計算機序論2

2010/11/01
(2010/10/25の予定だったもの)

亀田能成

スクリプト読み込みプログラム

- 膨大な仕事を一気にこなすのは大変
- 仕事を分割して(＝複数の関数)で処理

規模の大きいプログラム

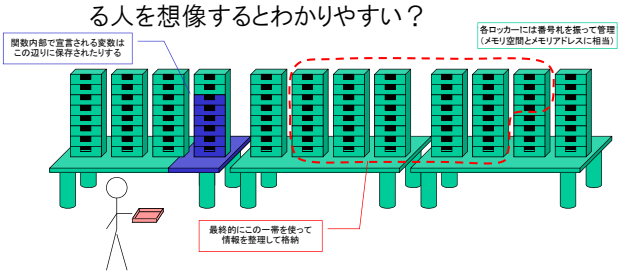
- 仕事だって大きくなれば一気にはできない
  - 作業を分割する
  - 分割した作業を人に任せる
  - 作業内容はとやかく詮索せずに結果だけ受領
- C言語でも基本は同じ
  - アルゴリズムを分割する
  - 分割したところを別の関数に任せる
  - 関数から結果だけ受領

スクリプト読込プログラム 概要

- アルゴリズム概観
  - スクリプトファイルから1行ずつ書かれたデータを1つずつ読み込み、メモリ中のデータ構造に保管
  - できあがったデータ構造を標準出力に書き出し
- 詳細な仕様はWWWのほうを参照のこと

スクリプト読込プログラムが働く様子(想像)

- データ構造はメモリ上に構成される
  - ロッカー(メモリに相当)みたいなもので作業している人を想像するとわかりやすい?



スクリプト読込プログラム チームプレイ(上層部)

- 社長
  - 「ファイルを読込ませ結果を標準出力に出させる」作業を部下の『読込』課長にさせる
  - 『読込』課長
    - ファイルを1行ずつ読み、読んだ行の中身に合わせて各「データ構造を構築してくれる係長」を呼び出す
    - 『物体』係長
    - 『線分』係長
    - 『パッチ』係長
    - 『アニメ』係長
    - 『光源』係長

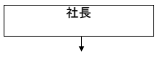
スクリプト読込プログラム チームプレイ（中層部）

- 『物体』係長
  - 新しい物体構造を1つ用意する（だけ）
- 『線分』係長
  - 現在の物体構造中の線分データ集合に線分を1個加える（だけ）
- 『パッチ』係長
  - 現在の物体構造中の三角形パッチ集合に三角形パッチを1個加える（だけ）
- 『アニメ』係長
  - 新しいアニメ構造を1つ用意し、その中身を設定
- 『光源』係長
  - 新しい光源構造を1つ用意し、その中身を設定

スクリプト読込プログラム チームプレイ（下っ端）

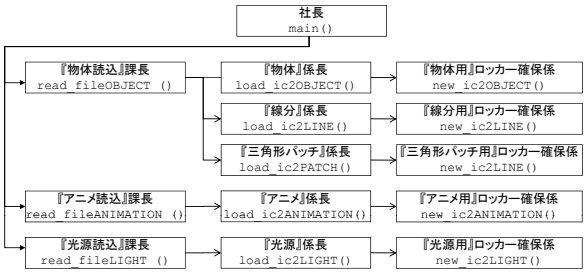
- 『物体』係長
  - 新しい物体構造を1つ用意する（だけ）
  - 『物体用』ロッカー確保アシスタント（係）
    - ロッカーに行って物体1つ分のロッカーを確保して係長にどこを確保したか伝える仕事をする
- 『線分』係長
  - 現在の物体構造中の線分データ集合に線分1個加える（だけ）
  - 『線分用』ロッカー確保アシスタント（係）
    - ロッカーに行って線分1つ分のロッカーを確保して係長にどこを確保したか伝える仕事をする
- （以下同様）

スクリプト読込プログラム 組織図



- 社長しかいないような会社（社長しか働かない会社）は大きくなれない

スクリプト読込プログラム 関数呼出関係図



構造体とポインタ （スクリプト読込プログラムにおける具体例）

目的

構造体の中にポインタ変数を含める

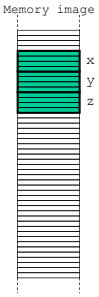
⇒Linked-Listを構築

構造体

- データをまとめて扱えるようにする
  - 変数を幾つでもまとめられる
  - 違う型の変数でもまとめて扱える
- 構造体はintやfloatやcharと同じく、変数を宣言するのに使われる

構造体/定義

- 例：3つの浮動小数点数をまとめて空間中の1点を規定



今後は"struct ic2POINT"という型となる

定義

```
struct ic2POINT {
  float x;
  float y;
  float z;
};
```

構造体中の各変数をメンバと呼ぶ。ここではメンバは3つ。

構造体/利用[単純な例]

- 例：3つの浮動小数点数をまとめて空間中の1点を規定

"struct ic2POINT"という型の変数を使います、ということ。

利用する前にはプログラムの前方で宣言が必要。

宣言

```
struct ic2POINT chouten;
float d;
```

利用

```
d = chouten.x;
chouten.y = d * 2;
```

「ドット」で構造体変数とそのメンバを繋ぐと、メンバの値を参照できる。

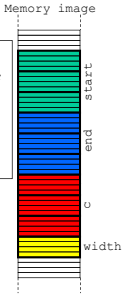
構造体の複雑な例

- 構造体自体も別の構造体に組み込める

```
struct ic2POINT {
  float x;
  float y;
  float z;
};
```

```
struct ic2COLOR {
  float r;
  float g;
  float b;
};
```

```
struct ic2LINEX {
  struct ic2POINT start;
  struct ic2POINT end;
  struct ic2COLOR c;
  float width;
};
```



