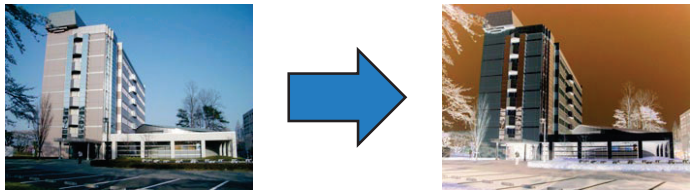
プログラミング演習II
課題4　画像処理
第4週目

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

モジュール設計

現時点のプログラムの構造

- main.cに全ての変数と関数を定義している
  - 可読性低い，毎回全てコンパイル必要，秘匿性低い

```
#include ...
#define ...
構造体の定義
関数プロトタイプ宣言

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

iioLoadFile関数の定義
iioSaveFile関数の定義
iioMallocImageBuffer関数の定義
iioFreeImageBuffer関数の定義
ipCopy関数の定義
```

iioLoadFileにしか関係しない定数

ファイルの読み書きや
領域の確保解放に関係するもの

画像処理（画素値の編集）に関係するもの

モジュール設計

- 機能ごとに独立したファイルに分割する
  - 可読性の向上，分割コンパイル可能，情報の秘匿

main.c
画像ファイルをロードする関数を呼び出す
画像処理関数を呼び出す
画像ファイルをセーブする関数を呼び出す
バッファを解放する

img_io.c
画像ファイルのロード関数の定義
画像ファイルのセーブ関数の定義
画素値用バッファを確保する関数の定義
画素値用バッファを解放する関数の定義

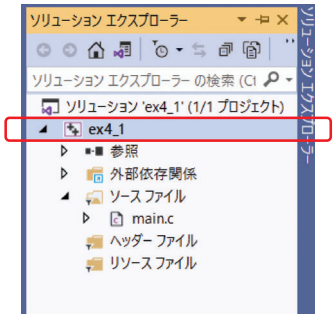
img_proc.c
画像処理のための関数の定義

モジュール設計の方針

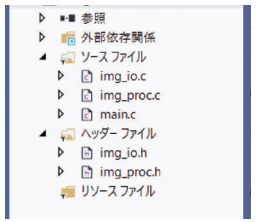
- main
  - 各ヘッダファイルのinclude
  - main関数
- img_io
  - PIXEL構造体およびIMAGE構造体の定義
  - iioLoadFile，iioSaveFile関数
  - iioMallocImageBuffer，iioFreeImageBuffer関数
- img_proc
  - ipCopy関数
  - その他の画像処理関数

【作業課題1/6】ファイルの追加

- main.cと同フォルダに以下のcファイルとヘッダファイルを新規に作成し，プロジェクトに追加
  - img_io.c
  - img_io.h
  - img_proc.c
  - img_proc.h

