

画素値の格納方法の改良

②二次元配列

現状と問題点

- plImage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

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

- iioFreeImageBuffer関数の中

```
free(plImage->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)
{
    .....

    plImage->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)
{
    .....

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

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

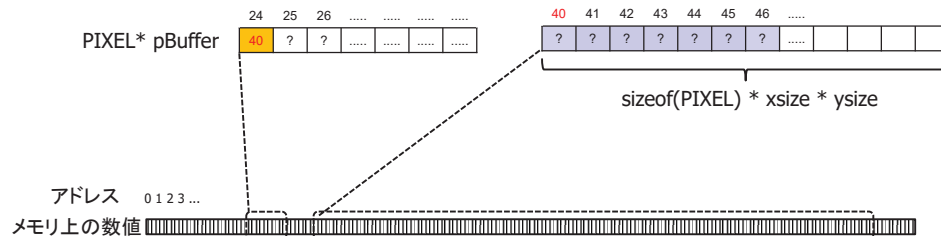
IMAGE構造体内の画素値の配列を
2次元配列のように確保した場合



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

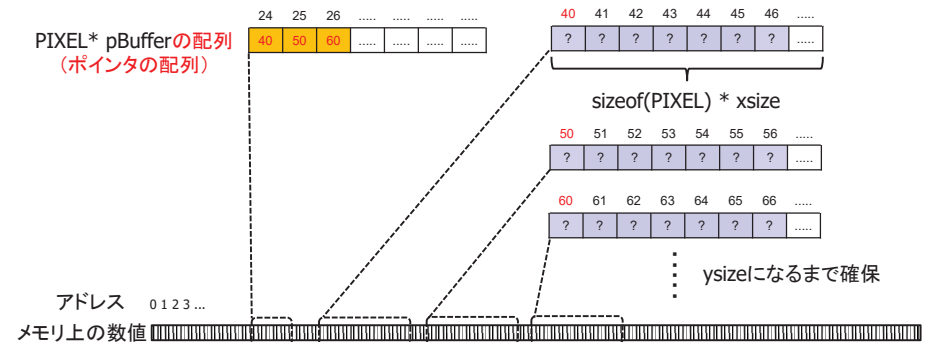


(以後、アドレス、メモリ上の数値とも10進表記。簡単化するため全ての型は1byteを想定する。
画素数は数画素のみを想定した場合、黄色はアドレスの値、青色は画素の値を意味する。)

5

2次元配列による画素値の管理

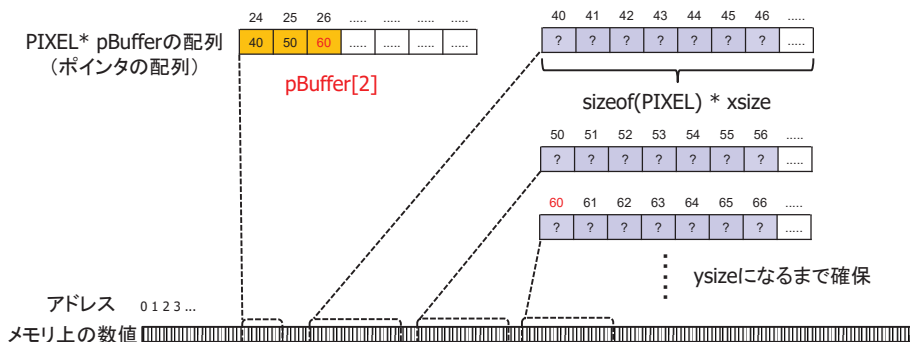
- 1行分ごとに別々の領域で確保した方が楽
