プログラミング第一同演習

慶應義塾大学 理工学部 情報工学科
講義担当: 河野 健二
演習担当: 杉浦 裕太

本日の内容

- 構造体 (struct)
  - 構造体の意味と役割
  - 構造体とポインタ

- 動的 (dynamic) なメモリ割り当て
  - プログラムの実行時に必要に応じてメモリを割り当てる方法
    - プログラムの実行時に何かすることを “動的” (dynamic) という
    - プログラムのコンパイル時に何かすることを “静的” (static) という

構造体 (1)

- 互いに関連した変数をまとめて扱いたい時がある

- 例1：平面上の点の座標を表したいとき
  - int x, y;　　/* ある点の x 座標と y 座標 */

- 例2：学生の成績を表したいとき
  - /* 学籍番号 1 番の学生の国語，数学，英語，理科，社会 の得点 */
    int kokugo_1, sugaku_1, eigo_1, rika_1, shakai_1;

    /* 学籍番号 2 番の学生の得点 */
    int kokugo_2, sugaku_2, eigo_2, rika_2, shakai_2;

構造体 (2)

- 構造体を使うと，2 つ以上の変数をまとめて扱える

- 例：平面上の座標を表す構造体

```
  /* 構造体 point の宣言 */
struct point {
    int x;
    int y;
};
```

← セミコロンを忘れないこと

- 構造体の各要素を構造体のメンバあるいはメンバ変数という
  - x, y は構造体 point のメンバ

構造体 (3)

- 構造体はひとつの型のように扱うことができる

- ```
  /* 構造体 point の宣言 */
  struct point {
      int x;
      int y;
  };

  /* 宣言した struct point は新しい型となる */
  /* 構造体 point 型の変数 p を宣言する     */
  struct point p;
  ```
  ```

構造体 (4)

- struct point p;
  /* この後，どうするの？ */

- メンバ変数に対して読み書きをする

- メンバ変数へのアクセス方法
  - ピリオド (.) を用いてアクセスする
  - p.x = 1;　　/* 構造体 p のメンバ変数 x に 1 を代入 */
    p.y = 2;　　/* 構造体 p のメンバ変数 y に 2 を代入 */
    /* 点 p(1, 2) を定義した気分 */

    printf("p(%d, %d)¥n", p.x, p.y);　　← p(1, 2) と表示される

例題1: 2 点間の距離を求める

- 2 つの点の座標を入力すると，その 2 点間の距離を求めるプログラムを作りなさい

- 実行例:

```
$ ./a.out
Input coordinates A: 5 0     <- 点 A の座標は (5, 0)
Input coordinates B: 5 5     <- 点 B の座標は (5, 5)
Length 5.000000
$
```

例題1: 考え方

- 平面上の点は座標の値の組 (x, y) で表される. 2つの値を別々に扱うのではなく, 一体にして点を表すデータとしてとらえる

- point 構造体を定義する

```
struct point {
    float x_axis;   // X座標
    float y_axis;   // Y座標
};
```

例題1: point 構造体の定義

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

struct point {
   float x_axis;  /* X */
   float y_axis;  /* Y */
};
```

・point構造体を定義

例題1: 変数の宣言

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

struct point {
   float x_axis;  /* X */
   float y_axis;  /* Y */
};


int main()
{
   float x, y, d;          /* 補助変数の宣言  */
   struct point pa, pb;

// continue to next page
```

数学ライブラリを使用するため
ライブラリとは便利な関数を集めたもの

・2 つの頂点を覚えるための変数を宣言

例題1: 座標の読み込み

```
    // continued from previous page

    printf("Input coordinates A: ");
    scanf("%f %f", &(pa.x_axis), &(pa.y_axis));
```

・頂点Aの座標の入力
・構造体のメンバにアクセスするにはピリオド (.) を使う
・この例では pa.x_axis, pa.y_axis は普通の float 型の変数と同じ

```
    printf("Input coordinates B: ");
    scanf("%f %f", &(pb.x_axis), &(pb.y_axis));
```

・頂点B の座標も同様に入力