構造体の複雑な例

- 構造体自体も別の構造体に組み込める

```
struct ic2LINEX {
  struct ic2POINT start;
  struct ic2POINT end;
  struct ic2COLOR c;
  float width;
};
```

```
struct ic2LINEX hasi;
float d;

d = hasi.w / 3.0;
hasi.start.y = d + 1.5;
```

構造体が入れ子になっている場合は、ドット演算子も続けて使う

構造体への演算

- 構造体は変数とはいえ、数値でもなければ文字列でもない（かもしれない）ので、可能な演算の種類は限られる
  - 代入

```
struct ic2LINEX hasi;
struct ic2POINT aa, bb;
float d;

aa = hasi.start;
bb = aa;
bb.z = aa.z * 2.0;
```

  - 四則演算はできない（そもそも想像できないでしょ？）

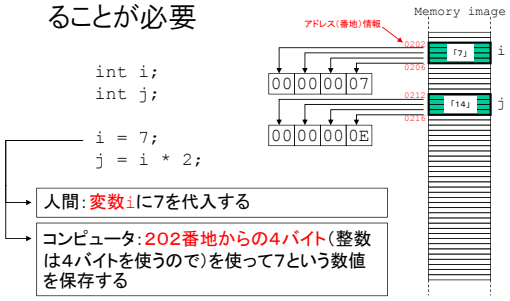
ポインタ・・の前に

- 変数とメモリ空間との関係を正確に把握することが必要。見慣れたプログラムは、実際、どのように動いているのか？

```
int i;
int j;

i = 7;
j = i * 2;
```

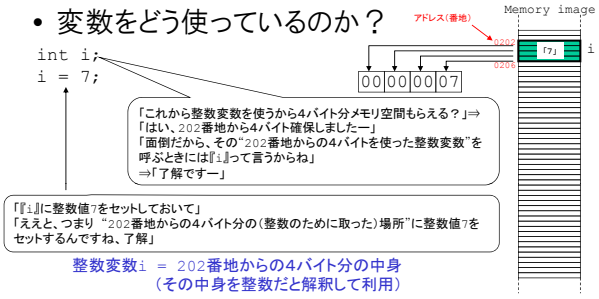
変数の実際

- 変数とメモリ空間との関係を正確に把握することが必要



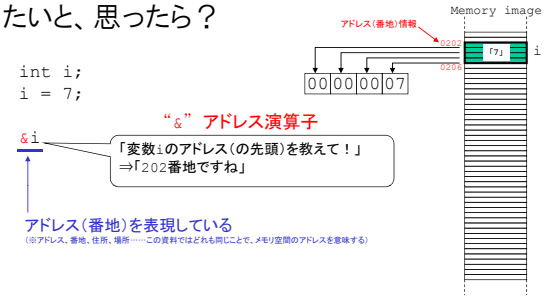
変数の宣言と利用

- 変数の宣言のときに何が起こったのか？
- 変数をどう使っているのか？



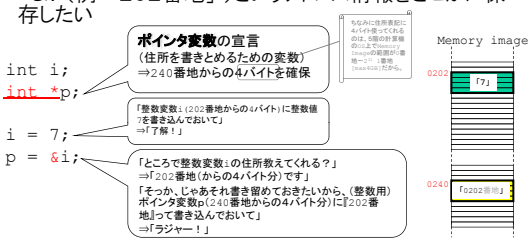
変数のアドレス(アドレス演算子＆)

- もし万一、変数が格納されている場所を知りたいと、思ったら？



ポインタ変数

- アドレスを扱うという概念(が先ほどのスライドで発生)
- 今後、変数として使いたくなる(かも)
  - "&i"(例:「202番地」)というアドレス情報をどこかに保存したい



特別なアドレス値 NULL

- 「アドレスが存在しない」状態を表現する

```
int i;
int *p = NULL;

i = 7;
p = &i;

if (p != NULL)
 printf("num = %d¥n", (*p));
```

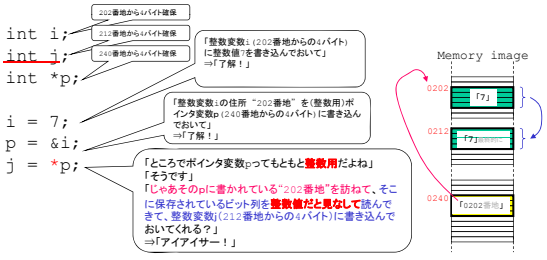
ポインタ変数を確保して、その内容は「まだどのアドレスでもない」状態をNULLで表現

このようにポインタ変数の先の内容を参照する必要があるときは、ポインタ変数がNULLでないことを確認してから実行

このような確認は、例えばa/bの演算をする前にbが0でないことを確認するようなもので、健全なプログラム作成に必須

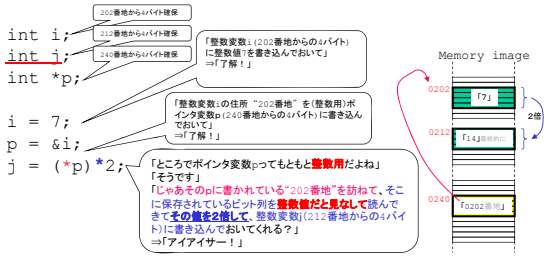
間接演算子＊
(ポインタ演算子ともいう)

- ポインタ変数を使うようになると、「アドレスしか知らない」状態(＝ポインタ変数に格納されたアドレス情報のみ)で、その中身を見たくなることがある(かも)
  - 「で結局202番地には何が入ってるのよ？」



間接演算子＊と乗算演算子＊

- 間接演算子と乗算演算子は全く同じ記号
  - 使い分けは？
    - 演算子＊の前後両方に変数（数値）があれば乗算演算
    - 演算子＊の後方にしか変数（数値）がなければ間接演算
  - とにかく読みにくいので、できるだけ括弧（）を使って見やすく書くことを推奨



