ソフトウェア2
[第3回]
(2020/12/03)

齋藤 大輔

講義用ページ

- URL

  https://www.gavo.t.u-tokyo.ac.jp/~dsk_saito/lecture/software2

- 適宜ページを参照すること

成績評価

- 全6回のレポート課題により評価
  - 基本課題については全て採点対象
  - 発展課題については全6回中上位3回分を採用
- レポート期限は締切日の23:59:59 （日本時間）

本日のメニュー

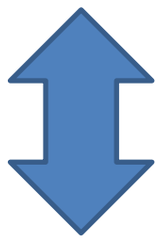
- C言語入門編
  - 動的メモリ確保（malloc, free）
- データ構造
  - 線形リスト
- 今日の題材
  - コマンドライン入力によるペイントソフト
- レポート課題について

本日のメニュー

- C言語入門編
  - 動的メモリ確保（malloc, free）
- データ構造
  - 線形リスト
- 今日の題材
  - コマンドライン入力によるペイントソフト
- レポート課題について

動的メモリ確保

- 配列
  - サイズを実行時に決められない
    - 注）C99での可変長配列は可能だが、宣言時に確定
  - スコープを抜けると解放される



- malloc による動的メモリ確保
  - サイズを実行時に指定
  - 明示的に解放（free）されない限り、プログラムの実行が終わるまでメモリ領域が確保されている

配列と malloc

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

int *func()
{
  int a[3];
  a[0] = 1;
  a[1] = 2;
  a[2] = 3;

  return a;
}

int main()
{
  int *b = func();

  printf("%d\n", b[0]);
  printf("%d\n", b[1]);
  printf("%d\n", b[2]);
}
```

← int へのポインタを返す関数

ダメ

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

int *func()
{
  int *a = (int *)malloc(3 * sizeof(int));
  a[0] = 1;
  a[1] = 2;
  a[2] = 3;

  return a;
}

int main()
{
  int *b = func();

  printf("%d\n", b[0]);
  printf("%d\n", b[1]);
  printf("%d\n", b[2]);
}
```

OK

malloc() 関数

- メモリの動的確保

```
int *ptr = (int *)malloc(10 * sizeof(int));
```

  - int 10個分のメモリ（40バイト）をヒープ領域に確保
  - 確保したメモリの先頭アドレスを返す
  - 確保に失敗した場合は NULL を返す
  - 確保した領域の値は初期化されていない

- メモリの解放

```
free(ptr);
```

  - そのメモリが不要になったら free 関数で解放する
  - 解放し忘れると…. → メモリリーク
  - ポインタへNULL代入は領域解放ではない！

malloc() 関数

- 使い方の例

```
int *ptr = (int *)malloc(10 * sizeof(int));

if (ptr == NULL) {
  exit(1); // メモリ確保に失敗
}

ptr[0] = 123;
ptr[1] = 555;

free(ptr);
```

- ポインタを利用して配列と似たように使える
  - ただしsizeofを利用した要素数取得はできない
    - `printf("%lu\n",sizeof(ptr)/sizeof(ptr[0]));` の結果は？

malloc() 関数

- 構造体にも使える
  - 構造体に対してはこの使い方が多い

```
// point 構造体の定義は一旦省略

struct point *init_point(int x, int y)
{
  struct point *ptr = malloc(sizeof(struct point));

  ptr->x = x;
  ptr->y = y;

  return ptr;
}

int main()
{
  struct point *ptr = init_point(10, 20);

  printf("%d\n", ptr->x);
  printf("%d\n", ptr->y);
}
```

malloc() 関数

- 初期化指示子、複合リテラルを利用すると…

```
// point 構造体の定義は一旦省略


struct point *init_point(int x, int y)
{
  struct point *ptr = malloc(sizeof(struct point));
  // この場合は最初の*を忘れないこと!!
  *ptr = (struct point){.x = x, .y = y};
  return ptr;
}

int main()
{
  struct point *ptr = init_point(10, 20);

  printf("%d\n", ptr->x);
  printf("%d\n", ptr->y);
}
```

初期化

- malloc()で割り当てられた領域の値は不定

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

int main()
{
  int *ptr = (int *)malloc(10 * sizeof(int));

  if (ptr == NULL) {
    exit(1); // メモリ確保に失敗

  }
  // 初期化せずに表示すると…
  for (int i = 0 ; i < 10 ; i++)
    printf("%d: %d\n",i,ptr[i]);

  free(ptr);
}
```

初期化

- 0で初期化したい場合calloc()を用いる

```
ptr = calloc(size_t count, size_t size);
```

free()関数

- malloc()で割り当てられたヒープ領域を解放する
  - 静的配列（スタック領域）および代入前のポインタは絶対にfreeしないこと
  - NULLの解放は基本的に問題ない
  - 二重解放（同じアドレスを2回free）はトラブルの元なので、解放後にNULL代入はあり