- 各領域の先頭アドレスを配列で管理



6

2次元配列による画素値へのアクセス

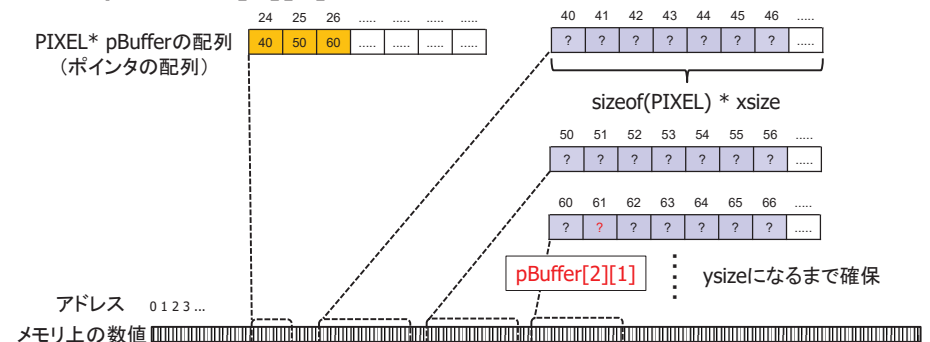
- pBuffer[2]で上から3番目の行のアドレス



7

2次元配列による画素値へのアクセス

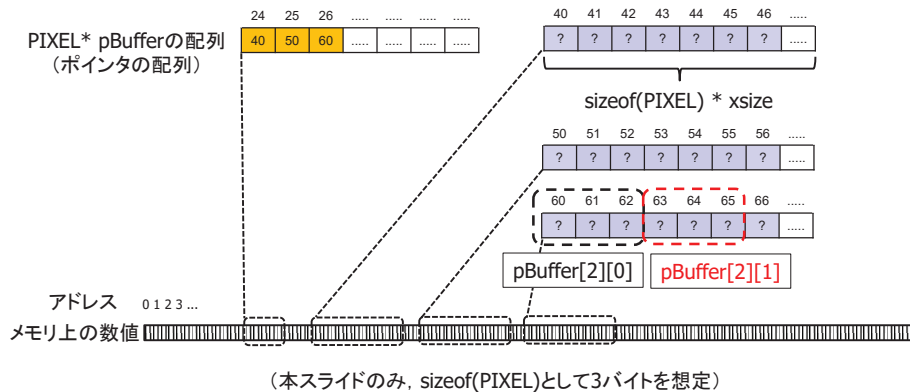
- pBuffer[2]で上から3番目の行のアドレス
- pBuffer[2][1]で上から3, 左から2の画素値



8

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



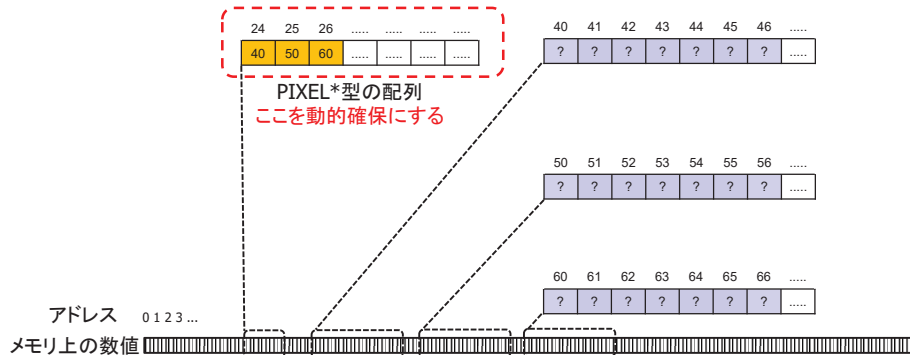
9

ポインタ配列の動的確保

10

ポインタ配列の動的確保の概要

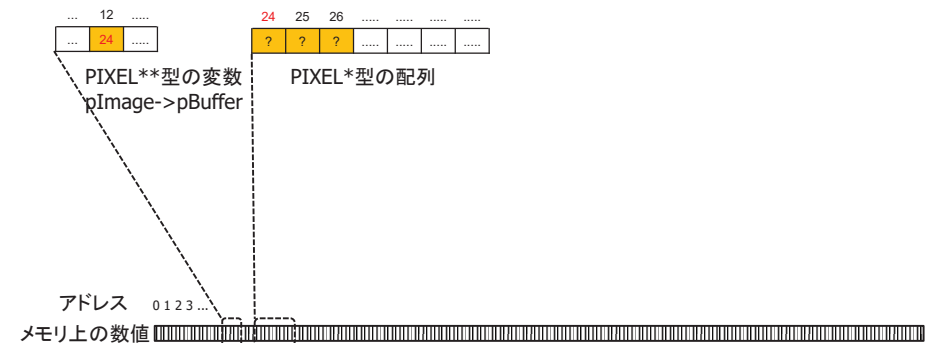
- ysizeは画像を読み込むまで不定
- PIXEL*を各要素に持つ配列を動的確保する必要



11

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



12

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;
```

13

(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

14

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

PIXEL**型の変数
pImage->pBuffer

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

...