構造体変数へのポインタ操作

- 構造体変数であっても、要領は概ね同じ

```
struct ic2LINEX ab;
struct ic2LINEX ac;
struct ic2LINEX *pl;
float *pf;
float f;
```

代入の場合、左辺の型と右辺の型が同じになるように注意すること

```
ab.start.x = 2.2;      float型
pl = &ab;              "struct ic2LINEX"型のためのアドレス
pf = &(ab.end.z);      float型のためのアドレス

f = (*pl).start.y;     float型
ac = (*pl);            "struct ic2LINEX"型
ac.end.x = (*pf) * 1.5;   float型（1.5倍する等の演算をしてもfloat型であることは変わらないため）
```


構造体ポインタからのメンバアクセス

- C言語において、構造体ポインタからメンバへのアクセスには特別にアロー演算子“－＞”が用意されている
  - (*p).x より p->x のほうが読みやすい（この2つは等価）と言われている
  - ちなみに　*p.x はアウト
    ["*"よりも“. ”のほうが優先順位が高いという約束があるので
    p.xは(p.x)を意味することになり意味不明]

```
struct ic2LINEX ab;
struct ic2LINEX *pl;
float f;

pl = &ab;
```

plは構造体struct ic2LINEX型のポインタ

```
f = (*pl).start.y;
f = pl->start.y;
```

どちらの書き方でも演算結果は同じ（だが、下の書き方のほうが読みやすいので推奨しておきます）

ポインタ変数に可能な演算

- 代入
- 間接演算
- アロー演算

```
struct ic2LINEX ab;
struct ic2LINEX *pl;
float f;
pl = &ab;
f = (*pl).start.y;
f = pl->start.y;
```

※アロー演算子はポインタ変数が構造体のために用意されていた場合のみ有効

- 加算・減算（特殊事例）
  - 配列の再勉強をしてから。

ちょっと変わった演算子 sizeof

- ある型があったとき、その型の宣言に必要なバイト数を教えてくれる
- ある変数があったとき、その変数が使用しているバイト数を教えてくれる

```
struct ic2LINEX x;
float f;
printf("size of float is %d¥n", sizeof(float));
printf("size of ic2LINEX is %d¥n", sizeof(struct ic2LINEX));
printf("size of x is %d¥n", sizeof(x));
```

表示結果は、上から順に，4，40，40
sizeof演算子へは、伝統的に引数を()で括って渡す

Linked List
（スクリプト読込プログラムを例として）

目的

事前に予想できない量のデータをメモリ上に動的に確保できるようにする

ポインタと構造体を駆使

Linked Listの利用

さて準備は整った。

データ構造を使って表現したいもの
- 線分
- パッチ
- 物体

問題はスクリプトファイルを開いてみるまで、
- 1つの物体に
  - 何本線分があるかわからない
  - 何枚パッチがあるかわからない
- 物体が何個入っているかわからない
- どれだけアニメーションが続くのかわからない

個数不明なデータに対する処理

Cプログラミング上の問題
- 配列では無理
  - C言語ではプログラム記述時に配列の要素数を指定
    （配列の数はコンパイル時に固定しなくてはいけない）

```
float ff[1000];
struct ic2LINE ll[100];
```

固定値…今回のような問題設定では固定値の指定不能

「十分に余裕をとった数値を設定」というのも1つの考え方だが、
今回はよろしくない(100万個データが来るかもしれない)

動的メモリ確保

- C言語における可変データ数に対する解決策
- データが増えるたびに必要なメモリをOSから貰ってくる
  - 貰ってくるための関数(malloc, calloc)を利用すること

```
#include <stdlib.h>
void *malloc(size_t NBYTES);
void *calloc(size_t N, size_t S);
```


calloc関数

calloc = cleared memory allocationという噂です

- Sバイトのメモリブロックを連続N個分確保
- 先頭のアドレスを返してくれる
  - 確保に失敗した場合はNULLを返してくる
- 確保されたメモリ空間は全て0で初期化済
  - malloc()では初期化してくれない

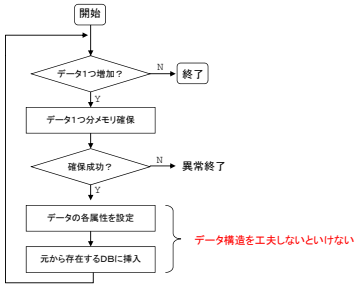
```
#include <stdlib.h>

struct ic2LINEX *ptr = NULL;
ptr = (struct ic2LINEX *)calloc(1, sizeof(struct ic2LINEX));
if (ptr == NULL) return;
ptr->start.x = 2.0;
ptr->end.x   = 4.0;
```

[1] struct ic2LINEを1つ分(40バイトが1つで合計40バイト)確保してもらい、その先頭番地をcalloc()関数が返す。

[2] calloc()が返す番地がどういう変数(struct ic2LINEX)のための番地なのかを明示するためのcast

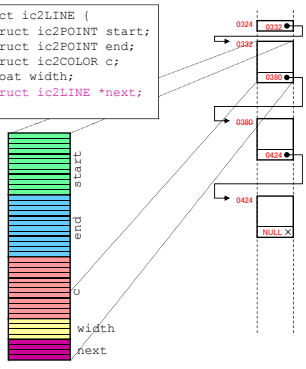
データの増加に対応できるプログラム

- 必要な分だけメモリを確保



Linked List

```
struct ic2LINE {
  struct ic2POINT start;
  struct ic2POINT end;
  struct ic2COLOR c;
  float width;
  struct ic2LINE *next;
};
```