```
int *a = (int *)malloc(10 * sizeof(int));
int *b;
int c[10];

free(a); // OK
free(b); // NG
free(c); // NG
free(NULL); // OK
```

メモリリークに注意

- 確保された領域のアドレスを忘れないこと
  - 以下のコードはmallocされた領域へのアクセス法がない

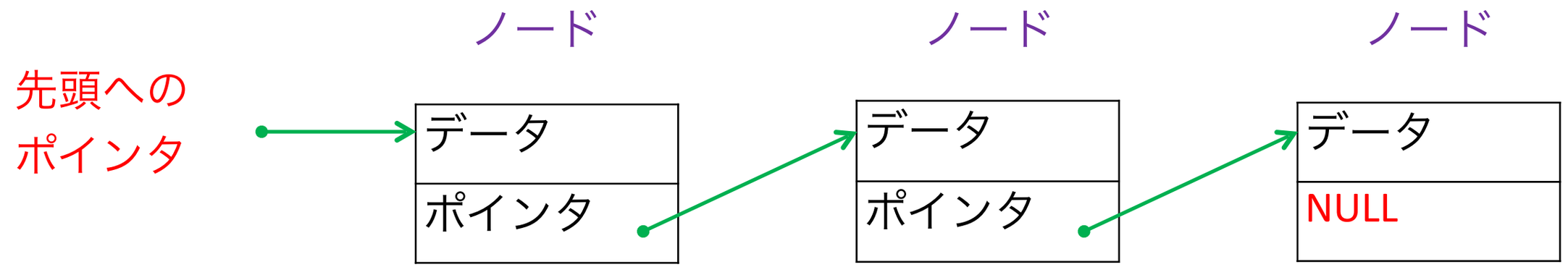
```
int number = 10;
int *a = (int *)malloc(10 * sizeof(int));
a = &number; // ここで前のアドレス情報を失う

// 以後最初にmallocされた領域へのアクセスはできない
```

本日のメニュー

- C言語入門編
  - 動的メモリ確保（malloc, free）
- **データ構造**
  - **線形リスト**
- 今日の題材
  - コマンドライン入力によるペイントソフト
- レポート課題について

線形リスト（linear list）

- 多数のデータを格納するためのデータ構造のひとつ
- 格納する各要素をポインタで連結（連結リスト）

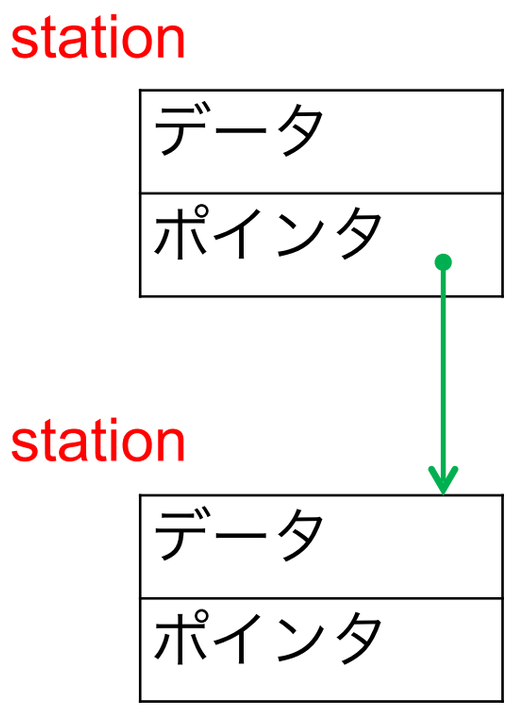


- 要素の削除、挿入のコストが小さい
  - 前後のポインタをつなぎかえるだけ
- 要素数の制限がない
  - ひとつ増えるたびに malloc すればよい
- ランダムアクセスのコストは高い

線形リストの実装法

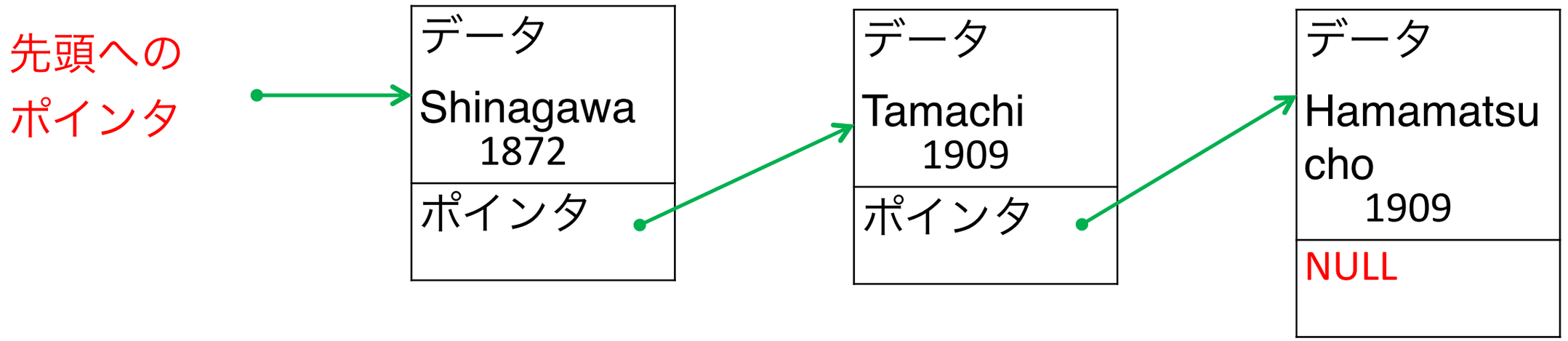
- 自己参照構造体を使う
  - 自己参照構造体： 自分と同じ構造体を指すポインタをメンバに持つ構造体

