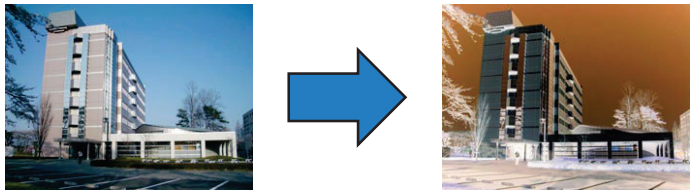
プログラミング演習II
課題4　画像処理
第3週目

担当：篠田一馬

課題4の概要

- 画像処理プログラムを作成することで，ファイル入出力，コマンドライン引数，ポインタ配列，ビット演算を学ぶ



全体の流れ

- 1週目（ファイル操作，コマンドライン引数）
  - 画像ファイルのヘッダ情報をテキストファイルに出力
- 2週目（ファイル操作）
  - 画像ファイルの画素情報をバイナリファイルに出力
- 3週目（ポインタ配列と動的確保）
  - 様々な画素数の画像ファイルの入出力に対応
- 4週目（ビット演算）
  - 画像を減色または上下反転させて出力

画素値の格納方法の改良
①動的な一次元配列

配列の動的確保

- 動的確保はmalloc(or calloc)とfreeが必須

```
int main(void)
{
  unsigned char a[3];

  a[0] = 3;
  a[1] = 4;
  a[2] = 5;

  return 0;
}
```

静的確保

```
int main(void)
{
  unsigned char *a;

  a = malloc(sizeof(unsigned char) * 3);
  a[0] = 3;
  a[1] = 4;
  a[2] = 5;
  free(a);

  return 0;
}
```

動的確保

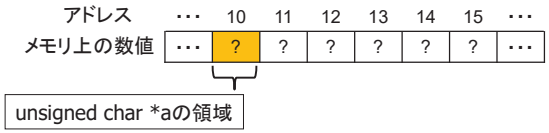
※どちらも，要素数3のunsigned charの配列を確保して値を代入するためのコード

mallocとfree

（アドレス，メモリ上の数値とも10進表記．全ての型は1byte想定．）
（黄色の領域はアドレスの値，青色はそれ以外の値を意味する．）

- unsigned char*の変数を確保
  - アドレスの値を1つ格納するための領域が確保される

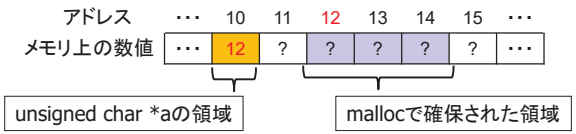
```
int main(void)
{
  unsigned char *a;
```



mallocとfree

- malloc関数
  - 引数分のバイト長の領域を確保し先頭アドレスを返す

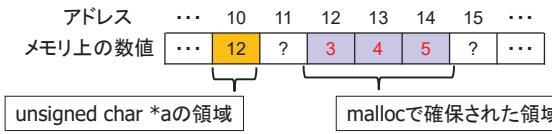
```
int main(void)
{
  unsigned char *a;
  a = malloc(sizeof(unsigned char) * 3);
```



mallocとfree

- 確保領域には静的配列と同様にアクセス可能

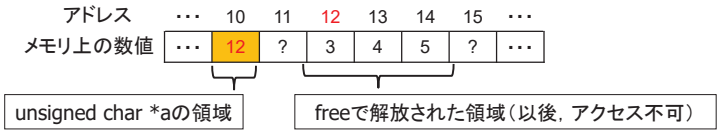
```
int main(void)
{
  unsigned char *a;
  a = malloc(sizeof(unsigned char) * 3);
  a[0] = 3;     //   *(a + 0) = 3;と書いてもOK
  a[1] = 4;     //   *(a + 1) = 4;と書いてもOK
  a[2] = 5;     //   *(a + 2) = 5;と書いてもOK
```



mallocとfree

- free関数
  - 引数で指定されたアドレスから始まる領域を解放

```
int main(void)
{
  unsigned char *a;
  a = malloc(sizeof(unsigned char) * 3);
  a[0] = 3;     //   *(a + 0) = 3;と書いてもOK
  a[1] = 4;     //   *(a + 1) = 4;と書いてもOK
  a[2] = 5;     //   *(a + 2) = 5;と書いてもOK
  free(a);
```