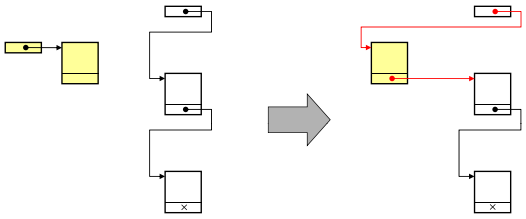
struct ic2LINE構造体の定義で、“次の構造体が存在する住所”を書き留める変数を用意する。

```
struct ic2LINE *next;
```

ポインタ変数nextの中身があれば、その住所を頼って次のデータを発見していく。
ポインタ変数nextの中身がなければ(NULL)、それ以上データがないことにする。

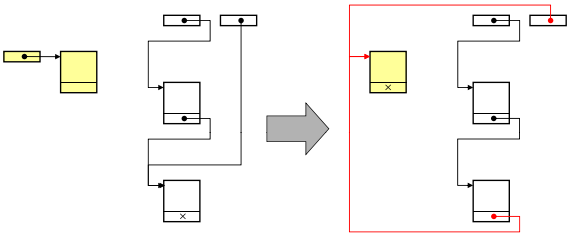
Linked Listの生成（データの挿入）

- 先頭に挿入
  - 本図よりも詳しく具体的な図と説明を課題A-2では行うこと
  - 特にプログラムソースとの具体的な対応を説明すること



Linked Listの生成（データの末端挿入）

- リストの末端に追加
  - 本図よりも詳しく具体的な図と説明を課題A-2では行うこと
  - 特にプログラムソースとの具体的な対応を説明すること



Linked List中のデータへのアクセス

- ポインタ変数の中身に書いてある住所を辿ればよい
- 起点は最初から用意されているポインタ変数

Linked Listの利点と欠点

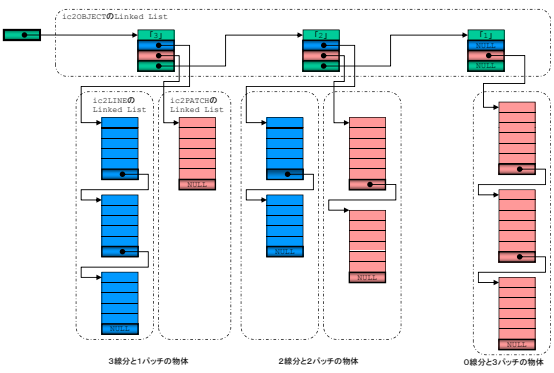
- 利点
  - データの個数に合わせて動的にメモリを確保するので、実行時までデータ数が不明でも対応できる
  - ・
- 欠点
  - ・
  - ・

スクリプト読込プログラム

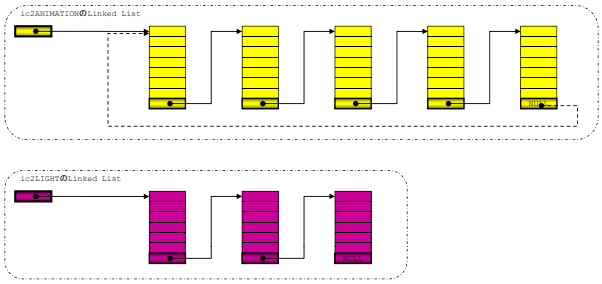
- スクリプトファイル中の全ての記述をメモリ上に保存
- データ構造の設計
  - Linked Listを多用（可変データ数に対応）
    - 物体
      - 線分
      - 三角形パッチ
    - アニメーション
    - 光源

`shortscript039a-*.txt`のメモリイメージを見てみよう

データ構造の概観I（物体）

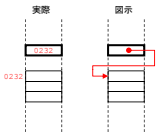


データ構造の概観II（他）

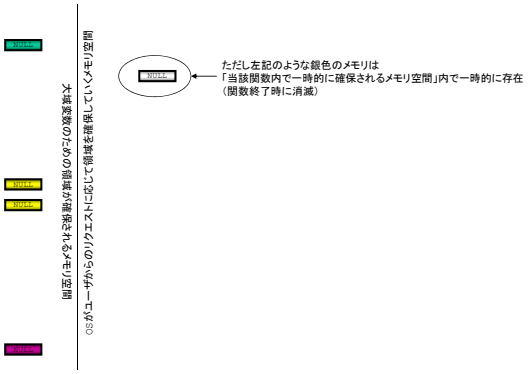


データ構造の構築の様子

- プログラムの実行の様子をメモリイメージで考えてみる



三種類のメモリ空間



データ構造の構築[01]

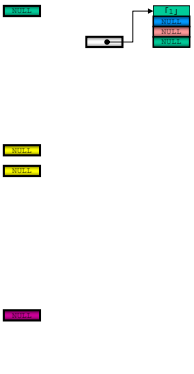
load_ic2OBJECT(), 1回目

データ構造の構築[02]

load_ic2OBJECT(), 1回目

データ構造の構築[03]

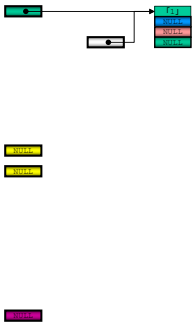
load_ic2OBJECT(), 1回目



Linked List

データ構造の構築[04]

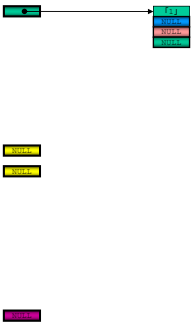
load_ic2OBJECT(), 1回目



Linked List

データ構造の構築[05]

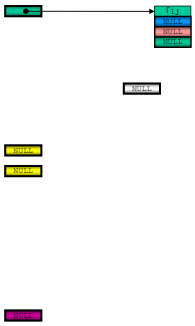
load_ic2OBJECT(), 1回目



Linked List

データ構造の構築[06]

