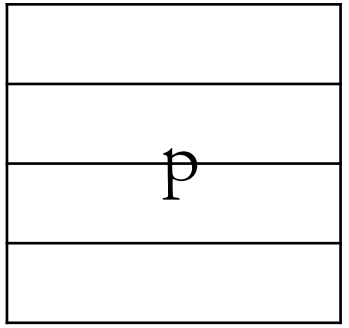
ポインタの使い方とメモリ上の表現

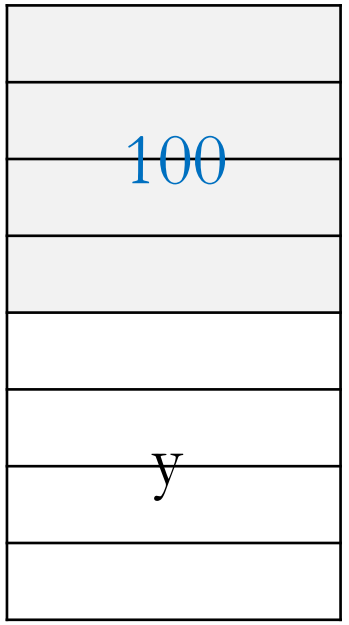
int *p, x, y;



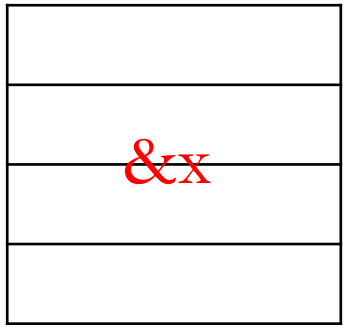


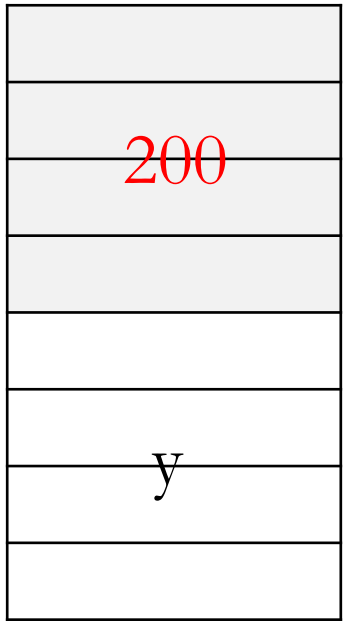
x=100;



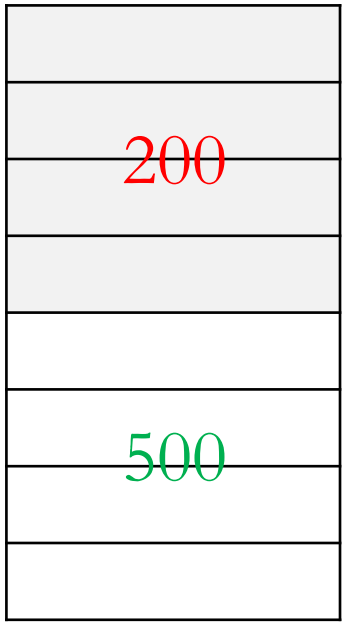


p=&x; *p=200;





p=&y; *p=500;



Question

- 実行結果を答えなさい。

```
int *p, x=100, y=200;
p=&x;
printf(“%d¥n”, *p);
p=&y;
printf(“%d¥n”, *p);
*p=1000;
printf(“%d¥n”, *p);
printf(“%d¥n”, x);
printf(“%d¥n”, y);
```