現状のプログラムのpBuffer

- 固定長なのでサイズが異なる画像は対応不可
- pBufferを動的に確保するよう改良

```
typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} PIXEL;

typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PIXEL pBuffer[49152];
} IMAGE;
```

画素値を格納するための構造体

img01.ppm専用に，49152要素
（256×192画素）のPIXELの配列を用意

【作業課題1/7】PIXEL配列の動的確保

- IMAGE構造体のメンバを変更

```
typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} PIXEL;

typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PIXEL *pBuffer;
} IMAGE;
```

pBuffer[49152]ではなく，
ポインタに変更

【作業課題2/7】PIXEL配列の動的確保

- 動的確保用の関数iioMallocImageBufferを新たにプロトタイプ宣言し，動作を定義

```
void iioMallocImageBuffer(IMAGE* pImage);  /* プロトタイプ宣言 */

...

void iioMallocImageBuffer(IMAGE* pImage)
{
  int i;

  pImage->pBuffer = malloc(sizeof(PIXEL) * pImage->xsize * pImage->ysize);
}
```

【作業課題3/7】PIXEL配列の動的確保

- 解放用の関数iioFreeImageBufferを新たにプロトタイプ宣言し，動作を定義

```
void iioFreeImageBuffer(IMAGE* pImage);  /* プロトタイプ宣言 */

...

void iioFreeImageBuffer(IMAGE* pImage)
{
  int i;

  free(pImage->pBuffer);
}
```


【作業課題4/7】PIXEL配列の動的確保

- iioLoadFileのヘッダ読み込みが終了した直後にiioMallocImageBuffer関数の呼び出しを追加

```
int iioLoadFile(IMAGE* pImage, char* fname)
{
   .....
  /* ここでヘッダ部読み込み終了 */

  /* バッファ用メモリ確保 */
  iioMallocImageBuffer(pImage);

  /* バッファに画像データを格納 */
  fread (...);
  .....
}
```


【作業課題5/7】PIXEL配列の動的確保

- ipCopyのp_out_imageに画像サイズを設定した後iioMallocImageBuffer関数の呼び出しを追加

```
void ipCopy(IMAGE* p_in_image, IMAGE* p_out_image)
{
  int i, j;

  p_out_image->xsize = p_in_image->xsize;
  p_out_image->ysize = p_in_image->ysize;
  p_out_image->level = p_in_image->level;
  iioMallocImageBuffer(p_out_image);

    .....
}
```


【作業課題6/7】PIXEL配列の動的確保

- mainのIMAGE構造体が不要になった箇所でiioFreeImageBuffer関数の呼び出しを追加
  - _CrtDumpMemoryLeaksの直前

```
int main(int argc, char* argv[])
{
    .....

  iioFreeImageBuffer(&in_image_data);
  iioFreeImageBuffer(&out_image_data);

  _CrtDumpMemoryLeaks();

  return 0;
}
```

【作業課題7/7】PIXEL配列の動的確保

- 以上の作業後，静的確保時と同様に画像のコピーが出力されることを確認する
- サイズの違う画像ファイルimg02.ppmを講義ページからダウンロードし，入力ファイルに指定することで，画像サイズが変わっても対応できるかどうか確認する
  - メモリリークが起きる場合は誤りがあるため要修正
  - 画像が欠ける場合は，fread時，fwrite時，ipCopy関数内のコピー時の画像サイズの指定を再確認（定数ではなくxsizeやysizeで指定されているかどうか）

作業課題の目安: 15分

画素値の格納方法の改良
②二次元配列

現状と問題点

- pImageのpBufferを動的確保にしたため，どのような画像サイズ（xsize, ysize）でも動作する
  - iioMallocImageBuffer関数の中

```
pImage->pBuffer = malloc(sizeof(PIXEL) * pImage->xsize * pImage->ysize);
```

  - iioFreeImageBuffer関数の中

```
free(pImage->pBuffer);
```

- ただし，ある特定の画素値にアクセスする方法はまだ不便

1次元配列上のアクセス方法

- 256x192画素の画像データを格納したと仮定
- 上から3行目，左から4列目のrの画素値を100にしたい場合，1次元配列だとアクセスが面倒

```
int main(void)
{
    .....

    pImage->pBuffer[515].r = 100;

    .....
    return 0;
}
```