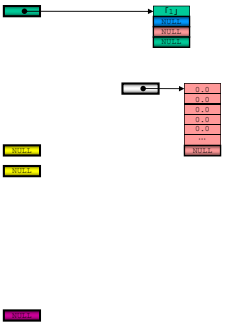
load_ic2PATCH(),1回目



Linked List

データ構造の構築[07]

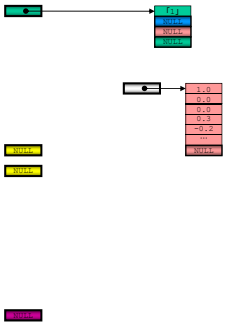
load_ic2PATCH(),1回目



Linked List

データ構造の構築[08]

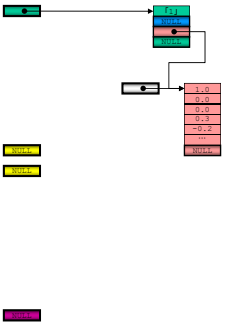
load_ic2PATCH(),1回目



Linked List

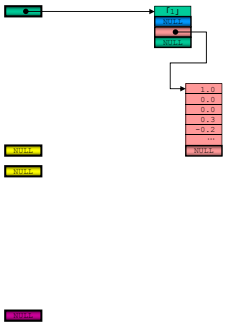
データ構造の構築[09]

load_ic2PATCH(),1回目



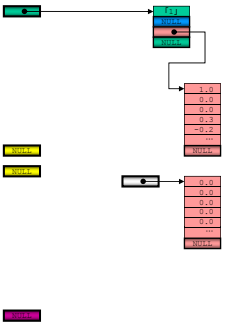
データ構造の構築[10]

load_ic2PATCH(),1回目



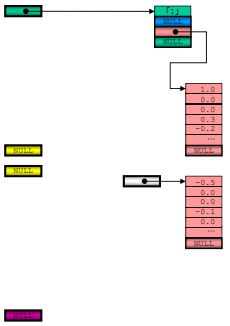
データ構造の構築[11]

load_ic2PATCH(),2回目



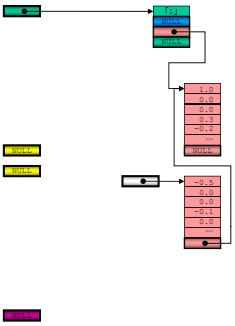
データ構造の構築[12]

load_ic2PATCH(),2回目



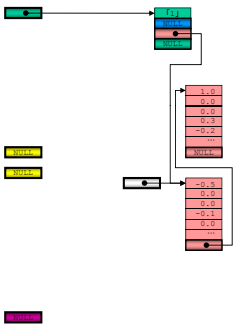
データ構造の構築[13]

load_ic2PATCH(),2回目



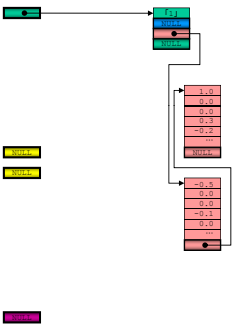
データ構造の構築[14]

load_ic2PATCH(),2回目



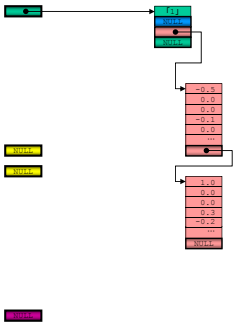
データ構造の構築[15]

load_ic2PATCH(),2回目



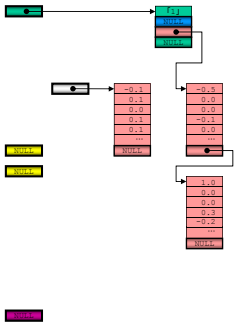
データ構造の構築[15']

load_ic2PATCH(),2回目



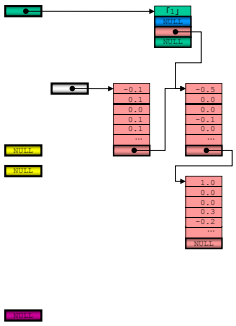
データ構造の構築[16]

load_ic2PATCH(),3回目



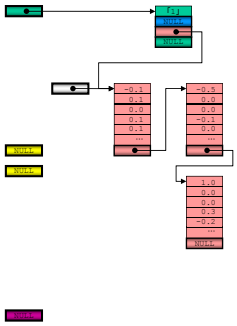
データ構造の構築[17]

load_ic2PATCH(),3回目



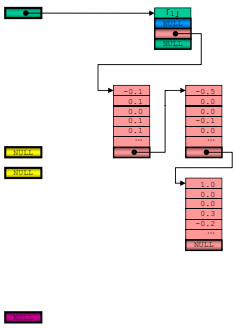
データ構造の構築[18]

load_ic2PATCH(),3回目



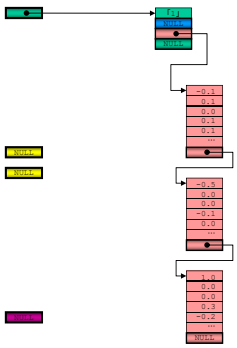
データ構造の構築[19]

load_ic2PATCH(),3回目



データ構造の構築[19']

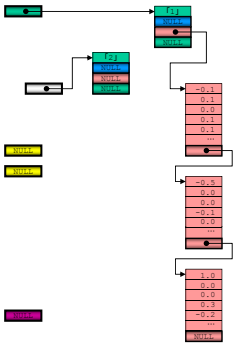
load_ic2PATCH(),3回目



Linked List

データ構造の構築[20]

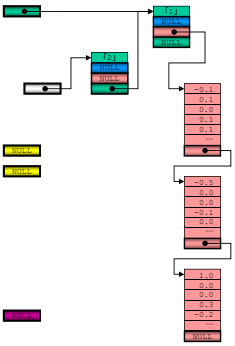
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[21]