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

この時点 →

アドレス 0 1 2 3 ...
メモリ上の数値

15

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

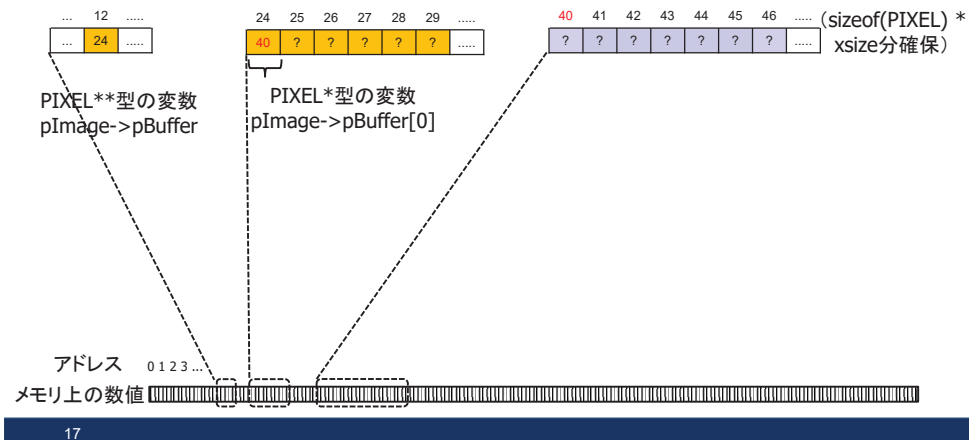
PIXEL**型の変数
pImage->pBuffer

アドレス 0 1 2 3 ...
メモリ上の数値

16

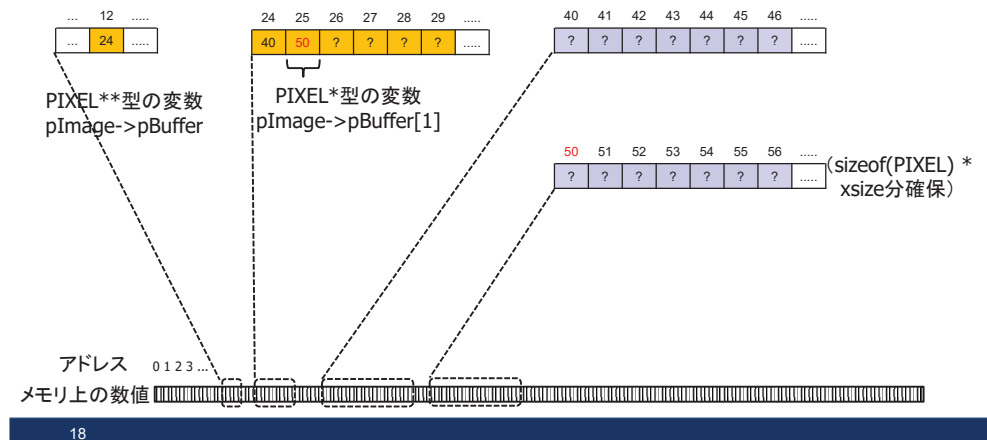
領域確保の手順

- 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