```
    return 0;
} // end of main()
```

例題1: 長さの計算と表示

```
// continued from previous page

    printf("Input coordinates A: ");
    scanf("%f %f", &(pa.x_axis), &(pa.y_axis));
    printf("Input coordinates B: ");
    scanf("%f %f", &(pb.x_axis), &(pb.y_axis));

    x = pa.x_axis – pb.x_axis;
    y = pa.y_axis – pb.y_axis;
    d = sqrt(x*x + y*y);        ←――― 平方根を返す関数. すでに用意されている


    printf("Length =%f¥n", d);

    return 0;
}
```

構造体の一般的な説明 (1)

- 構造体：1つ以上の要素の集まりからなるデータ型

```
struct タグ名 {
    データ型1 メンバ名1;
    ...
    データ型m メンバ名m;
};  ←――――― セミコロンを忘れないこと
```

  - タグ名：
    - 定義する構造体の名前
  - struct タグ名：
    - “struct タグ名” という新しい型を定義したのと同じ
    - int や char と同じように型として使うことができる

構造体の一般的な説明 (2)

- 構造体と一緒に（その構造体の型の）変数を宣言できる

```
struct タグ名 {
    データ型1 メンバ名1;
    ...
    データ型m メンバ名m;
} 変数名1, … , 変数名n;
```

← セミコロンの位置に注意

  - “struct タグ名” という構造体を宣言している
    - タグ名は省略可．初心者のうちは省略しないほうが安全
  - と同時に，その構造体の変数 変数名$_1$, … , 変数名$_n$ を宣言している

構造体の初期化

- 構造体型の変数を宣言するときのみ，初期化ができる
  - メンバの定義の順に従って，初期化したい値を書く
  - 例：

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

struct point p = {12.8, 34.0};

// 次のように書くのと（ほぼ）同じ
p.x = 12.8;
p.y = 34.0;
```

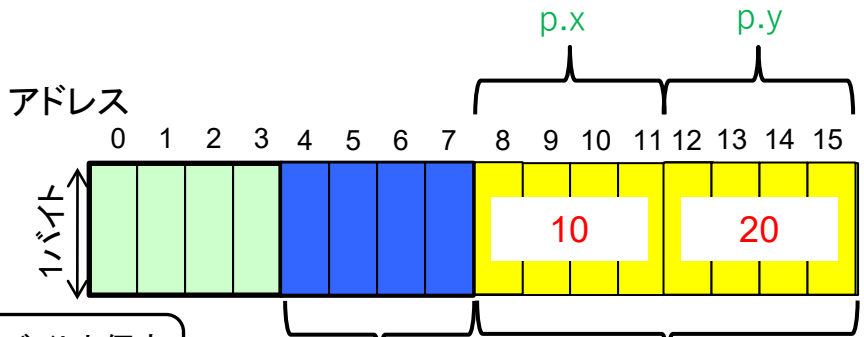
他の初期化方法もある
プロ 1 では省略

構造体とポインタ (1)

- ポインタを使って構造体を参照することができる
  - struct point {
    int x;
    int y;
    };

```
struct point p = {10, 20};     // 変数 p は point 構造体
struct point *q;               // 変数 q は point 構造体を指すポインタ
```