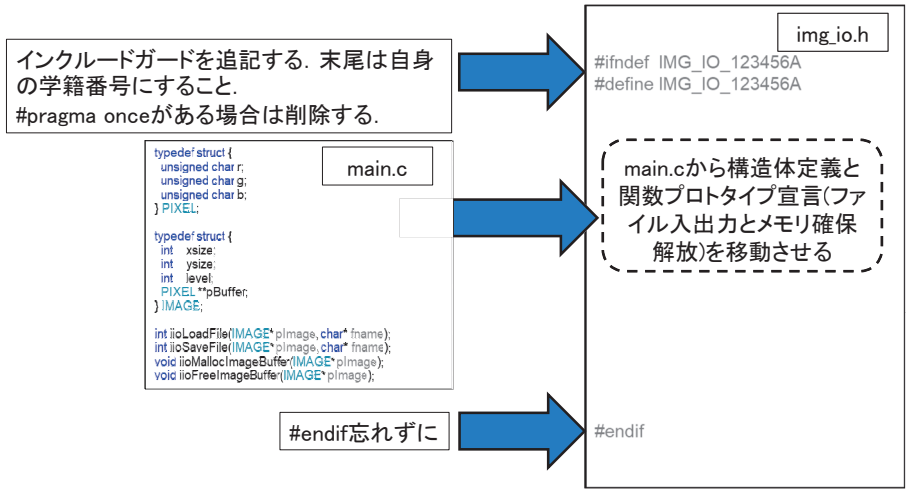


プロジェクト（赤枠）上で右クリック
→「追加」→「新しい項目」
でcファイルとhファイルを一つずつ追加する

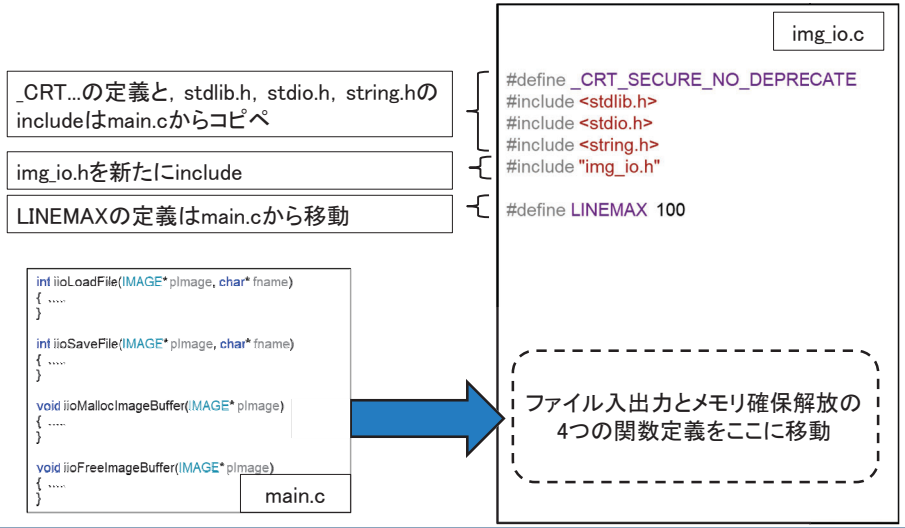


追加後，こうなればOK

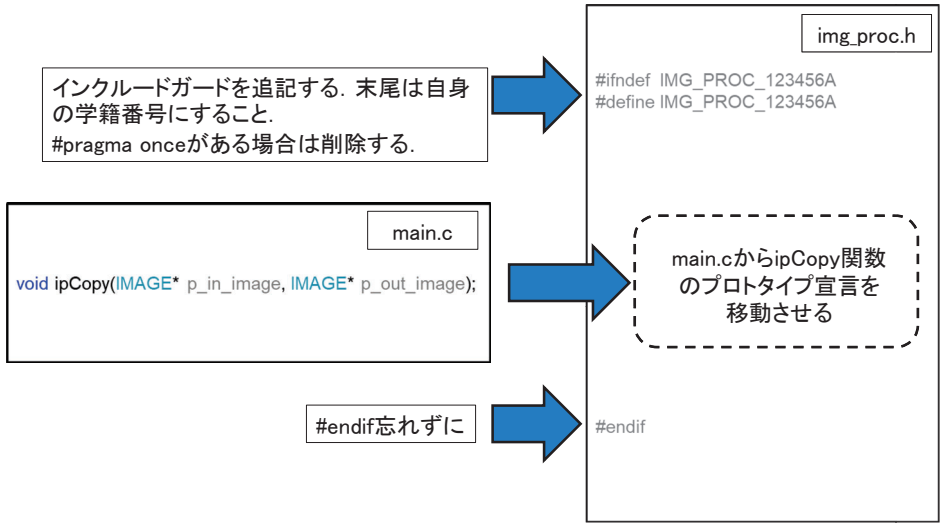
【作業課題2/6】img_io.hの作成



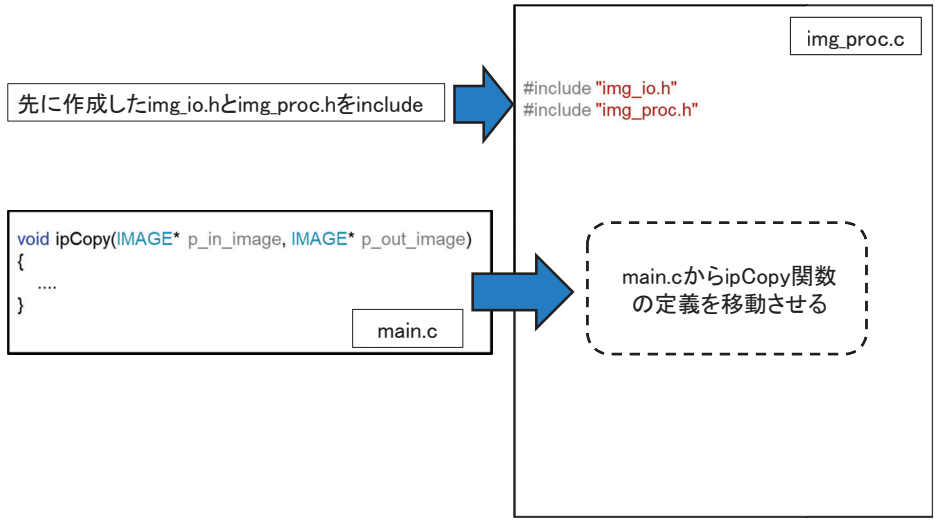
【作業課題3/6】img_io.cの作成



【作業課題4/6】img_proc.hの作成



【作業課題5/6】img_proc.cの作成



【作業課題6/6】main.cの修正

2つのヘッダファイルを追加でinclude

main.c

```
....

#include "img_io.h"
#include "img_proc.h"

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

main.cではiioFreeImageBuffer関数やipCopy関数を呼び出す必要があるため，どちらのヘッダも必要になる．

モジュール化の確認

- これでモジュール化が完了したので，実行する
  - これまでと同じ結果が得られるか確認する
  - また，main.cがいかにコンパクトになったか確認する

main.c
画像ファイルをロードする関数を呼び出す