manaba小テスト：02-1

- 10分
- 6点

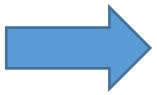
ポインタのポインタ

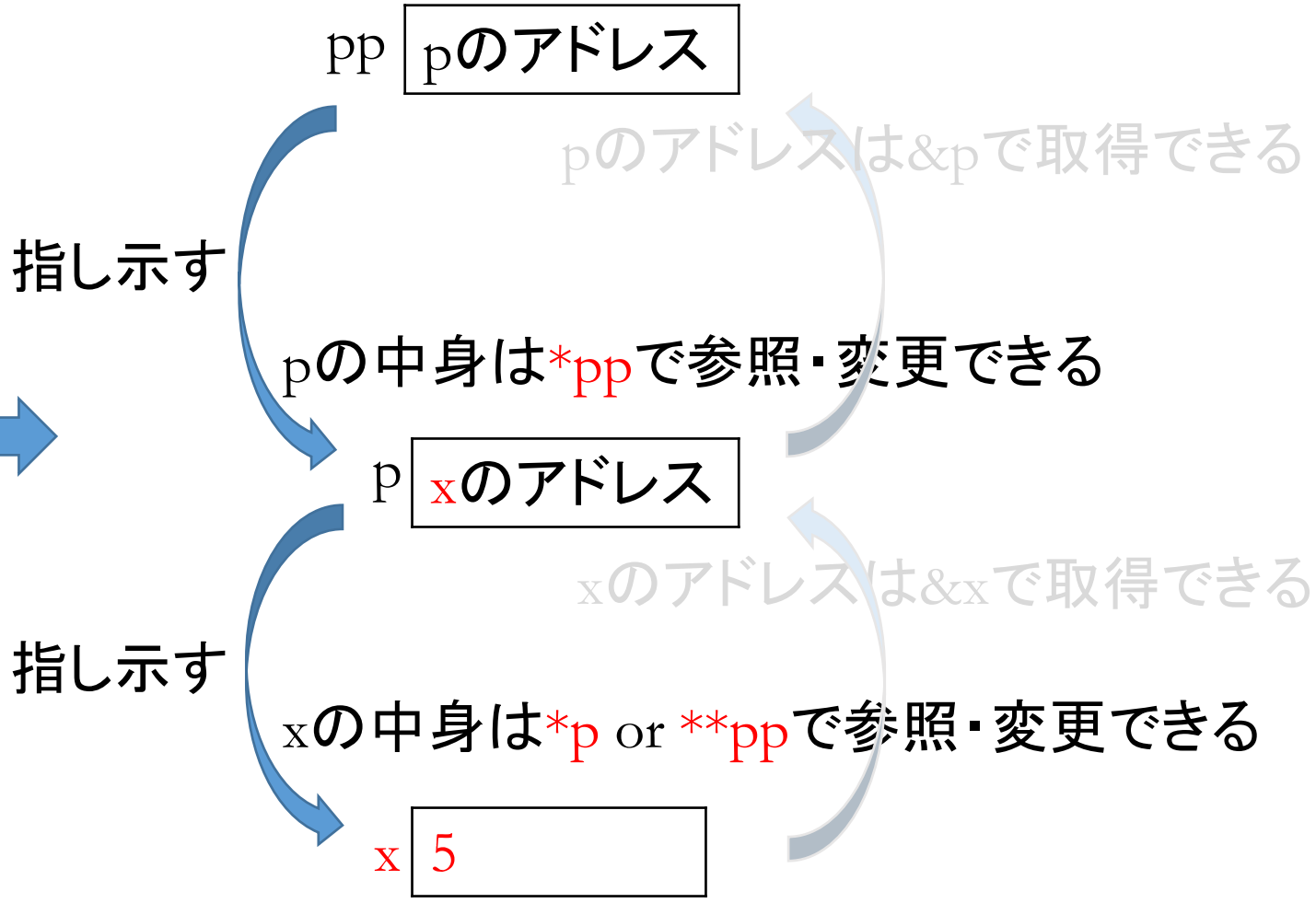
- ポインタはデータを記憶しているメモリを指し示すもの（＝メモリのアドレス）
- ポインタのポインタはそのポインタ（メモリのアドレス）を記憶しているメモリを指し示すもの（＝メモリのアドレスを記憶するメモリのアドレス。以降「アドレスのアドレス」と略す）

- 連結リストにおいてはポインタ部（ポインタを記憶するメモリ）を指し示すのに使われる

ポインタのポインタ

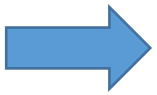
```
int *p, **pp;
int x=5, y=-5;
p=&x;
pp=&p;
//ppがxのアドレスを指し示す
//pの中身がxのアドレス
```