ポインタ変数には
アドレスが入る



構造体のメモリ・イメージ：
各メンバ変数が並んで
確保されている感じ

アドレスは 4 バイトと仮定
（スペース節約のため）

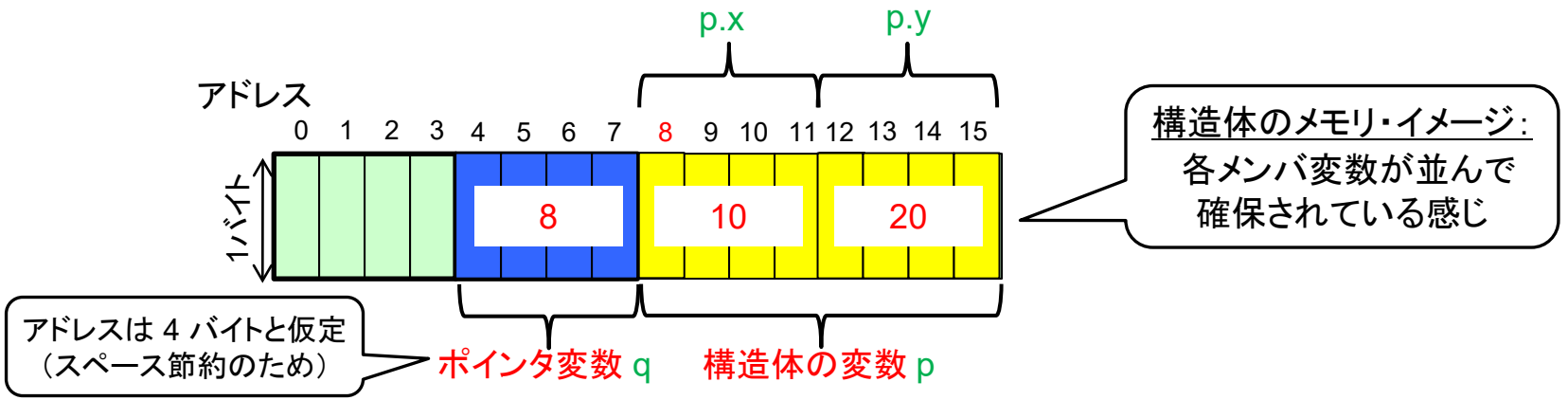
ポインタ変数 q　構造体の変数 p

構造体とポインタ (2)

- 続けて次の代入を行うと・・・
  - struct point p = {10, 20};　// 変数 p は point 構造体
    struct point *q;　// 変数 q は point 構造体を指すポインタ
    ...
    q = &p;

変数 p が格納されている
アドレスを変数 q に代入する

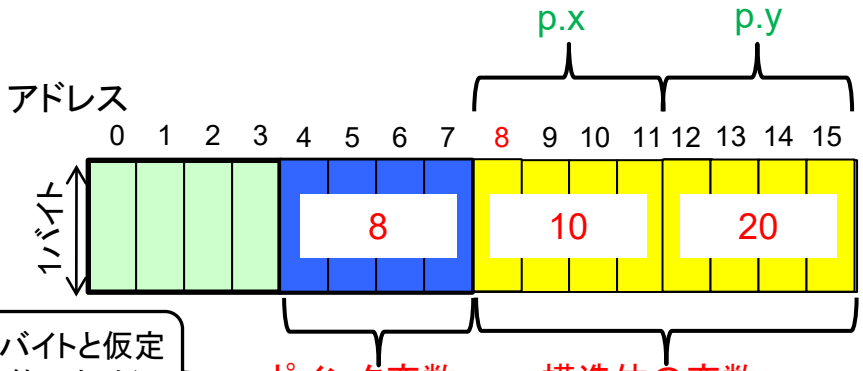
構造体 p はアドレス 8 番地にある
よって q には 8 が入る



構造体とポインタ (3)

- ポインタ変数 q を使って構造体にアクセスしてみる
  - struct point p = {10, 20};　　// 変数 p は point 構造体
    struct point *q;　　// 変数 q は point 構造体を指すポインタ
    ...
    q = &p;
    printf("p.y = %d¥n", (*q).y);

*q でアドレス 8 番地にある構造体を取り出すイメージ.
.y でその構造体のメンバ変数 y を参照する



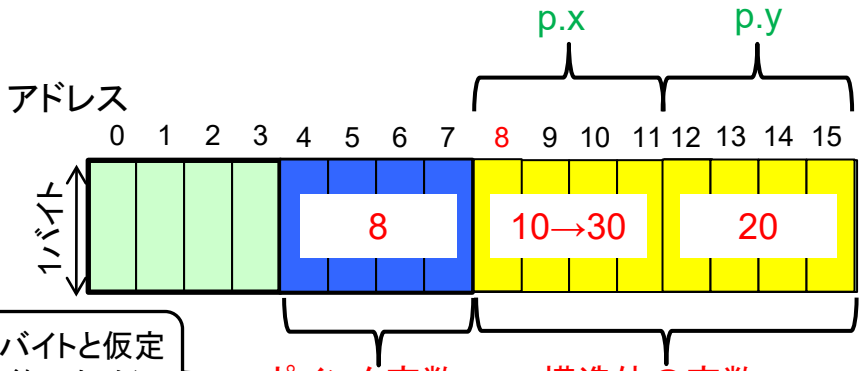
構造体のメモリ・イメージ:
各メンバ変数が並んで確保されている感じ

アドレスは 4 バイトと仮定（スペース節約のため）

構造体とポインタ (4)