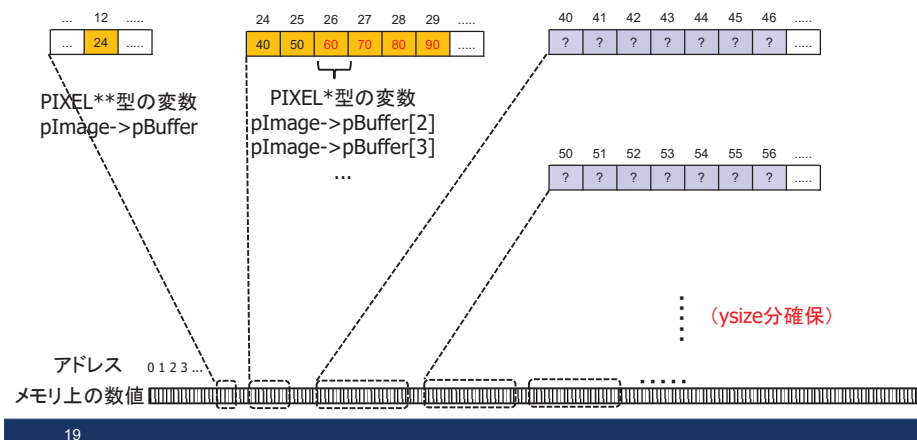
領域確保の手順

- 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



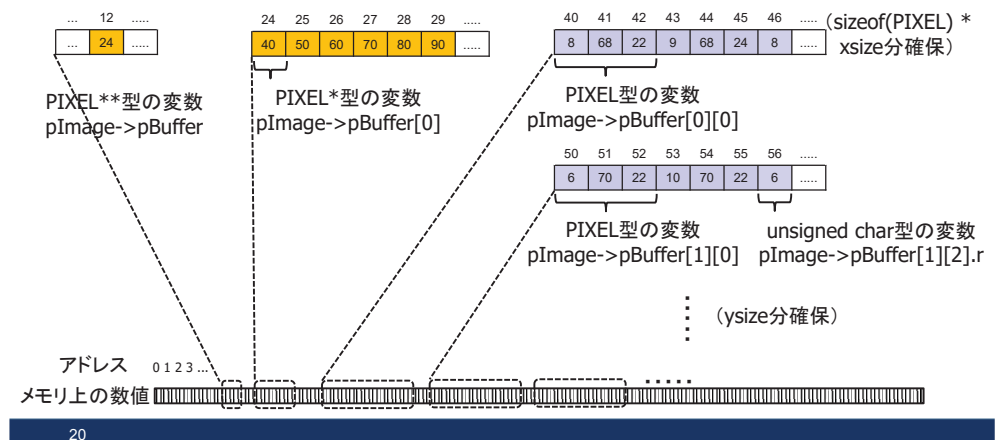
領域確保の手順

- これをysize回繰り返す



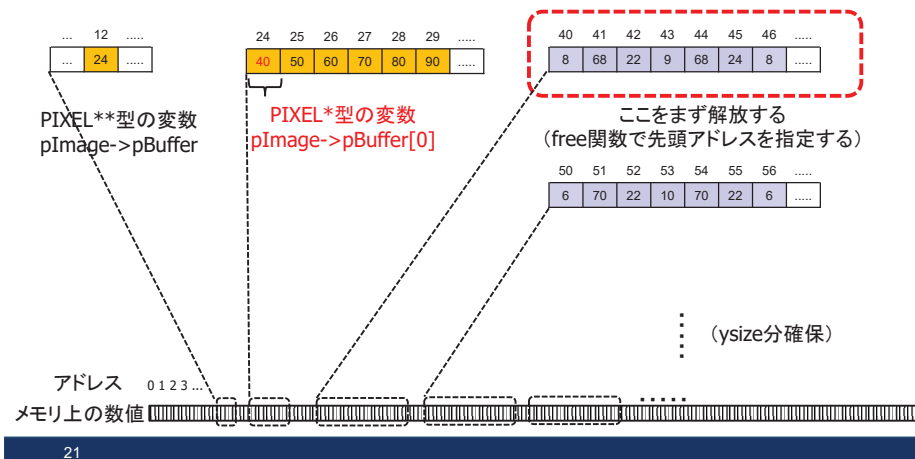
領域確保が完了したら

- 画素値の書き込み/読み込みが可能になる



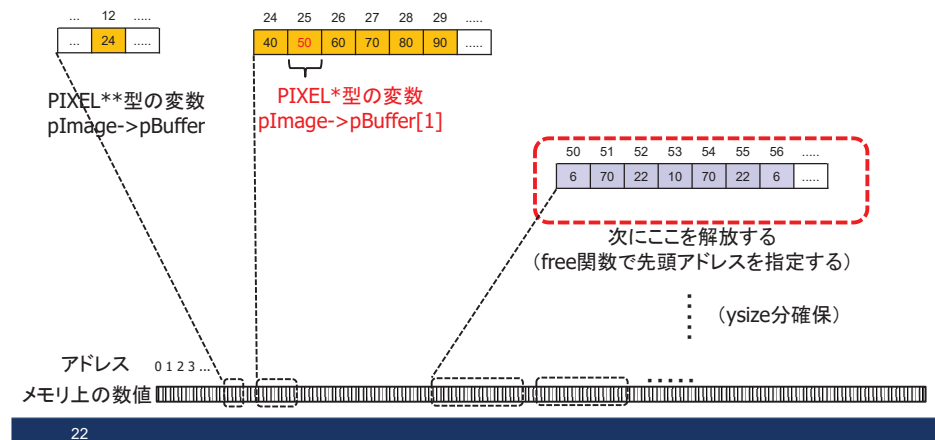
領域解放の手順

■ 確保手順とは逆に、画素値の領域から解放必要