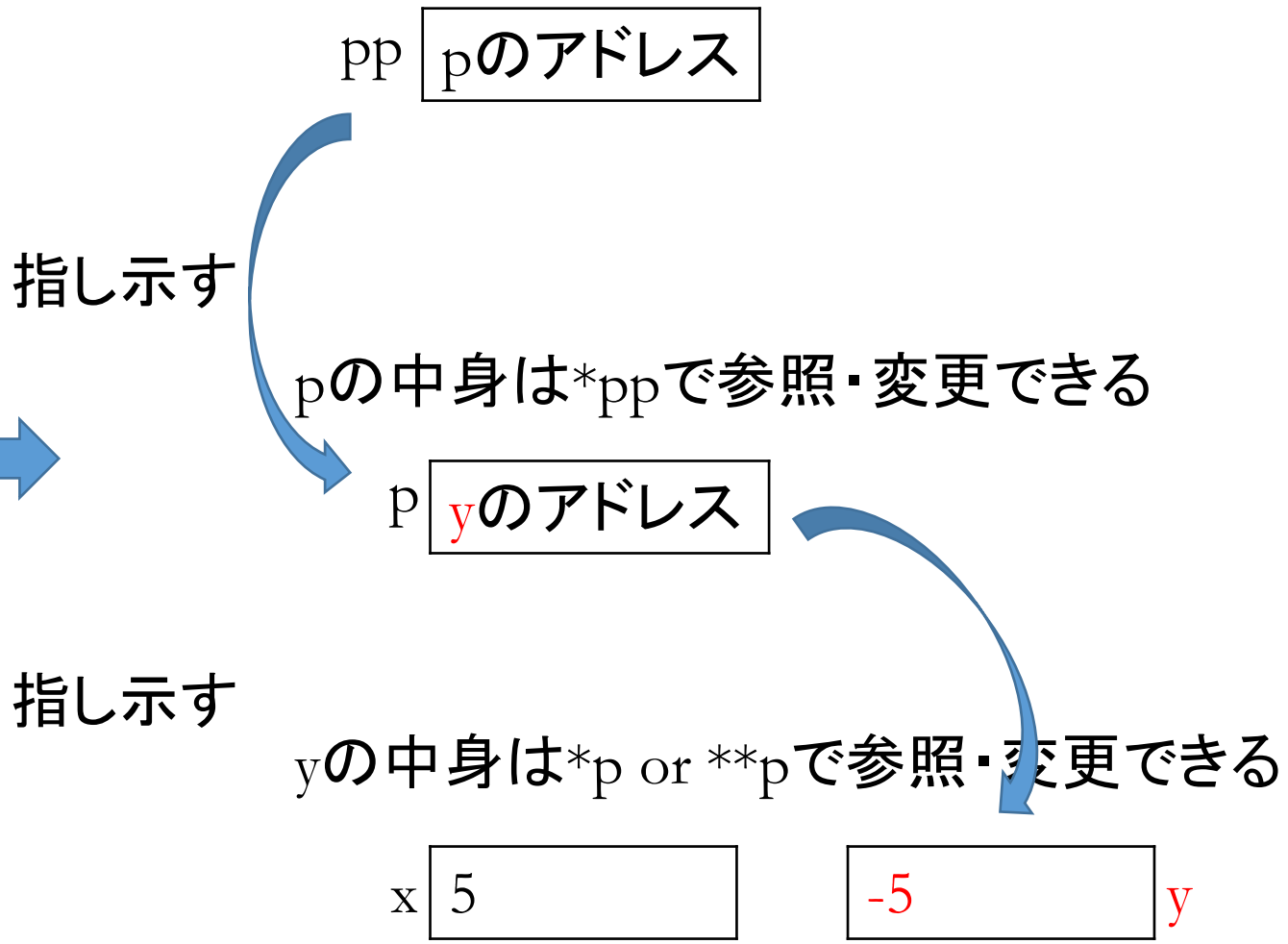




ポインタのポインタ

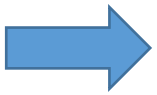
```
int *p, **pp;
int x=5, y=-5;
p=&x;
pp=&p;
//ppがxのアドレスを指し示す
*pp=&y;
//ppがyのアドレスを指し示す
//pの中身がyのアドレス
```