- ポインタ変数 q を使って構造体にアクセスしてみる
  - struct point p = {10, 20};　　// 変数 p は point 構造体
    struct point *q;　　// 変数 q は point 構造体を指すポインタ
    ...
    q = &p;
    (*q).x = 30;

*q でアドレス 8 番地にある
構造体を取り出すイメージ.
.x でその構造体の
メンバ変数 x を参照する



構造体のメモリ・イメージ:
各メンバ変数が並んで
確保されている感じ

アドレスは 4 バイトと仮定
（スペース節約のため）

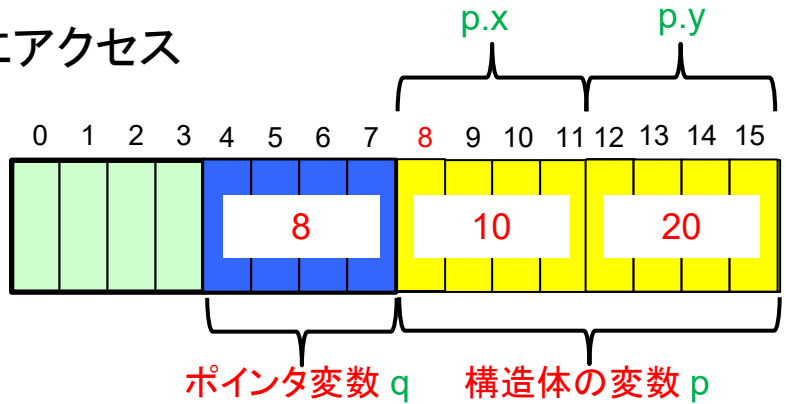
-> 演算子

- (*q).x という形はよく出てくる
  - 構造体へのポインタ q を使ってメンバ変数 x にアクセス

- q->x と書いてもよい（こっちが普通）
  - 変数 q は構造体を指すポインタ
  - そのポインタの指す構造体のメンバ変数 x にアクセス

同じこと

  - 変数 q にはアドレスが入っている
  - そのアドレスには構造体が格納されている
  - その構造体のメンバ変数 x にアクセス



- 例: printf("p.x = %d, p.y = %d¥n", q->x, q->y);

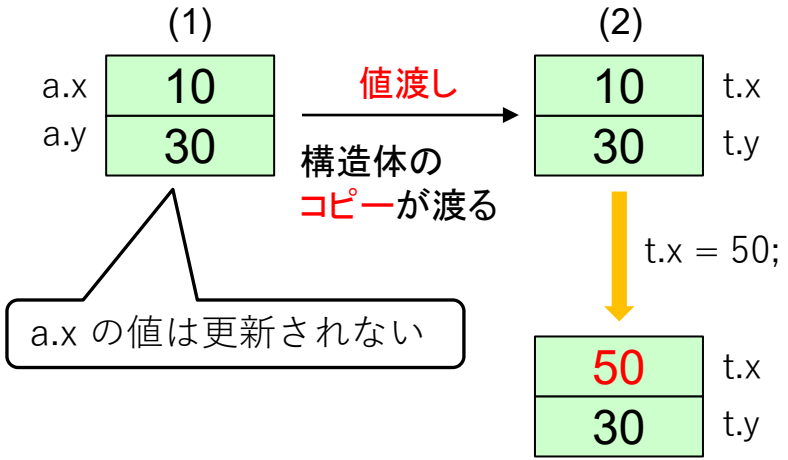
関数の引数としての構造体

```
struct tab {
    int x;
    int y;
} a ={10, 30};

void func(struct tab t)          ——— (2)
{
    t.x =50;
}

int main()
{
    ...
    func(a);          ——— (1)
    ...
}
```