```
struct station
{
        char name[20];
        int birthyear;
        struct station *next;
};
```



線形リスト（linear list）

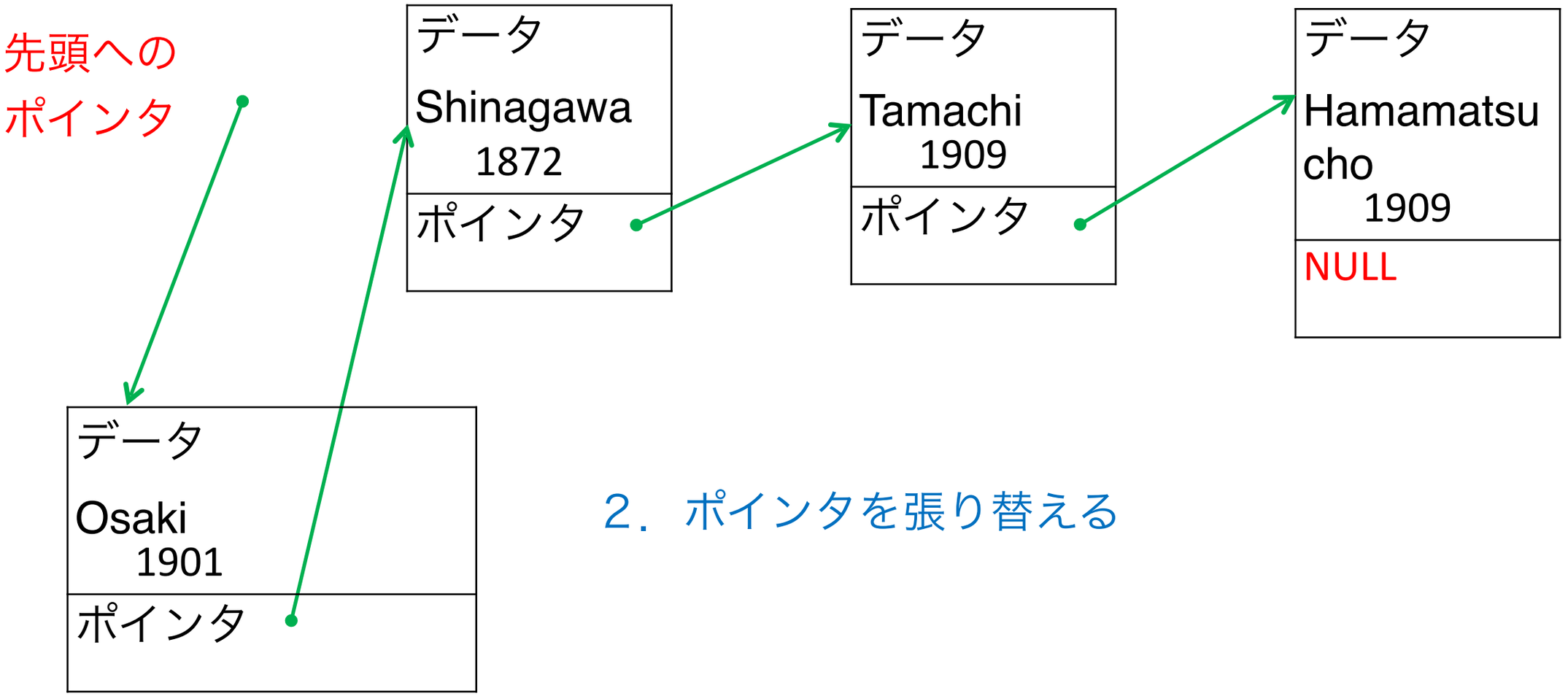
- 要素の挿入（先頭に挿入する場合）



1．挿入したいデータをヒープメモリにコピー
（malloc して領域確保）

線形リスト（linear list）

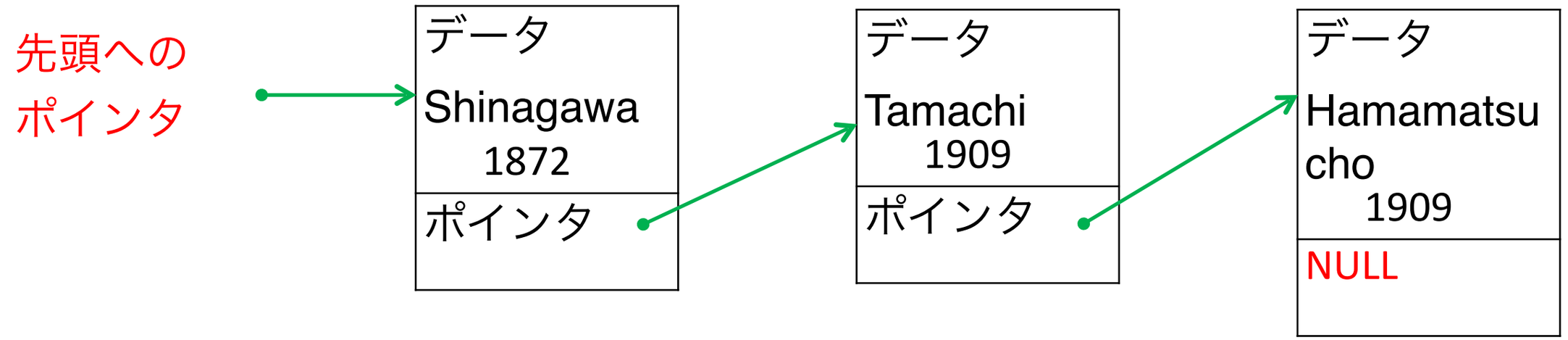
- 要素の挿入（先頭に挿入する場合）