IMAGE構造体内のpBufferを
1次元配列として確保した場合

2次元の画素値を1次元のメモリ上に
並べているため，上から3かつ左から4は
516番目の画素となる

2次元的なアクセス方法

- 下記のコードのようにアクセスできると簡単
  - インデクスは0から開始されることに注意
  - 実装には「ポインタの配列」が必要

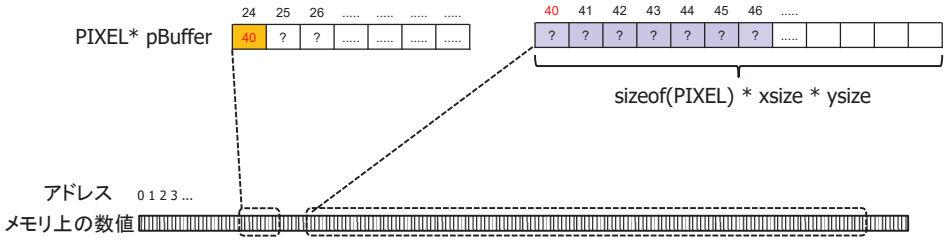
```
int main(void)
{
    .....

    pImage->pBuffer[2][3].r = 100;

    .....
    return 0;
}
```

IMAGE構造体内の画素値の配列を
2次元配列のように確保した場合

2次元の画素値を上図の矢印のような
概念でアクセスできれば簡単

1次元配列による画素値の管理

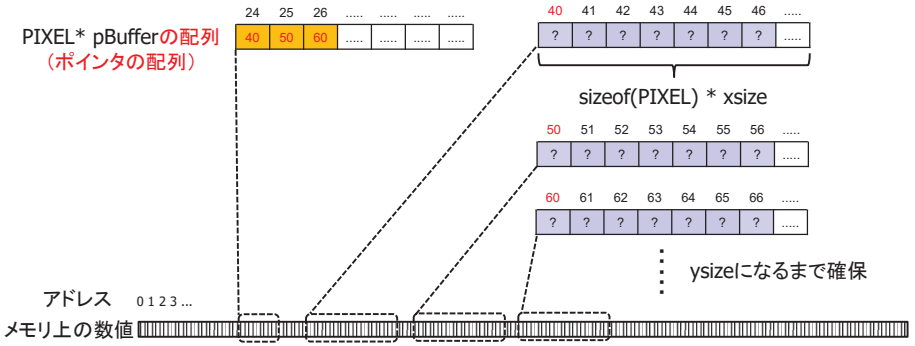
- これまでは縦×横の全画素分を一度に確保
  - 1次元配列なのでアクセスが面倒



（以後，アドレス，メモリ上の数値とも10進表記．簡単化するため全ての型は1byteを想定する．
画素数は数画素のみを想定した場合．黄色はアドレスの値，青色は画素の値を意味する．）