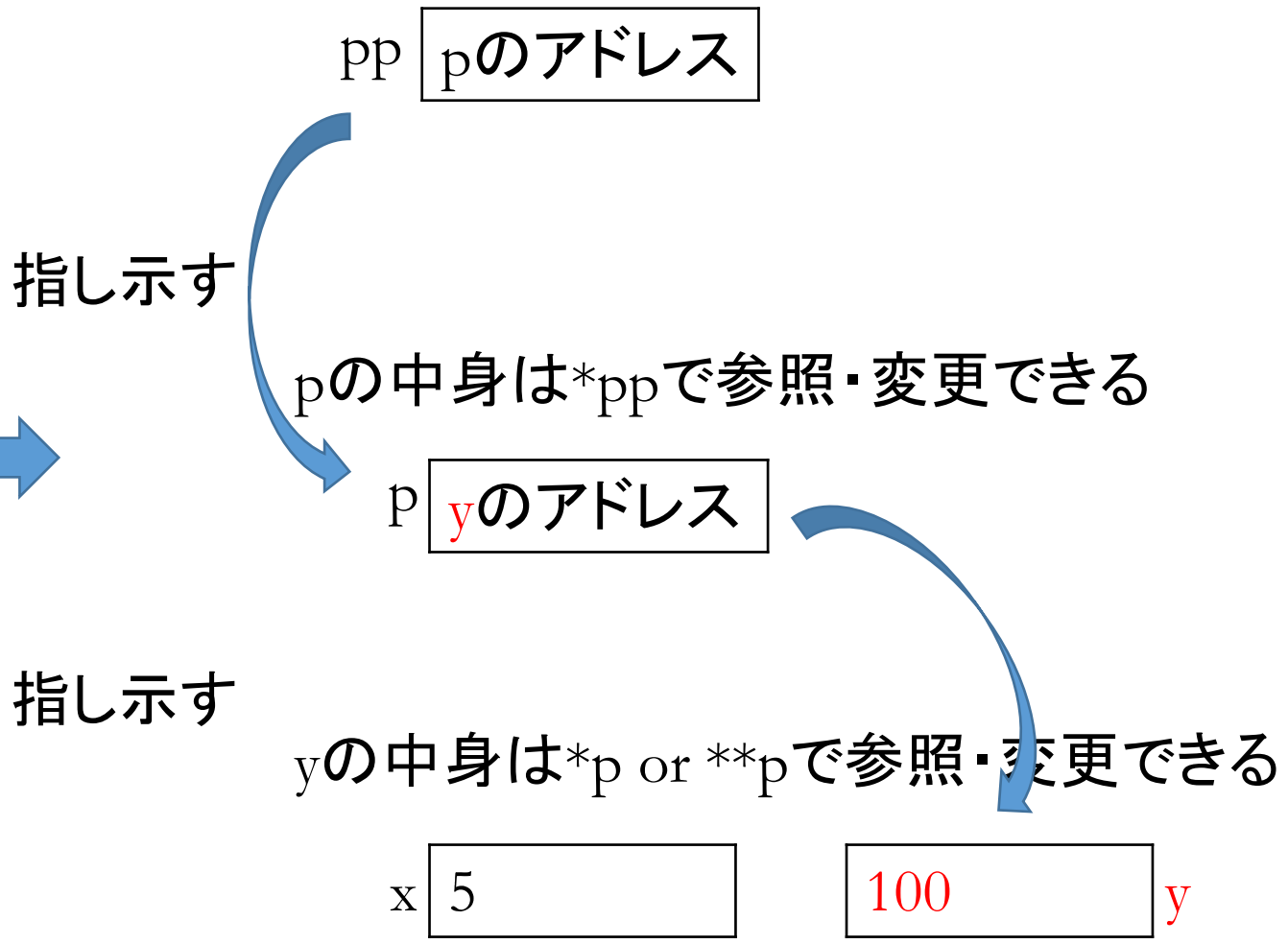




ポインタのポインタ

```
int *p, **pp;
int x=5, y=-5;
p=&x;
pp=&p;
//ppがxのアドレスを指し示す
*pp=&y;
//ppがyのアドレスを指し示す
//pの中身がyのアドレス
*p=100; //**pp=100;
```