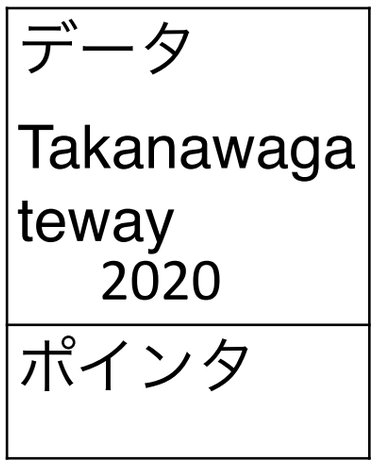


2．ポインタを張り替える

線形リスト（linear list）

- 要素の挿入（途中に挿入する場合）

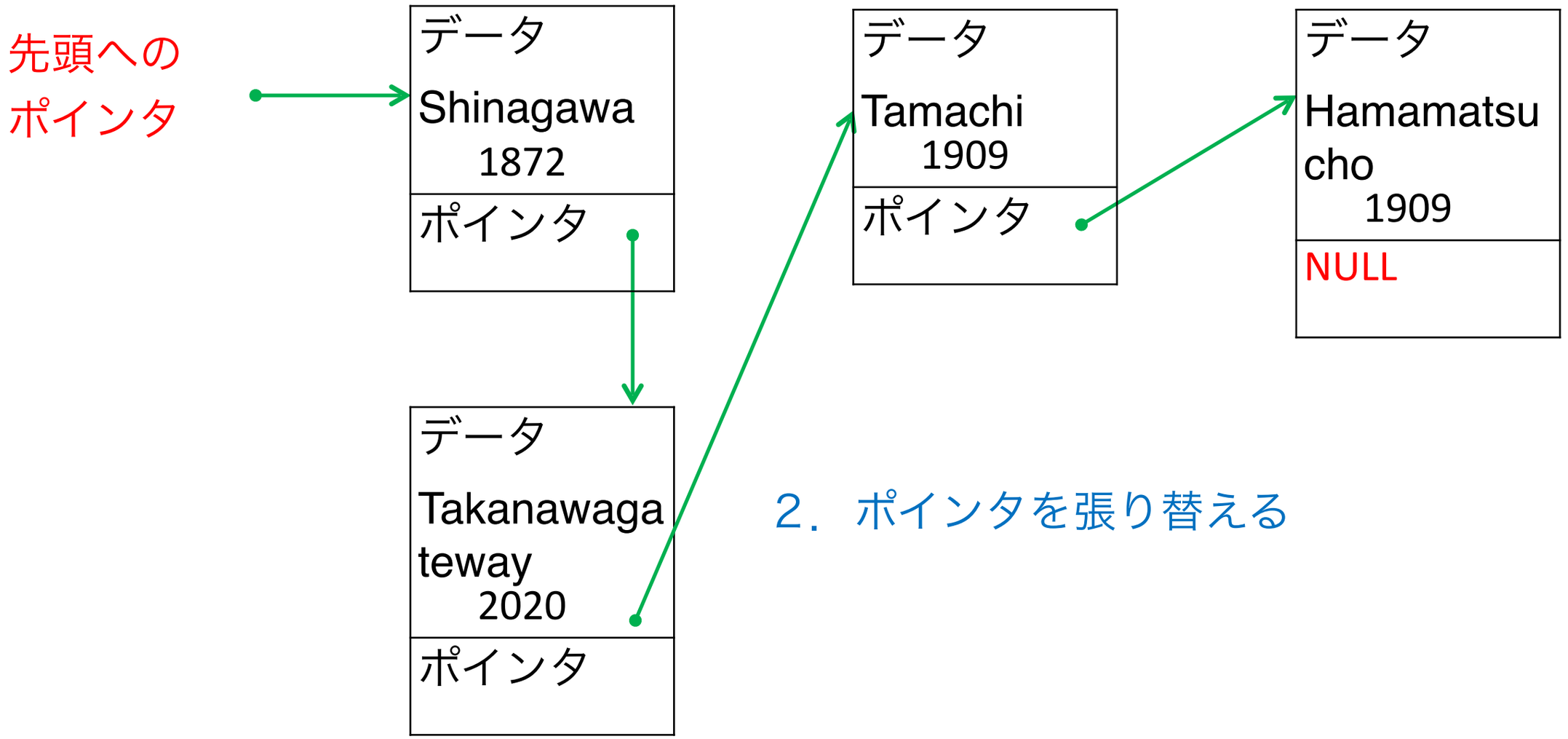




1．挿入したいデータをヒープメモリにコピー（mallocして領域確保）

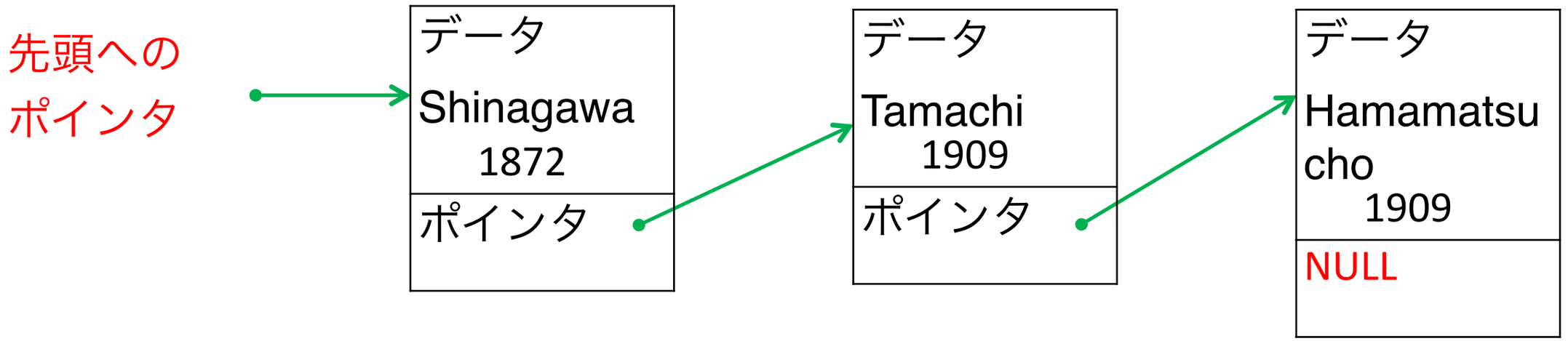
線形リスト（linear list）

- 要素の挿入（途中に挿入する場合）



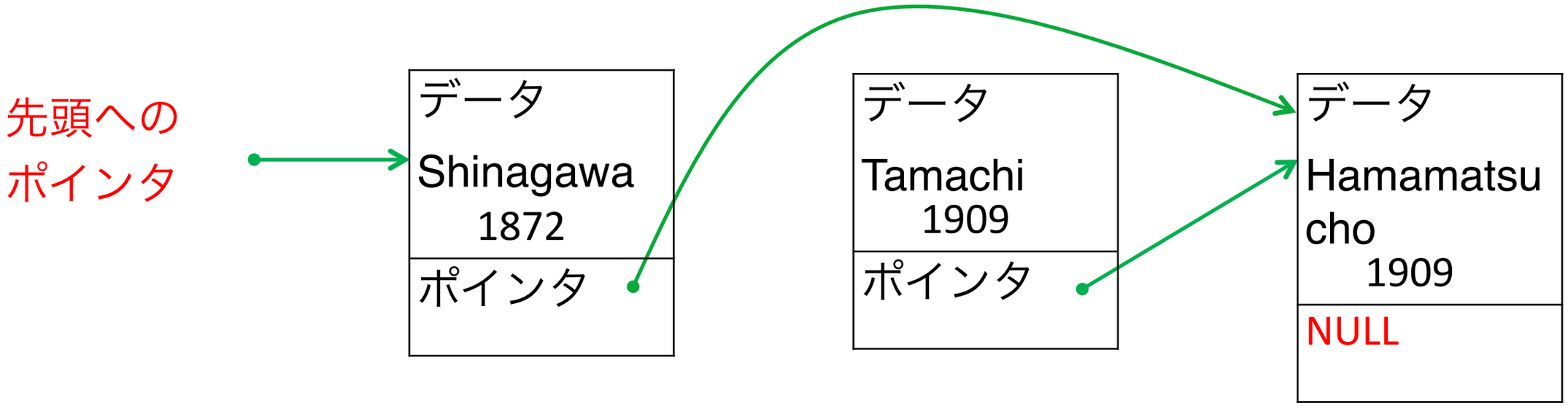
線形リスト（linear list）

- 要素の削除の例



線形リスト（linear list）