C 言語では，引数は必ず
call-by-value（値渡し）で受け渡される



関数の引数としての構造体ポインタ

```
struct tab {
    int x;
    int y;
} a ={10, 30};

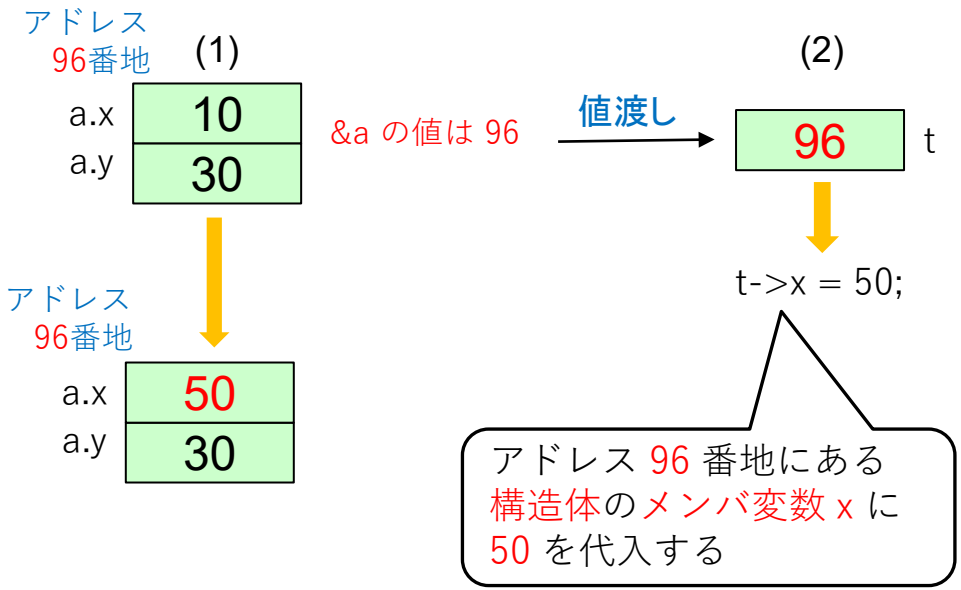
void func(struct tab *t)          ―― (2)
{
    t->x =50;
}

int main()
{
    ...
    func(&a);                     ―― (1)
    ...
}
```

ポインタを受け取るようにする

変数 a のアドレスを渡すようにする

C 言語では，引数は必ず
call-by-value（値渡し）で受け渡される



Quiz: プログラムの実行結果は？

```
struct s {
    int x;
    float y;
};

int main()
{
    struct s a;
    struct s *b = &a;

    a.x = 10;
    b->y = 2.0;

    printf("%d", b->x);
    printf("%f", a.y);

    return 0;
}
```

```
struct t {
   int x;
   int y;
} a = {1, 2};

void
foo(struct t p)
{
    p.x = 10;
    p.y = 20;
}

int main()
{
    foo(a);
    printf("%d", a.x);
    printf("%d", a.y);
    return 0;
}
```

Quiz: プログラムの実行結果は？

```
struct t {
   int x;
   int y;
} a = {1, 2};

void foo(struct t *p)
{
    p->x = 10;
    p->y = 20;
}

int main()
{
    foo(&a);
    printf("%d", a.x);
    printf("%d", a.y);
    return 0;
}
```

```
struct t {
   int x;
   int y;
} a[1];

void foo(struct t p[])
{
    p[0].x = 10;
    p[0].y = 20;
}
int main()
{
    a[0].x = 1;
    a[0].y = 2;
    foo(a);
    printf("%d", a.x);
    printf("%d", a.y);
    return 0;
}
```

動的なメモリの割り当て

特別なポインタ: NULL

- ポインタが指し示すものがないという意味
  - NULL の実体はアドレス 0 番地
  - 慣習として，アドレス 0 番地に変数等を置くことはない
  - したがって，アドレス 0 番地を指すということは，何も指さないという意味になる

- NULL ポインタを参照すると実行時エラーとなる
  - NULL pointer dereference とか，ヌルポインタ参照などという

```
int *p = NULL;

// p は NULL ポインタ
*p = 100;    ←―― エラー！
```