Question

- 実行結果を示しなさい。

```
int *p, **pp, x=5, y=-5;
p=&x;
pp=&p;
printf("%d¥n", **pp);
*pp=&y; // p=&y;
printf("%d¥n", **pp);
**pp=100; // *p=100;
printf("%d %d¥n", x, y);
*pp=&x;
*p=500;
printf("%d %d¥n", x, y);
```

構造体

- 構造体は、char, int, doubleなど異なるデータ型を必要に応じてまとめ、自ら定義したデータ型である
- たとえば以下のような表を扱うとき、1レコード（1行）を1データとして考えて、それを1つの構造体で定義すると便利

学籍番号	氏名	所属学科	成績
000010	龍谷太朗	数理情報	100
000105	瀬田花子	電子情報	90
⋮	⋮	⋮	⋮
000100	大津三郎	知能情報	80

構造体の定義(例)

学籍番号	氏名	所属学科	成績
000010	龍谷太朗	数理情報	100
000105	瀬田花子	電子情報	90
⋮	⋮	⋮	⋮
000100	大津三郎	知能情報	80

```
typedef  struct rc{
  char id[128], name[128], dep[128];
  int score;
} RC;
```

メンバー

構造体の使用例

構造体の単純変数

```
typedef  struct {
   int x, y;
}PT;

int main(void)
{
   int i;
   PT a;
   a.x=10; a.y=20;
   ⋮
}
```

構造体のポインタ変数

```
int main(void){
   PT a, *p;
   a.x=10;
   a.y=20;
   p=&a;
   p->x=100;
   printf("a.x=%d, a.y=%d¥n", a.x, p->y);
   return 0;
}
```

構造体の使用例

構造体の配列変数

```
typedef struct {
  char s[64];
  int p;
}DATA;
int main(void){
  int head=2;
  DATA a[100];
  strcpy(a[0]. s, “東京”); a[0].p=-1;
  strcpy(a[1]. s, “京都”); a[1].p=3;
  strcpy(a[2]. s, “新大阪”); a[2].p=1;
  strcpy(a[3]. s, “名古屋”); a[3].p=4;
  strcpy(a[4]. s, “新横浜”); a[4].p=0;
  return 0;}
```

```
int main(void){
  int head = 2;
  DATA a[100] = {
          {"東京", -1},
          {"京都", 3},
          {"新大阪", 1},
          {"名古屋", 4},
          {"新横浜", 0}};
   return 0;
}
```

自己参照構造体

- 定義

```
typedef  struct node{
    int data;                  // データ型は構造体でもなんでもOK
    struct node *next; // 自己参照構造体
} NODE;
```

自己参照構造体

- 定義

```
typedef int data_t; // データ型をここで定義しておくと、変更などに便利

typedef struct node{
    data_t data;
    struct node *next; // 自己参照構造体
} NODE;
```