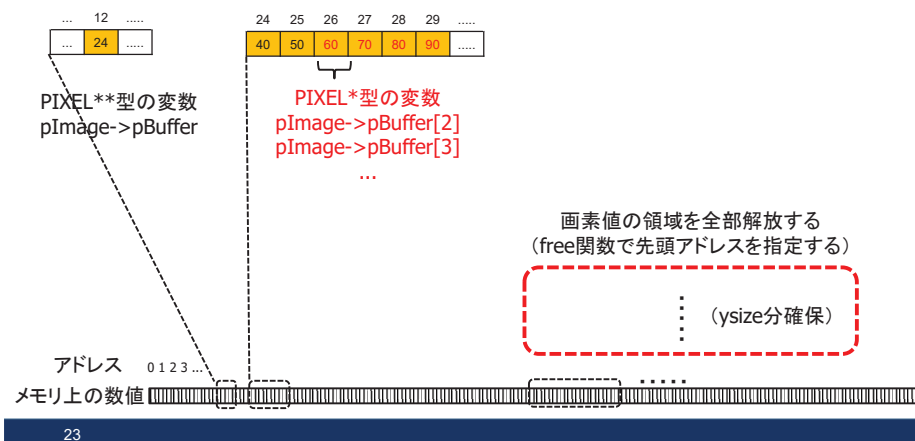
領域解放の手順

■ 確保手順とは逆に、画素値の領域から解放必要



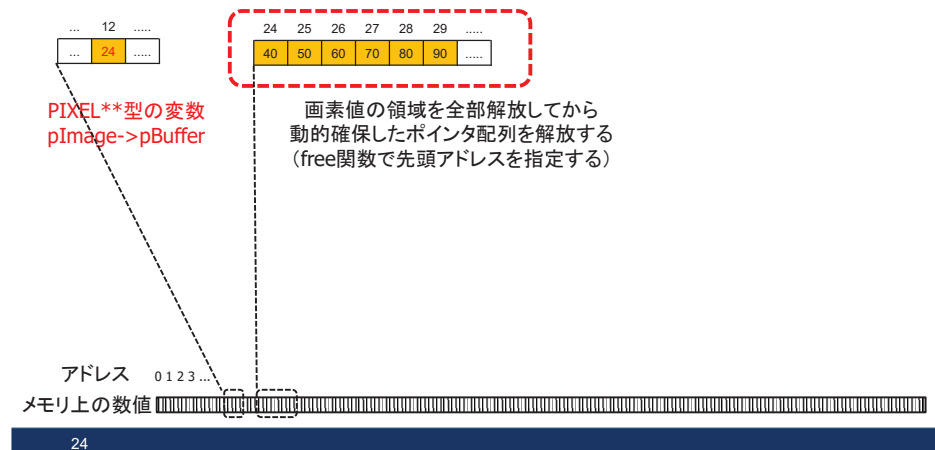
領域解放の手順

■ 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
}
```

25

【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。

26

【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

27

演習課題の目安: 授業終了時刻まで

28