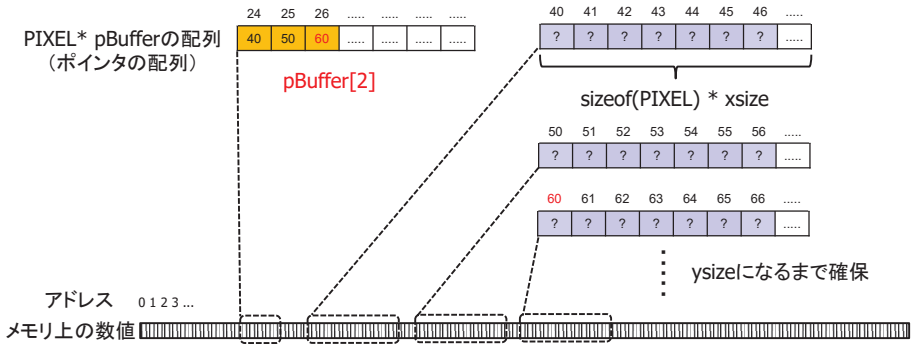
2次元配列による画素値の管理

- 1行分ごとに別々の領域で確保した方が楽
- 各領域の先頭アドレスを配列で管理



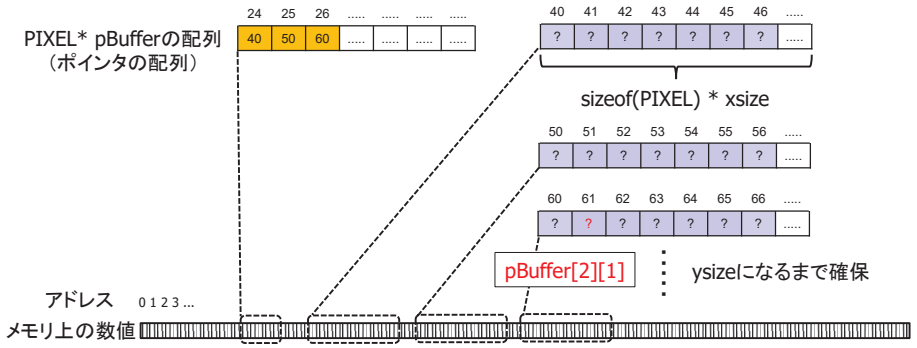
2次元配列による画素値へのアクセス

- pBuffer[2]で上から3番目の行のアドレス



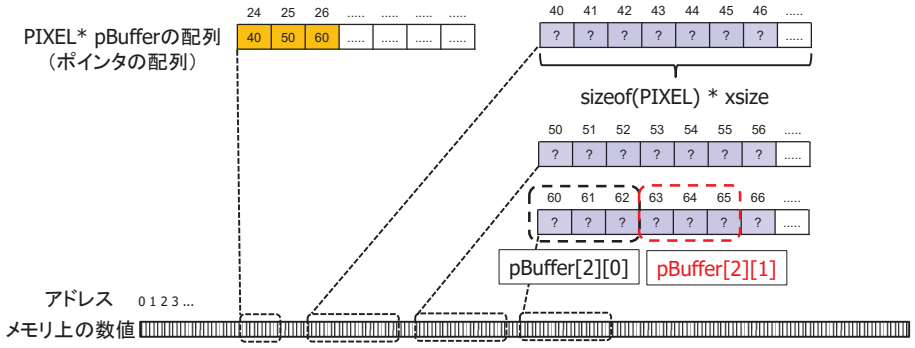
2次元配列による画素値へのアクセス

- pBuffer[2]で上から3番目の行のアドレス
- pBuffer[2][1]で上から3，左から2の画素値



より正確には

- 一つの画素は3バイト（PIXEL構造体）
  - インデクス+1で一つ先の構造体要素を参照

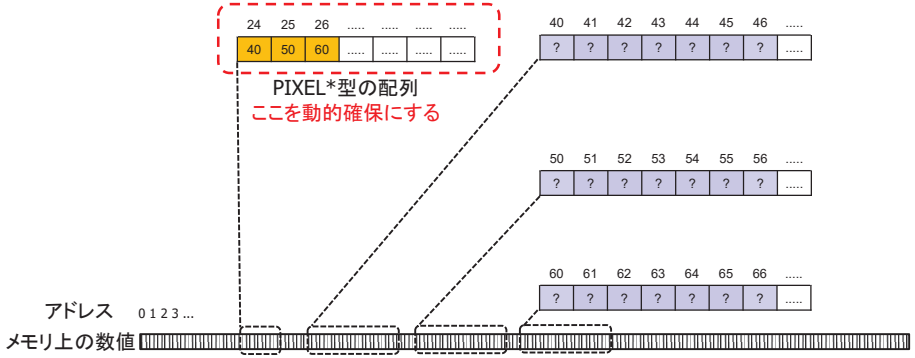


（本スライドのみ，sizeof(PIXEL)として3バイトを想定）

ポインタ配列の動的確保

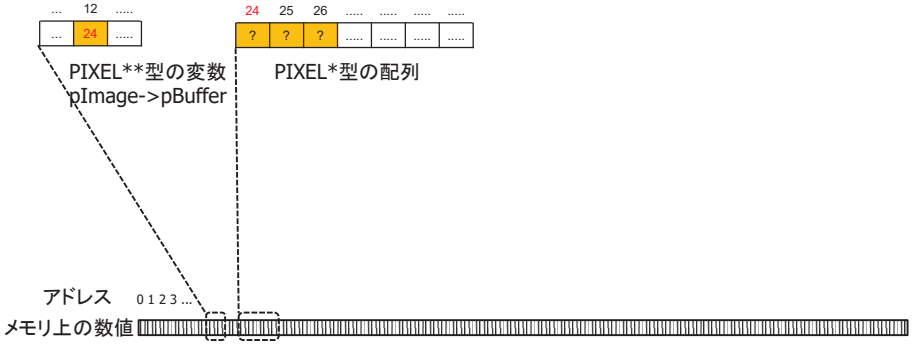
ポインタ配列の動的確保の概要

- ysizeは画像を読み込むまで不定
  - PIXEL*を各要素に持つ配列を動的確保する必要



ポインタ配列の動的確保の概要

- PIXEL*を要素とする配列をmallocで確保し，PIXEL**変数に先頭アドレスを格納すればよい



ポインタ配列の動的確保の準備

- pBufferでポインタ配列の先頭アドレスを管理
  - PIXEL型の領域のアドレスを格納したい→PIXEL*
  - PIXEL*型の領域のアドレスを格納したい→PIXEL**

```
typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} PIXEL;

typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PIXEL *pBuffer;
} IMAGE;
```

```
typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} PIXEL;

typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PIXEL **pBuffer;
} IMAGE;
```


（補足）PIXEL *, PIXEL **

- 下記の通りPIXEL*をPPIXELと定義してもよい
  - その場合，PIXEL **pBufferはPPIXEL *pBuffer

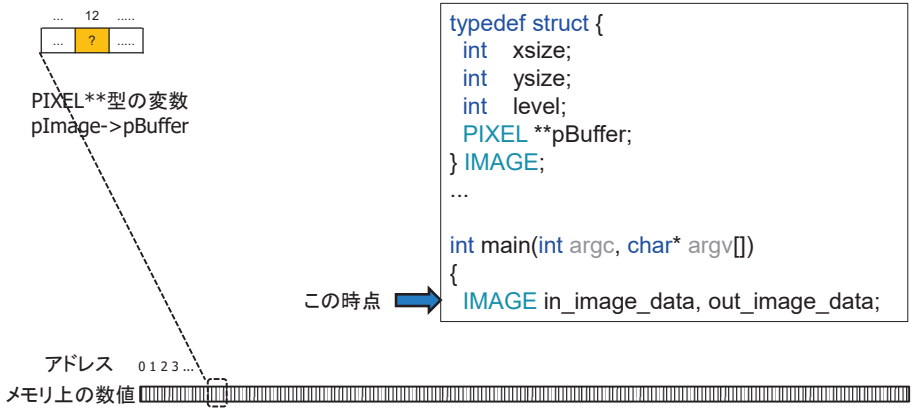
```
typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} PIXEL;

typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PIXEL **pBuffer;
} IMAGE;
```

```
typedef struct {
  unsigned char r;
  unsigned char g;
  unsigned char b;
} PIXEL, *PPIXEL;

typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PPIXEL *pBuffer;
} IMAGE;
```

上記はどちらでもよい

領域確保の手順

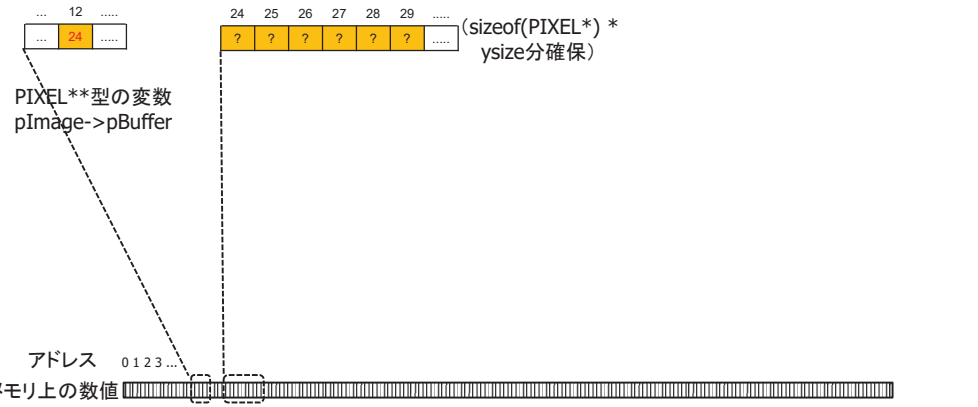
- IMAGE構造体の変数を宣言したときには，pImage->pBufferの格納場所が確保されるだけ




```
typedef struct {
  int    xsize;
  int    ysize;
  int    level;
  PIXEL **pBuffer;
} IMAGE;
...

int main(int argc, char* argv[])
{
  IMAGE in_image_data, out_image_data;
```