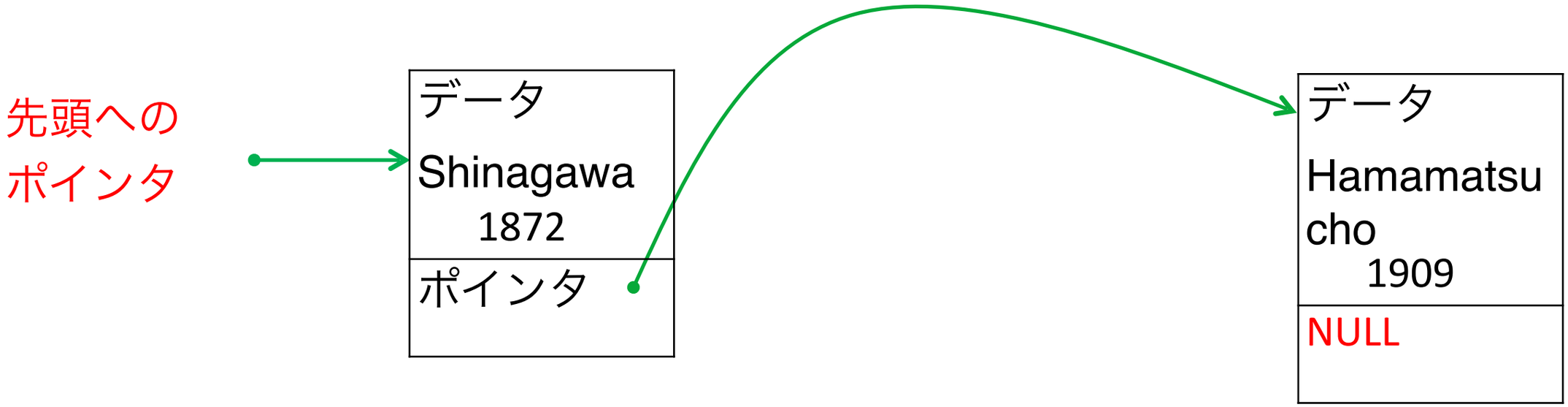
- 要素の削除の例



1．ポインタを張り替える

線形リスト（linear list）

- 要素の削除の例



2．削除するデータのメモリを解放

簡単な線形リストの実装例

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

struct student
{
  int id;
  struct student *next;
};

typedef struct student Student;

Student *push_front(Student *p, int id)
{
  Student *q = malloc(sizeof(Student));

  *q = (Student){.id = id, .next = p};

  return q;
}
```

```c
int main()
{
  Student *begin = NULL;
  begin = push_front(begin, 1);
  begin = push_front(begin, 2);
  begin = push_front(begin, 3);

  for (Student *p=begin; p != NULL; p = p->next){
    printf("%d\n", p->id);
  }

  return 0;
}
```