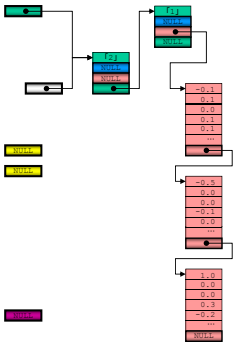
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[22]

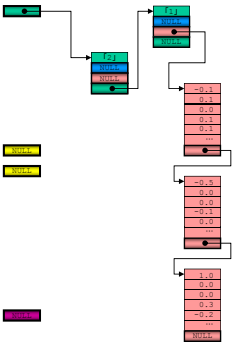
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[22]

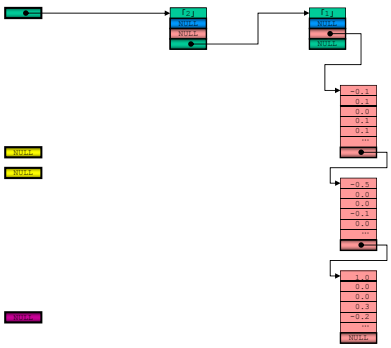
load_ic2OBJECT(),2回目



Linked List

データ構造の構築[22']

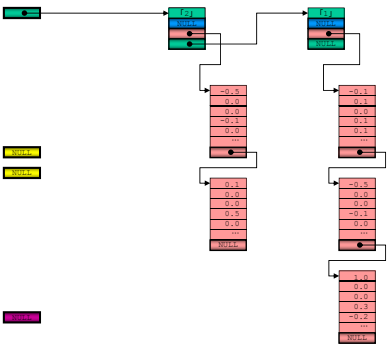
load_ic2OBJECT(),2回目



Linked List

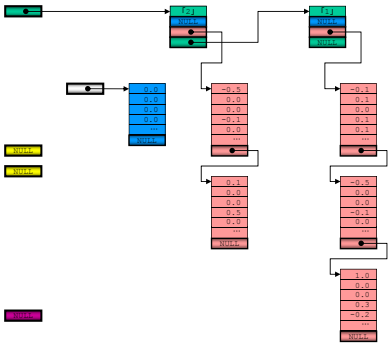
データ構造の構築[23]

load_ic2LINE(),4~5回目



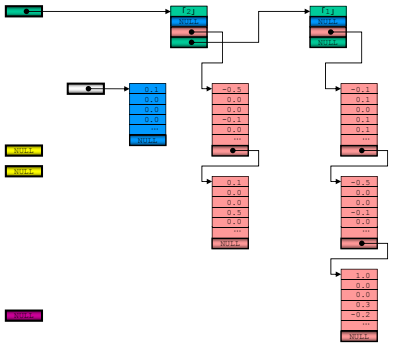
データ構造の構築[24]

load_ic2LINE(),1回目



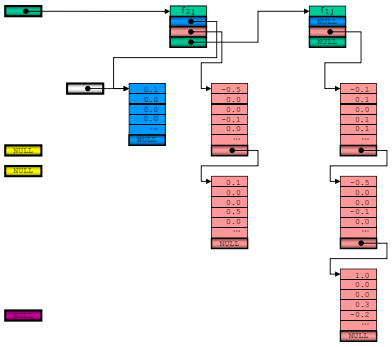
データ構造の構築[25]

load_ic2LINE(),1回目



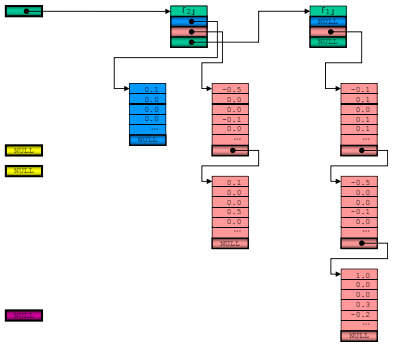
データ構造の構築[26]

load_ic2LINE(),1回目



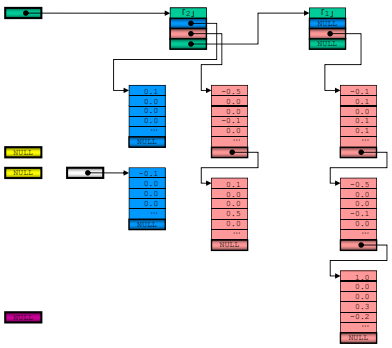
データ構造の構築[27]

load_ic2LINE(),1回目



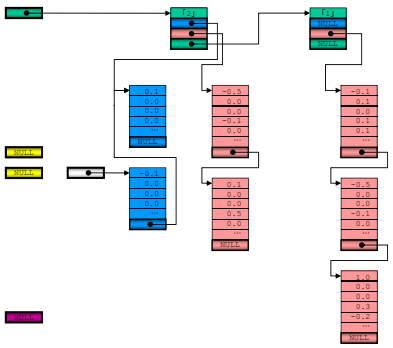
データ構造の構築[28]

load_ic2LINE(),2回目



データ構造の構築[29]

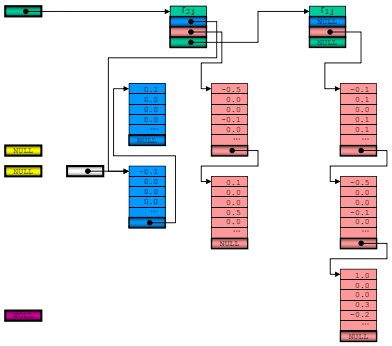
load_ic2LINE(),2回目



Linked List

データ構造の構築[30]