画像処理関数を呼び出す
画像ファイルをセーブする関数を呼び出す
バッファを解放する

img_io.c
画像ファイルのロード関数の定義
画像ファイルのセーブ関数の定義
画素値用バッファを確保する関数の定義
画素値用バッファを解放する関数の定義

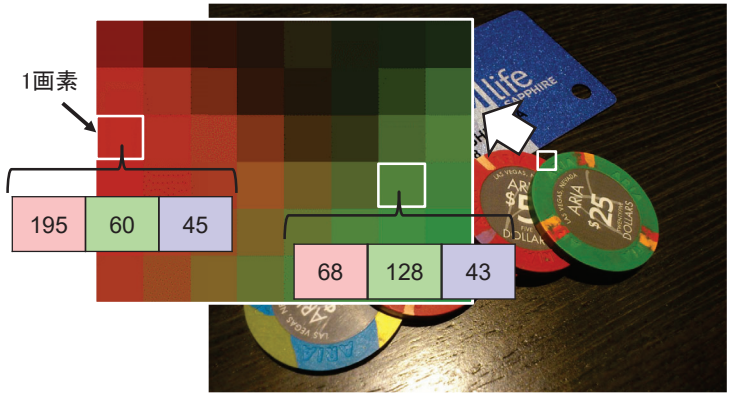
img_proc.c
画像処理のための関数の定義

作業課題の目安: 10分

画素値の基本（復習）

ディジタル画像データの基本

- 画素ごとに赤，緑，青の3色の情報を持っている
- 赤，緑，青（RGB）の組み合わせで色を表現



画素値の例

- 一般的には0から255（8 bit）で表現
- 全部で16,777,216色を表現可能

R	G	B	色
0	0	0	
255	0	0	
0	255	0	
0	0	255	
255	255	0	
0	255	255	
255	0	255	

R	G	B	色
10	10	10	
128	128	128	
200	200	200	
255	255	255	
255	192	0	
202	254	207	
235	189	173	

画素値の範囲について

- 0から255なので，unsigned char に格納
- 型変換やオーバーフロー/アンダーフローに注意

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

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

例：