実行結果

```
$ ./a.out
3
2
1
```

サンプルプログラム list.c

- 処理内容
  - 標準入力から１行ずつ読み込み、文字列を線形リストに保存
  - 線形リストを先頭から順にたどり、文字列を標準出力に書きだす
  - 実行例
  - `% ./a.out < yamanote.txt`
  - 線形リストではなく配列で同じ処理を実装したらどうなるか？

- 線形リストの操作
  - begin: 先頭ノードへのポインタ
    - 「リスト」としてこのポインタを要素にする構造体を定義することも多い（NULLの場合に安全）
  - push_front() 関数: 先頭に要素を追加
  - push_back() 関数: 末尾に要素を追加
  - pop_front() 関数: 先頭の要素を削除

list.c 冒頭

- 自己参照構造体の宣言

```
typedef struct node Node;

struct node
{
   char *str;
   Node *next;
};
```

- データは文字（列）へのポインタのみ
- 文字列そのものは保持しないことに注意
  - node生成時にmalloc と strcpy でセットする

push_front() 関数

- 先頭に要素を挿入
  - 入力：リストの先頭へのポインタ、格納する文字列
  - 出力：（挿入後の）リストの先頭へのポインタ
  - malloc で、ノードおよび文字列のメモリを確保し、データ（文字列）を保存した後、ポインタを張り替える

```
Node *push_front(Node *begin, const char *str)
{
  Node *p = (Node *)malloc(sizeof(Node));
  char *s = (char *)malloc(strlen(str) + 1);
  strcpy(s, str);
  p->str = s;
  p->next = begin;

  return p;
}
```

strlenでは末端の'\0'がカウントされないので

pop_front() 関数

- 先頭の要素を削除
  - 入力：リストの先頭へのポインタ
  - 出力：（削除後の）リストの先頭へのポインタ
  - freeで、ノードおよび文字列のメモリを解放
  - ２番目だったノードが新たな先頭ノードになる

```
Node *pop_front(Node *begin)
{
  Node *p = begin->next;

  free(begin->str);
  free(begin);

  return p;
}
```

push_back() 関数

- 末尾に要素を追加
  - 末尾の要素に行きつくまでリストを先頭からたどり、その後ろに新たな要素を追加
  - ただし、リストが空のときは特別扱い

```
Node *push_back(Node *begin, const char *str)
{
  if (begin == NULL) return push_front(begin, str);

  Node *p = begin;
  while (p->next != NULL) {
    p = p->next;
  }

   :  // 末尾に要素を追加
}
```

実習1

- list.c に末尾の要素を削除する `pop_back()` 関数を追加せよ
  - 計算量は O(n) で構わない　リスト中の要素数 n に比例する計算量という意味
  - 新たに末尾となる要素の処理に注意
  - `main`関数で使う関数を変えてテストしてみる
- 先頭へのNode構造体へのポインタをメンバに持つ新たな構造体List を定義し、前述の関数をこれを操作する関数に書き換えよ。

方針

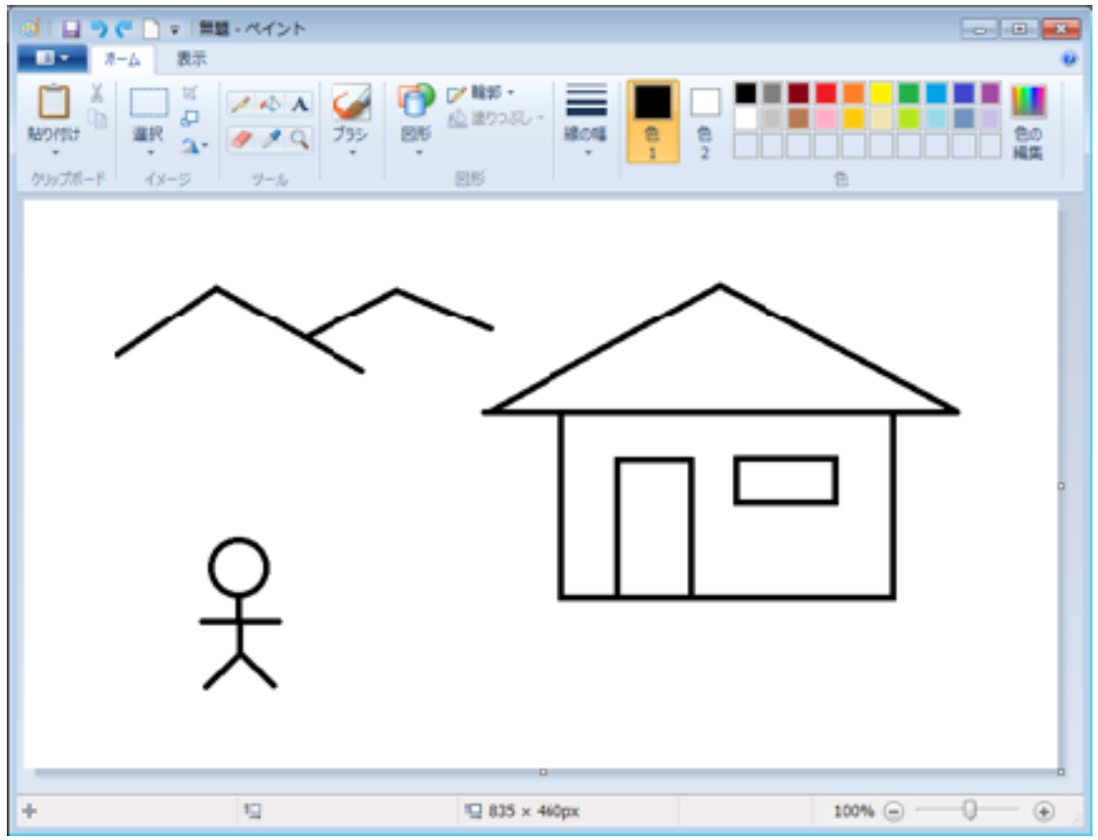
- まずはlist.c を読んでみよう
- 読んでわからない場合にlist_comment.c を読んで解釈を勉強してみよう
- この実習をやっておかないと課題はつらくなる