- 連結リストのノードとの対応

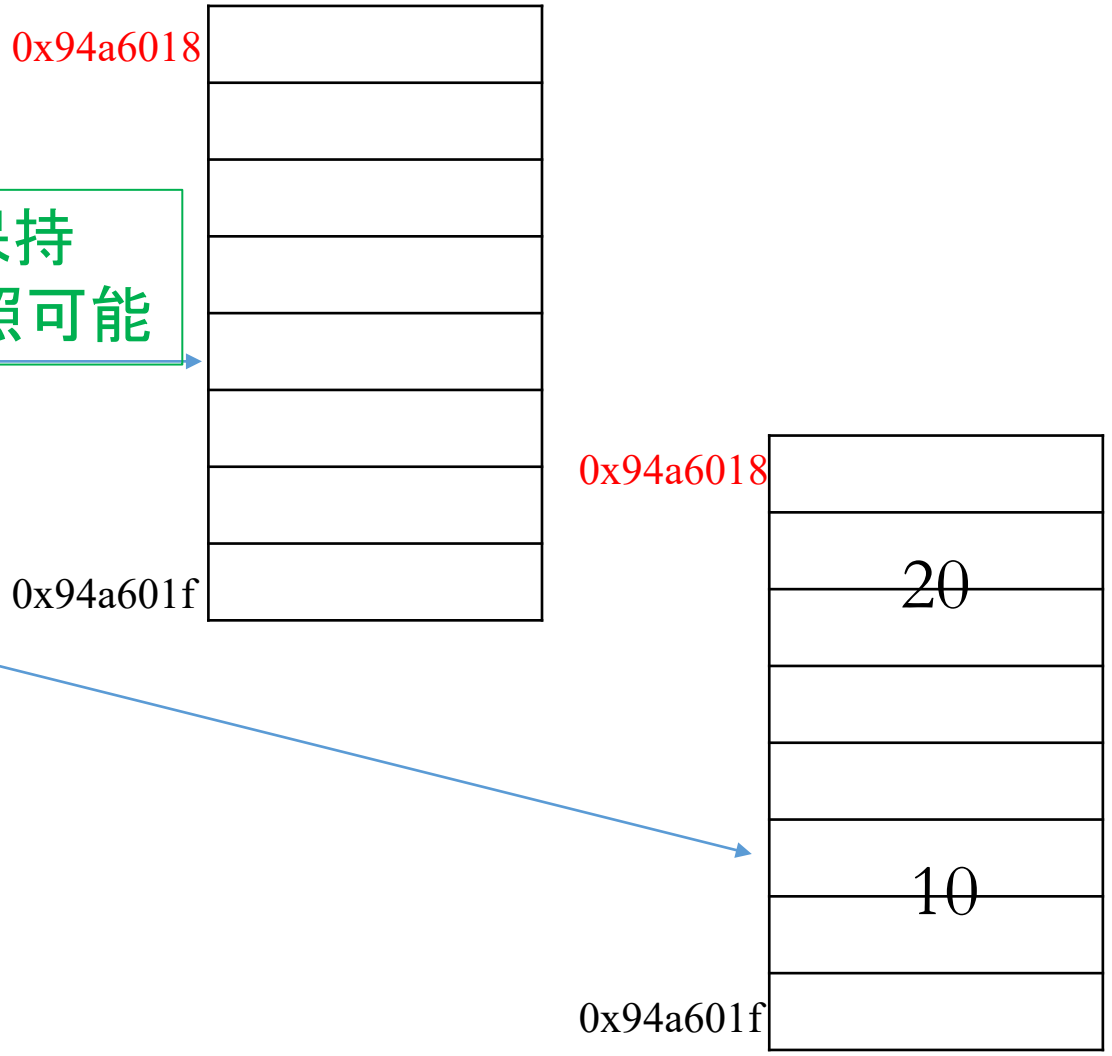
メモリの動的確保と解放

- 連結リストは、配列のようにデータの追加をあらかじめ予測してメモリを大きめに確保する必要はない
- int a[100]のような宣言で静的に（プログラム実行の最後まで）メモリを確保するのではなく、プログラムの中で必要に応じて、その都度メモリを確保したりその確保をやめたりすることが必要
- 前者をメモリの動的確保という。ノードの追加に対応。後者をメモリの解放という。ノードの削除に対応
- メモリの動的確保にはmalloc()関数、メモリの解放にはfree()関数を使う

メモリの静的確保（配列の例）

```
#include <stdio.h>
int main(void)
{
  int i, a[2];
  printf("Input the values¥n");
  for(i=0; i<2; i++)
    scanf("%d", &a[i]);
     ⋮
}
```