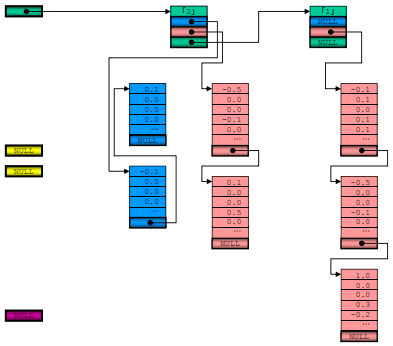
load_ic2LINE(),2回目



Linked List

データ構造の構築[31]

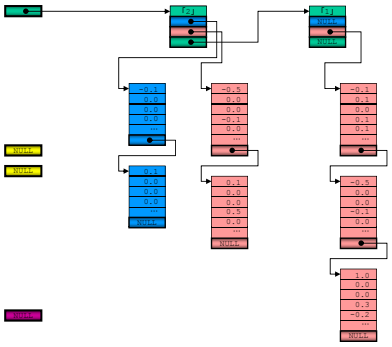
load_ic2LINE(),2回目



Linked List

データ構造の構築[31']

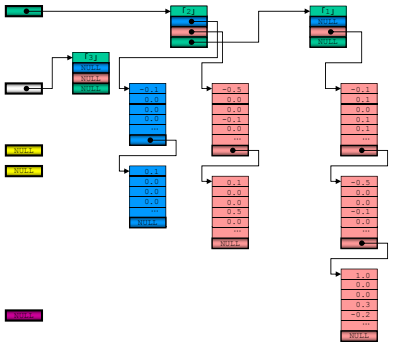
load_ic2LINE(),2回目



Linked List

データ構造の構築[32]

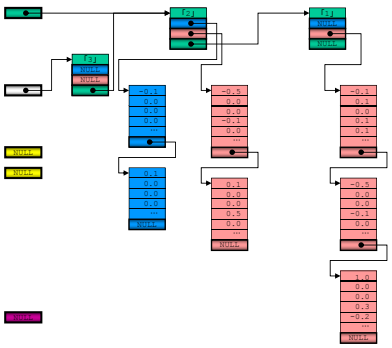
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[33]

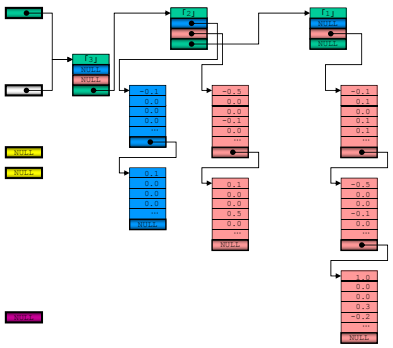
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[34]

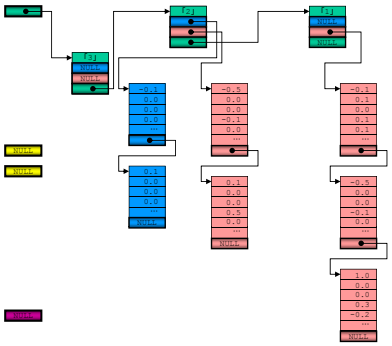
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[35]

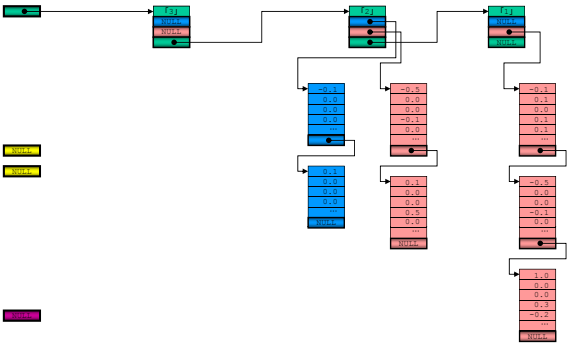
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[35']

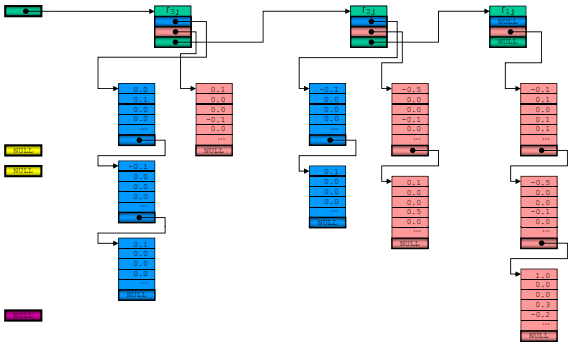
load_ic2OBJECT(),3回目



Linked List

データ構造の構築[36]

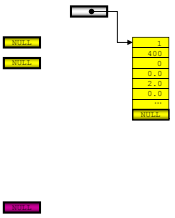
load_ic2PATCH(),6回目
Load_ic2LINE(),3~5回目



Linked List

データ構造の構築[37]

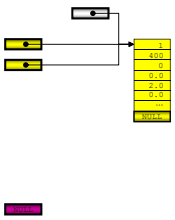
load_ic2ANIMATION(),1回目



Linked List

データ構造の構築[38]

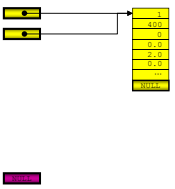
load_ic2ANIMATION(),1回目



Linked List

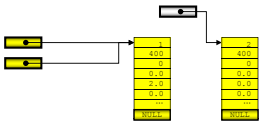
データ構造の構築[39]

load_ic2ANIMATION (),1回目



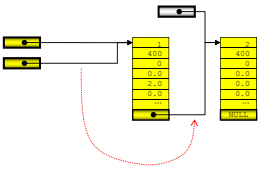
データ構造の構築[40]

load_ic2ANIMATION ()2回目



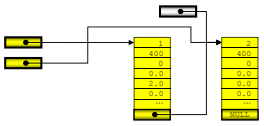
データ構造の構築[41]

load_ic2ANIMATION ()2回目



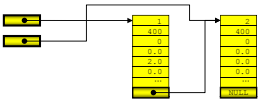
データ構造の構築[42]