動的メモリ割り当て

- これまでは，ポインタはすでに存在する変数を指し示していた
  - int x;　// 変数 x がメモリ上に割り当てられる
    int *p;
    p = &x; // ポインタ変数 p には（すでに存在する）変数 x のアドレスが入る

- 変数を宣言せずにメモリを確保したいこともある
  - メモリを アロケート (allocate) するともいう
  - これを動的メモリ割り当てという

- 本当の使い方は「アルゴリズム」で学ぶ
  - リスト構造（次回，紹介する），ツリー構造など
  - 実用プログラム内では，動的メモリ割り当ては頻繁に行われている

malloc(): メモリ領域の動的な割り当て

- 書式

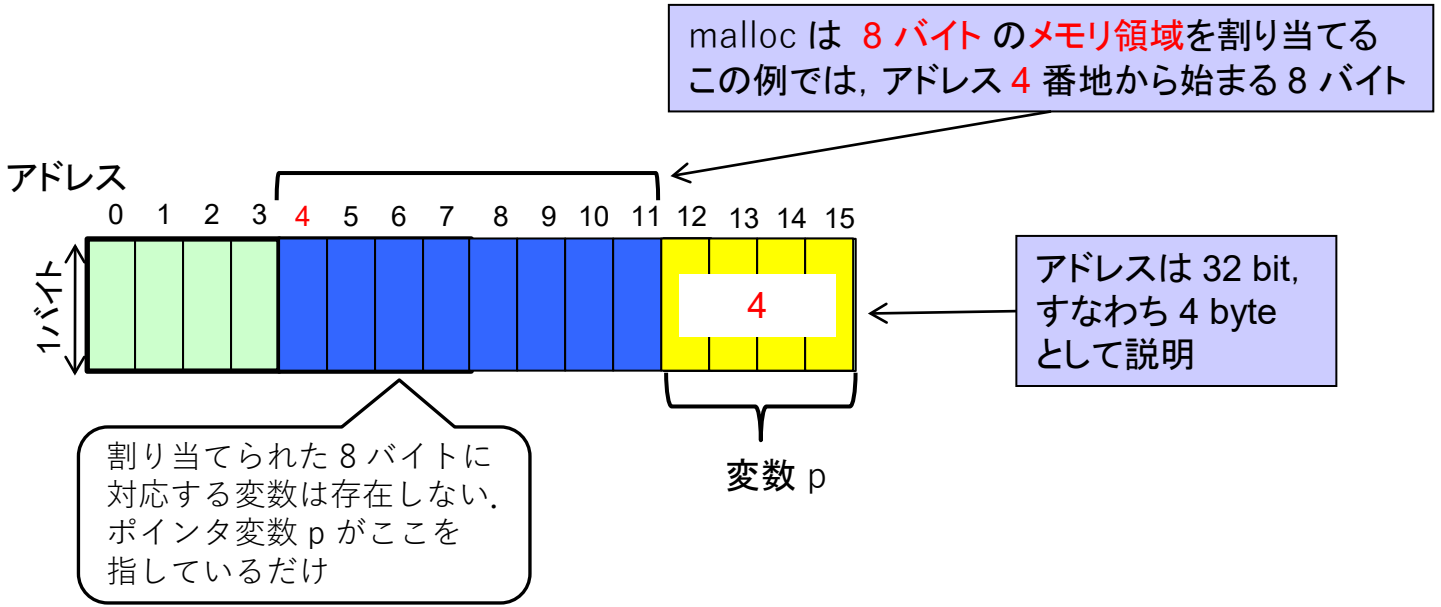
```
#include <stdlib.h>          ←――― malloc を使うときは必須

/* malloc のプロトタイプ宣言 */
void *malloc(size_t size);
```

- 引数: 割り当てるメモリの大きさ（バイト数）を指定する
  - malloc は指定されたバイト数のメモリ領域を割り当てる
- 戻り値:その領域へのポインタ（割り当てたメモリ領域の先頭アドレス）を返す
  - 戻り値の型である void * の意味は後述

malloc(): 呼び出したときの動作

- 例：8 バイトのメモリ領域を確保するように依頼する
  - p = malloc(8);
    // 割り当てられたメモリ領域の先頭アドレスを返す
    // この例では，アドレス 4 番地が返されている



malloc(): 実際の使い方 (1/3)