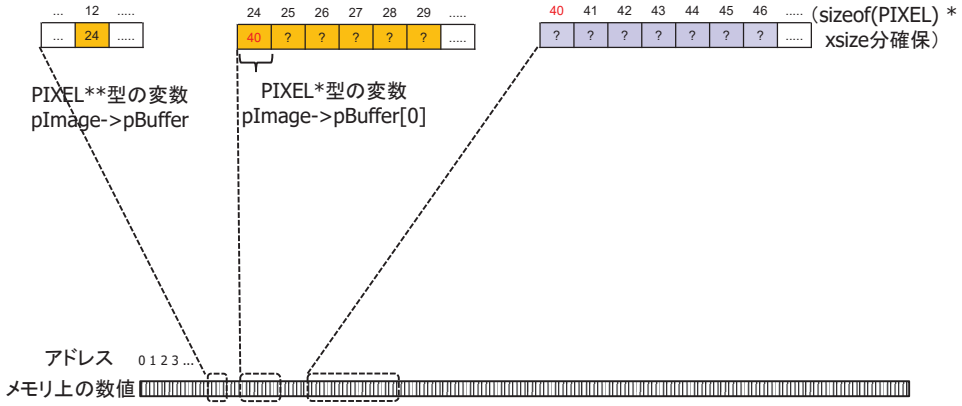

領域確保の手順

- まず，ポインタ配列をysize分だけmallocで動的確保して先頭アドレスをpBufferに格納



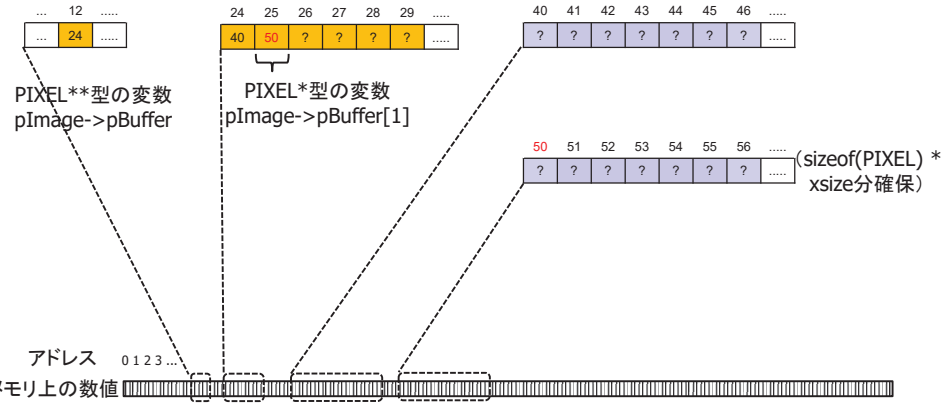
領域確保の手順

- xsize分のPIXEL構造体をmallocで動的確保して先頭アドレスをpImage->pBuffer[0]に格納



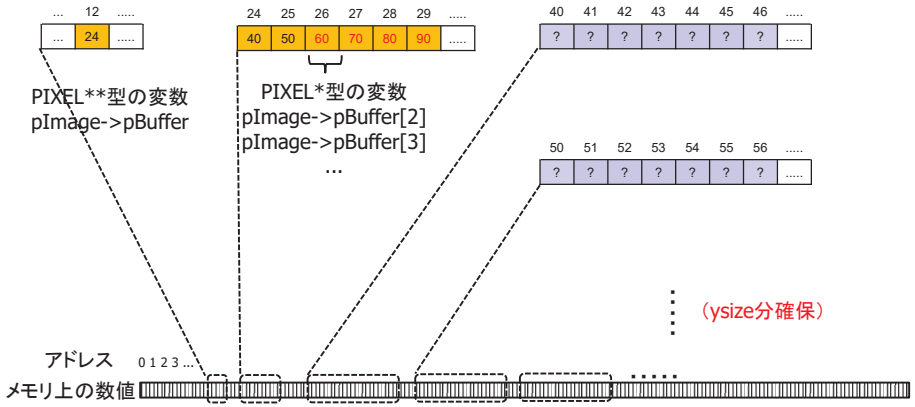
領域確保の手順

- xsize分のPIXEL構造体をmallocで動的確保して先頭アドレスをpImage->pBuffer[1]に格納



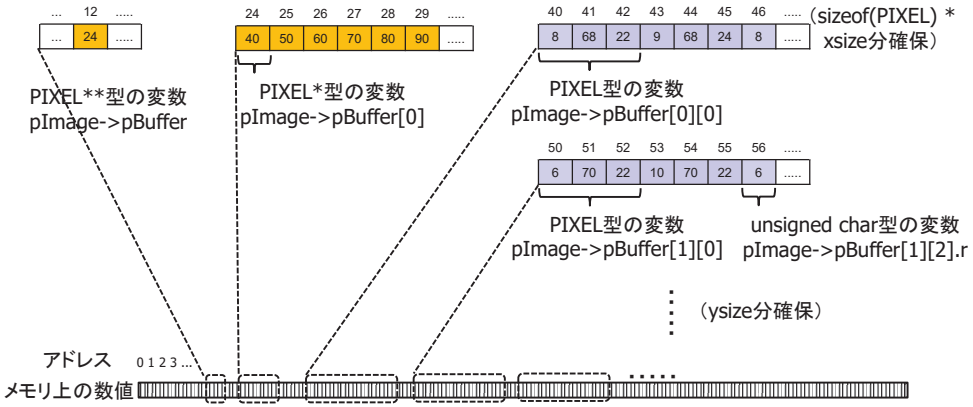
領域確保の手順

■これをysize回繰り返す



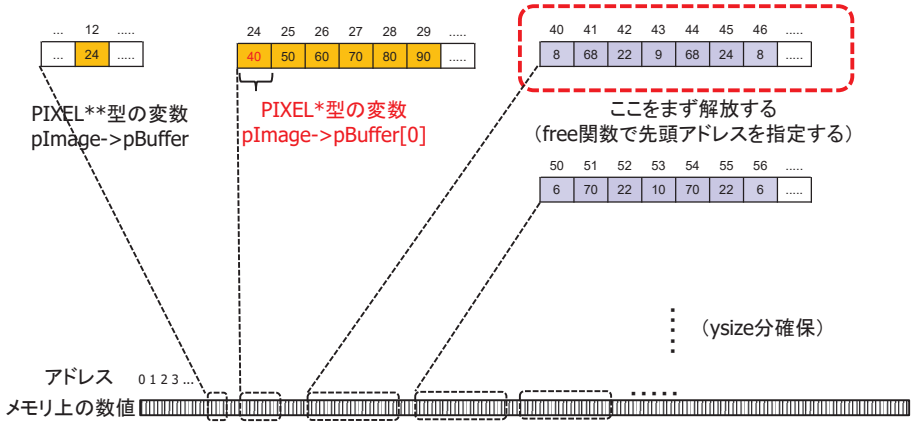
領域確保が完了したら

■画素値の書き込み/読み込みが可能になる



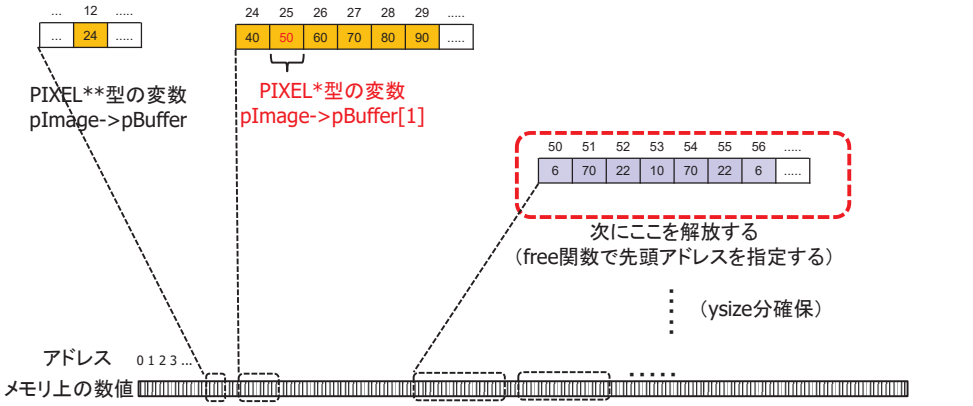
領域解放の手順

■確保手順とは逆に，画素値の領域から解放必要



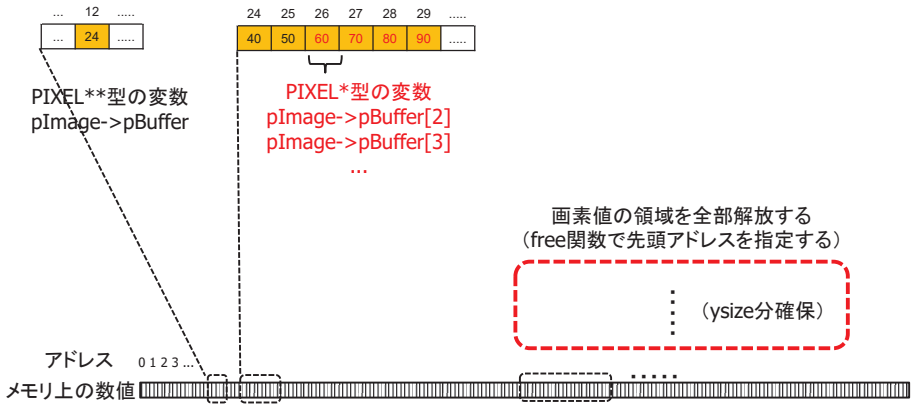
領域解放の手順

■確保手順とは逆に，画素値の領域から解放必要



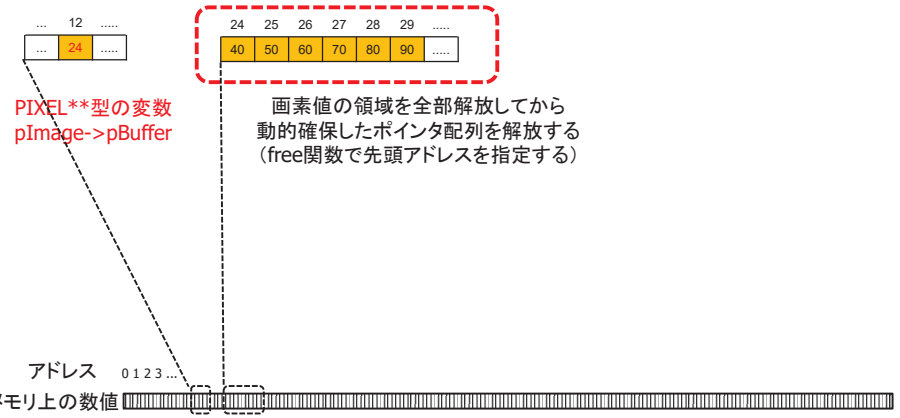