本日のメニュー

- C言語入門編
  - 動的メモリ確保（malloc, free）
- データ構造
  - 線形リスト
- **今日の題材**
  - **コマンドライン入力によるペイントソフト**
- レポート課題について

ペイントソフト

- コマンド入力方式で絵を描く
  - 描画機能
  - 線を描く
  - 長方形を描く
  - 円を描く
  - ：
- Undo
  - 直前のコマンドの取り消し
- 履歴の保存



サンプルプログラム paint.c

- コンパイル＆実行

```
% gcc -o paint paint.c
% ./paint 80 40
0 > line 10 10 20 10
1 > save
2 > quit
```

w = 80, h = 40

(10, 10) から (20, 10) まで線を引く

コマンド履歴の保存

終了

- ターミナルを適度に大きくする（キャンバス用）
- Ctrl + L を押す（ターミナルリセット）
- 実行する

```
%./paint 80 40
```

interpret_command() 関数

- コマンド文字列をトークン列に分解して最初の単語を取り出す
  - strtok関数で最初のトークンを取得
    - デリミタ（区切り文字）は空白文字
    - 文字列が破壊されるのでコピーしたものを渡す

```
int interpret_command(const char *command, int *hsize, Canvas *c)
{
  char buf[BUFSIZE];
  strcpy(buf, command);

  char *s = strtok(buf, " ");
```

draw_line() 関数

- 線を引く
  - 始点と終点をn等分して点を打つ
  - nは点と点の隙間が空かないように決める

```
void draw_line(Canvas *c, int x0, int y0, int x1, int y1)
{
  // 初期化中略

  const int n = max(abs(x1 - x0), abs(y1 - y0));
  c->canvas[x0][y0] = pen; // 0割算防止

  for (int i = 1; i <= n; i++) {
    const int x = x0 + i * (x1 - x0) / n;
    const int y = y0 + i * (y1 - y0) / n;
    canvas[x][y] = '#';
  }
}
```

Undo について

- ひとつ前のコマンドを取り消す
  - キャンバスを初期化し、最初のコマンドから直前のコマンドまでを実行しなおす
  - コマンドの履歴の長さを１減らす
    - ひとつ前のコマンドのもの

実習2

- paint_arrayhistory.c をもとに履歴の管理を線形リストで再実装せよ（paint.c とする）。
  - list.cを応用し、メモリの無駄、コマンド履歴長の制限がなくなるように
- ヒント
  - History 構造体はコマンドを表す構造体（線形リストの先頭）を指すようにする
  - 適宜関数を書き換え、追加していく
- 適宜、paint_arrayhistory_comment.c を読む

レポート課題（締切12/09）

1. paint.c に長方形を描くコマンドと円を描くコマンドを追加せよ。
2. ファイルに保存されたコマンド履歴を読み込み絵を再描画するコマンド “load” を追加せよ
3. 描画の文字の種類を変更する機能を追加せよ。
4. ［発展課題］paint.c に、他に有用なコマンド（塗りつぶし、エフェクトをかける、コピー＆ペースト、BMP形式で保存、など）を追加せよ

＊課題の実装にあたり、実習2に相当する「コマンド履歴の線形リストによる実装」を行なっていることを前提とする

コマンドの仕様等については、サイトの課題記述を確認すること

課題の提出方法（ITC-LMS）

- 形式: ファイルアップロード
  - 全てのプログラム/ファイルをまとめ、zipやtar.gzで圧縮
  - git archive コマンドやzipコマンド等を用いる
  - SOFT-12-03-NNNNNNNN.zip または
    SOFT-12-03-NNNNNNNN.tar.gz
    - NNNNNNNN 部分は学籍番号（ハイフン除く）
    - JについてはJ??????? のようにしてください。
- 課題について
  - [基本課題] 毎回提出
  - [発展課題] 成績計算に全６回中上位３回分を採用する