(1)メモリを確保。終了まで保持
(2)aでメモリのアドレスを参照可能

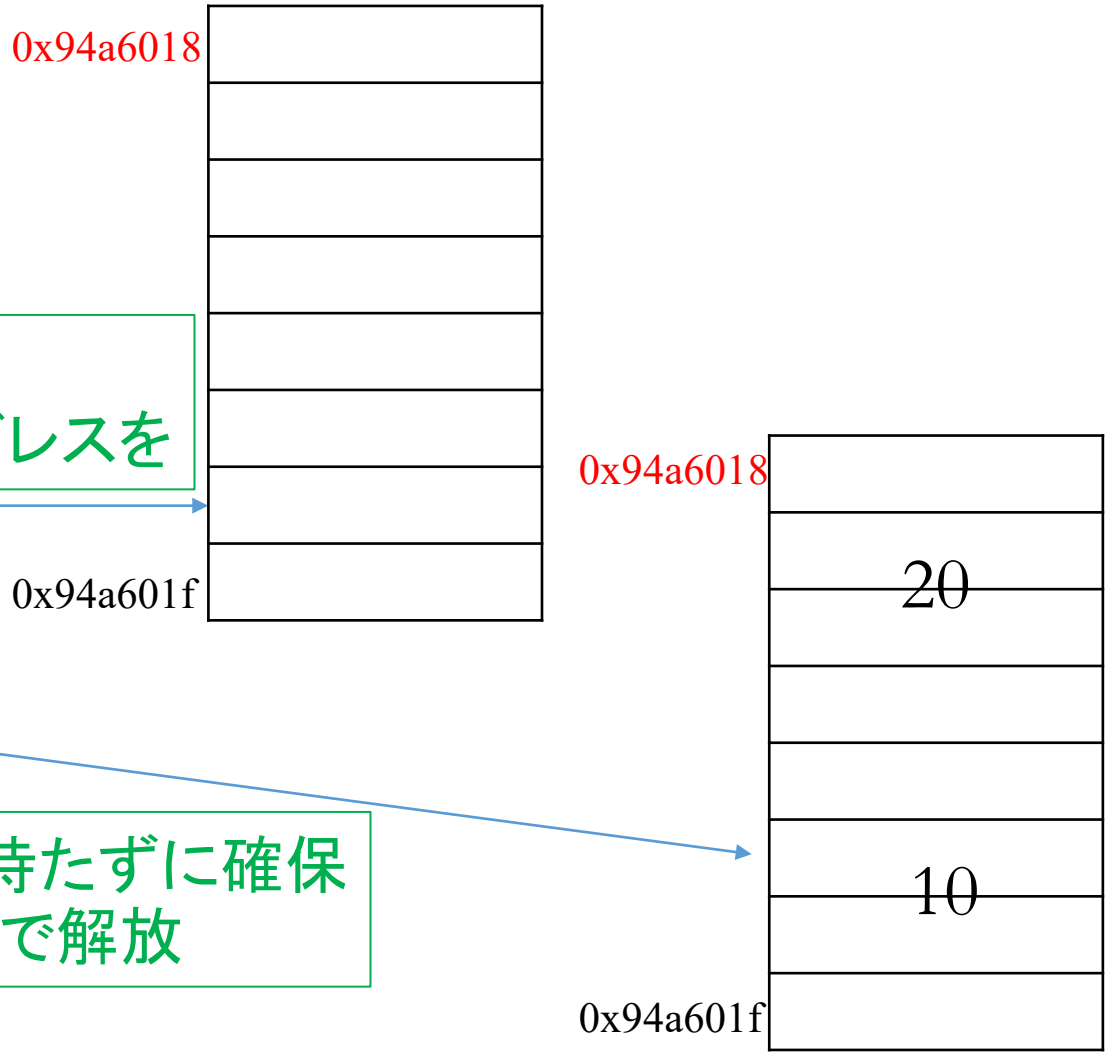


メモリの動的確保（配列の例）

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
  int i, *p;
  p=malloc(sizeof(int)*2);
  printf("Input the values¥n");
  for(i=0; i<2; i++)
    scanf("%d", &p[i]); //p+i
      ⋮
  free(p);
      ⋮
}
```

(1)メモリを確保
(2)pにメモリのアドレスを

プログラムの終了を待たずに確保したメモリをこの時点で解放



manaba小テスト:02-2, 02-3

- 8分，4点
- 12分, 8点