領域解放の手順

■確保手順とは逆に，画素値の領域から解放必要



領域解放の手順

■確保手順とは逆に，画素値の領域から解放必要



動的確保と解放の実装方針

```
void iioMallocImageBuffer(IMAGE* pImage)
{
    int i;
    pImage->pBuffer ← 各要素がPIXEL*となる長さpImage->ysizeの配列を動的確保
    for (i = 0; i < pImage->ysize; ++i) {
        pImage->pBuffer[i] ← 各要素がPIXELとなる長さpImage->xsizeの配列を動的確保
    }
}
```

```
void iioFreeImageBuffer(IMAGE* pImage)
{
    int i;
    for (i = 0; i < pImage->ysize; ++i) {
        pImage->pBufferの各要素を解放
    }
    pImage->pBufferを解放
}
```


【演習課題4_3_1】

■ポインタ配列の動的確保を利用し，IMAGE構造体のpBufferを2次元配列のようにアクセスできるよう修正する

■ipCopyの画素値コピーを下記の通り修正して，動作するプログラムにすることが目標

```
for (j = 0; j < p_in_image->ysize; j++) {
    for (i = 0; i < p_in_image->xsize; i++) {
        p_out_image->pBuffer[j][i].r = p_in_image->pBuffer[j][i].r;
        p_out_image->pBuffer[j][i].g = p_in_image->pBuffer[j][i].g;
        p_out_image->pBuffer[j][i].b = p_in_image->pBuffer[j][i].b;
    }
}
```

※rgbを分けずに p_out_image->pBuffer[j][i] = p_in_image->pBuffer[j][i];としてもよい.

【演習課題4_3_1】

- 修正箇所
  - IMAGE構造体のpBufferをPIXEL **型にする
  - iioMallocImageBufferでメモリの動的確保
  - iioFreeImageBufferでメモリの解放
  - freadとfwrite部分も書き換え必要
    - 一度に全画素を読み書きするのではなく，1行ごと（1回に横画素数分ごと）に読み書きする必要がある
  - ipCopyを前ページの通り修正
- img01.ppmとimg02.ppmのどちらもコピーできることを確認する

演習課題の目安: 授業終了時刻まで

終わった人は

- 余裕があれば次週の課題に取り組むことを推奨
- 最終レポートは，画像処理の実装種類数が多いほど評価UP
  - 例年，10種類以上実装する人も数名