- 例：構造体 point を動的に割り当てる

```
struct point {
    float x_axis;   // X座標
    float y_axis;   // Y座標
};

struct point *p;
p = malloc( ここに point 構造体の占めるバイト数を指定する );
```

どうやって知るの？ 自分で計算？？？

sizeof

- ある型の変数がメモリ上で占めるバイト数を返す

- 例:

```
char a;   // char は 1 バイト
int b;    // int は 4 バイト
sizeof(char)     → 1     型名を与えてもよい
sizeof(int)   → 4
sizeof(a)        → 1     変数名を与えてもよい
sizeof(b)        → 4
```

- ただし，配列に適用する場合は注意が必要
  - 少し先の講義で教える

malloc(): 実際の使い方 (2/3)

- 例：構造体 point を動的に割り当てる

```
struct point {
    float x_axis;   // X座標
    float y_axis;   // Y座標
};

struct point *p;
p = malloc( sizeof(struct point) );
```

sizeof() **を使って** struct point 型の構造体の占めるバイト数を得る

sizeof(*p) と書いてもよい．(sizeof(p) は間違いなので注意)

変数 p はポインタ型．よって sizeof(p) で得られるのは，アドレスを格納するのに必要なバイト数となる

malloc(): 実際の使い方 (3/3)

- 例：構造体 point を動的に割り当てる
  - malloc の戻り値を，正しい型のポインタになるようにキャストする

```
struct point {
    float x_axis;   // X座標
    float y_axis;   // Y座標
};

struct point *p;
p = (struct point *)malloc(sizeof(struct point));
```

必ずキャストする.
malloc の戻り値の型は void *
変数 p の型は struct point *
キャストしないと型が合わないと怒られる

malloc(): void * 型の意味

- int * 型の変数には・・・
  - アドレスが入っていて，
    そのアドレスには int 型の値が置かれている
- void * 型の変数には・・・
  - アドレスが入っていて，
    そのアドレスには void 型の値が置かれている???

意味不明！
void 型は戻り値がないことを表していた・・・

- void * 型の意味：
  - アドレスが入っているが，
    そのアドレスに置かれている値の型がわからないという意味

free(): 割り当てた領域を解放する

- 書式

```
#include <stdlib.h>          ← free を使うときは必須

void free(void *ptr);
```

- malloc で割り当てたメモリ領域を解放する
  - 引数：malloc で割り当てたメモリ領域を指すポインタ

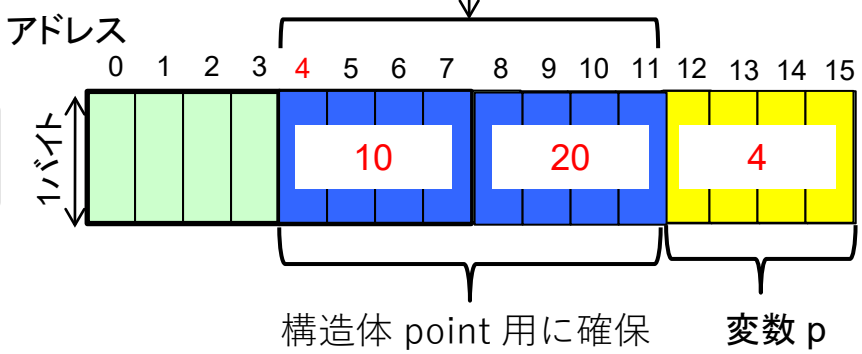
- free を呼び出すまで，割り当てた領域はメモリ上に確保され続ける
- 不要になったら free を呼び出して解放する
  - 解放しないと不要になったメモリ領域を再利用できない

free(): 呼び出すと・・・

```
struct point *p;
p = (struct point*)malloc(sizeof(struct point));
p->x = 10;    // p は point 構造体を指すポインタだと思って使える
p->y = 20;
...
free(p);
p->30;
```

