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

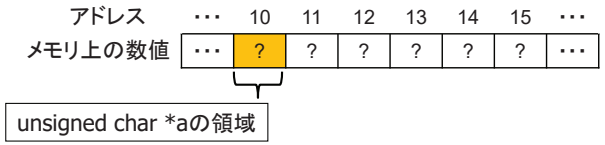
※どちらも，要素数3のunsigned charの配列を確保して値を代入するためのコード

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

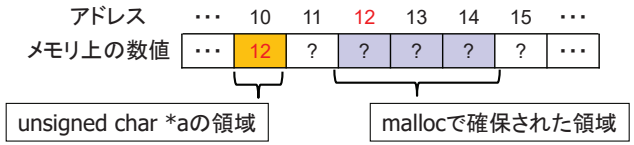
```
int main(void)
{
  unsigned char *a;
```



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

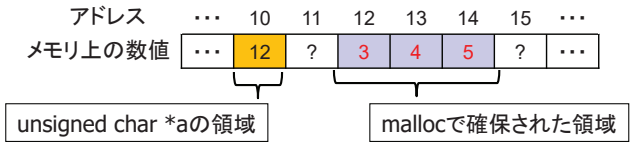
```
int main(void)
{
  unsigned char *a;
  a = malloc(sizeof(unsigned char) * 3);
```



mallocとfree

- 確保領域には静的配列と同様にアクセス可能

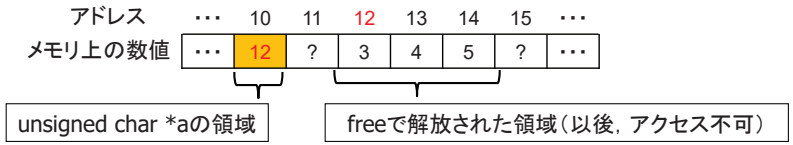
```
int main(void)
{
  unsigned char *a;
  a = malloc(sizeof(unsigned char) * 3);
  a[0] = 3;     //   *(a + 0) = 3;と書いてもOK
  a[1] = 4;     //   *(a + 1) = 4;と書いてもOK
  a[2] = 5;     //   *(a + 2) = 5;と書いてもOK
```



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とすることで，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