load_ic2ANIMATION ()2回目



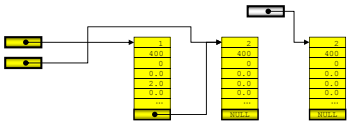
データ構造の構築[43]

load_ic2ANIMATION ()2回目



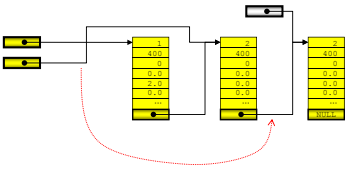
データ構造の構築[44]

load_ic2ANIMATION ()3回目



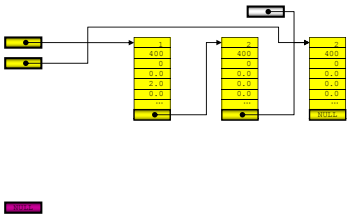
データ構造の構築[44]

load_ic2ANIMATION ()3回目



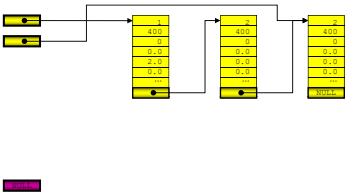
データ構造の構築[45]

load_ic2ANIMATION ()3回目



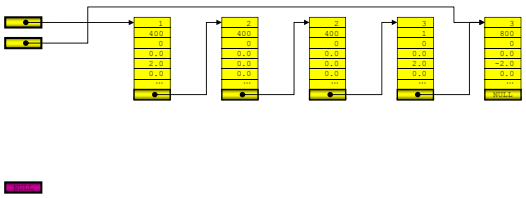
データ構造の構築[46]

load_ic2ANIMATION ()3回目



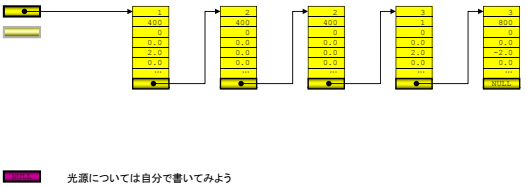
データ構造の構築[47]

load_ic2ANIMATION ()4~5回目



データ構造の構築[48]

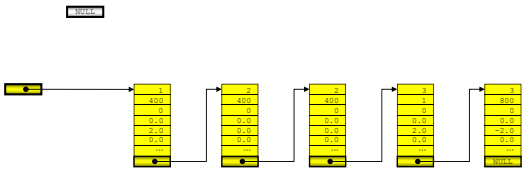
load_ic2ANIMATION ()4~5回目



光源については自分で書いてみよう

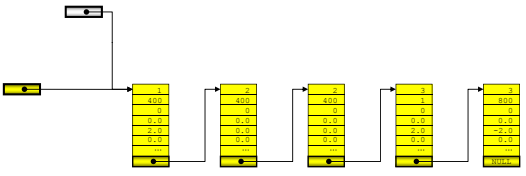
Linked Listの全探索[1]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



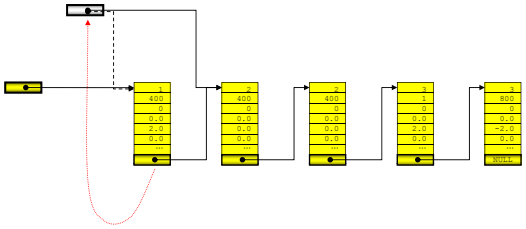
Linked Listの全探索[2]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



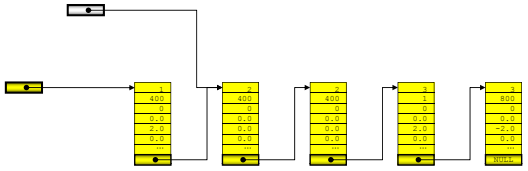
Linked Listの全探索[3]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



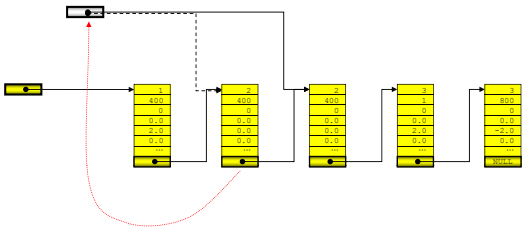
Linked Listの全探索[4]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



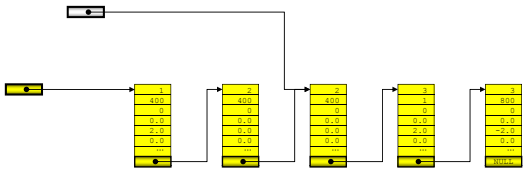
Linked Listの全探索[5]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



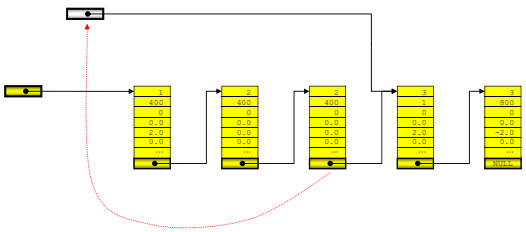
Linked Listの全探索[6]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



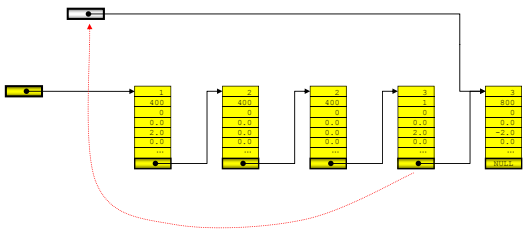
Linked Listの全探索[7]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



Linked Listの全探索[8]

print_all_ic2OBJECTs(), print_all_ic2ANIMEs()等



Linked Listの全探索[9]

print_all_ic2OBJECT(), print_all_ic2ANIMATION ()等