これは間違い.
p の指す先は解放済み
したがって，使ってはいけない

free(p) によって，アドレス 4 番地から始まる 8 バイトが解放される



その結果，以降の malloc でこの 8 バイトが割り当て可能になる

malloc(): 再び

- 書式

```
#include <stdlib.h>          ←――― malloc を使うときは必須

/* malloc のプロトタイプ宣言 */
void *malloc(size_t size);
```

- 引数: 割り当てるメモリの大きさ（バイト数）を指定する
  - malloc は指定されたバイト数のメモリ領域を割り当てる
- 戻り値:その領域へのポインタ（割り当てたメモリ領域の先頭アドレス）を返す
  - メモリ割り当てに失敗すると，NULL を返す
    （メモリ不足などでメモリ割り当てに失敗することもある）

malloc: エラーチェックをすること！

- 稀ではあるが，malloc が NULL を返すことがある

```
struct point {
    float x_axis;   // X座標
    float y_axis;   // Y座標
};

struct point *p;
p = (struct point *)malloc(sizeof(struct point));
if (p == NULL) {
                エラー処理 (エラーメッセージを表示して，exit(1) することが多い)
}
```

  - とりあえず，次のように書いておく（もっといいやり方は先の話）
    - if (p == NULL) {
      printf("out of memory¥n");
      exit(1);
      }

注意：その 1

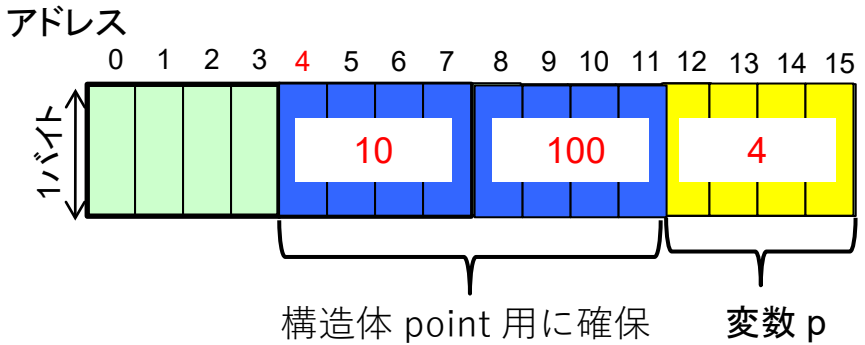
- メモリの解放を忘れないようにしよう
  - malloc で確保した領域は，不要になったら free する
  - free し忘れると，メモリの無駄使いになる
    - メモリリーク (memory leak) というバグになる
    - 最悪の場合，メモリ不足でアプリが停止する
- 長時間動き続けるシステムでは致命的になりかねない
  - 実際の事故の事例も多い

注意: その 2

- 確保した領域が迷子にならないようにしよう

```
void foo() {
    struct point *p;
    p = (struct point *)malloc(sizeof(struct point));
    if (p == NULL) { … }
    p->x = 10;
    p->y = 100;
}
```

関数 foo の実行が終了したあと，
malloc された領域を指すポインタが
存在しない



注意: その 3

- malloc で確保していないメモリは使えない

```
void foo() {
    struct point *p;
    p->x = 10;
    p->y = 100;
    ...
}
```

間違い．ポインタ変数 p は
確保されたメモリを指していない

- free で開放したメモリは使えない

```
void foo() {
    struct point *p;
    p = (struct point *)malloc(sizeof(struct point));
    ...
    free(p);
    p->x = 10;
    ...
}
```

間違い．ポインタ変数 p の
指す先は開放済み

【余談】Garbage Collection

- 日本語に訳すと「ごみ集め」
- C では自分で free を呼ぶ必要がある
- Python や Java では free を呼ぶ必要がない
  - 不要になったメモリ領域を自動で解放してくれる
  - この機能を garbage collection という
    - 不要になったメモリ領域を「ごみ」と言っている

まとめ

- 構造体
  - 定義の仕方，使い方
  - -> の使い方に早めに慣れること

- 動的メモリ管理
  - malloc/free のやり方だけ学んだ
  - 役に立つ使い方は次回学ぶ
    - 「アルゴリズム」の授業で学ぶ内容