```
pImage->pBuffer[0][0].r = 0;     /* rは0 */
pImage->pBuffer[0][0].r = 100; /* rは100 */
pImage->pBuffer[0][0].r = 300; /* rは44 */
pImage->pBuffer[0][0].r = 255 - 256; /* rは255 */
```


ビット演算を利用した減色処理

減色処理

- RGB各色の数値の組み合わせを減らしつつ，原画像の色に近い色に割り当て

原画像
RGB各色　0から255
色数：256^3 = 16,777,216色

減色画像の例
RGB各色　0, 64, 128, 192のみ
色数：4^3 = 64色
下位6ビットを0にしている

減色処理の実装方法

- If文で切り分ける
  - 例：0以上64未満の画素値は全て0にする，64以上128未満の画素値は全て64にする・・・
- 割り算
  - 例：画素値を64で割ってから小数点を切り捨てて，64倍する
- ビット演算を使うと最も簡単

ビット演算

- 2進表記の各桁で論理演算を行う
  - AND(&)，OR(|)，右ビットSHIFT(>>)の例

```
unsigned char A = 113;   /* 0111 0001 */
unsigned char B = 31;    /* 0001 1111 */
unsigned char D;

D = A & B;               /* AND:   0001 0001 (17) */
D = A | B;               /* OR:    0111 1111  (127) */
D = A >> 3;              /* SHIFT: 0000 1110  (14) */
```

他にもXOR(^)，左ビットSHIFT(<<)，NOT(~)などがある

右シフト演算の補足

- 符号なし変数:
  - 隙間は0で埋められる (論理シフト)
- 符号あり変数のシフト
  - 隙間は最上位ビットで埋められる* (算術シフト)

```
unsigned char A = 113;   /* 0111 0001 */
char C = -64;            /* 1100 0000 */
char D;

D = A >> 3;              /* 0000 1110 (14) */
D = C >> 3;              /* 1111 1000 (-8) */
```

AND演算のマスクと16進数表記

- AND演算を利用すると，所望のビットを残して，他を切り捨てることができる
  - Bにおいて0の桁は，ANDをとると必ず0
  - Bにおいて1の桁は，ANDととると必ずAと一致
  - このような処理をビットマスクとよぶ

```
unsigned char A = 113;   /* 0111 0001 */
unsigned char B = 31;    /* 0001 1111 */
unsigned char D;

D = A & B;               /* AND:  0001 0001 (17) */
```

上位3ビットを0にするマスク

必ず0　Aのビットと一致

マスクBの計算が面倒であれば

- 10進表記ではなく16進表記を利用すると楽
  - 2進数の下位4ビットは16進数の下一桁目，2進数の上位4ビットは16進数の次の桁に相当
  - 0001→1，1111→F

```
unsigned char A = 113;   /* 0111 0001 */
unsigned char B = 0x1F;  /* 0001 1111 */
unsigned char D;

D = A & B;               /* AND:  0001 0001 (17) */
```

上位3ビットを0にするマスク

必ず0　Aのビットと一致

【演習課題4_4_1】

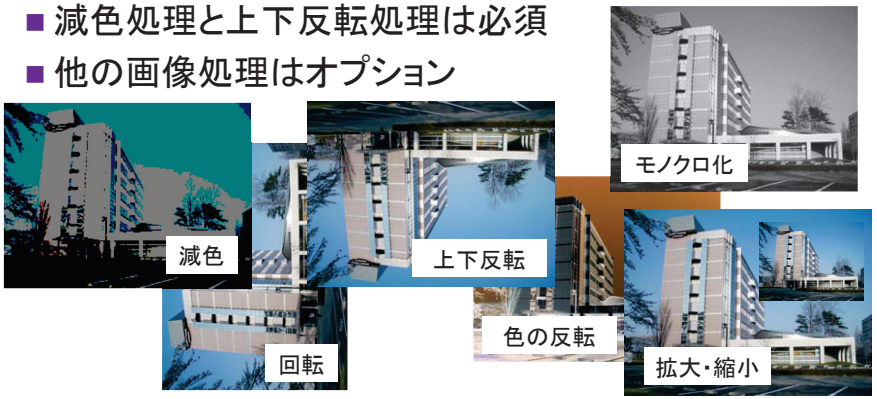
- 画像の各画素の下位nビットを0でビットマスクすることで，減色画像を作成する
  - n = (自身の学籍番号の下一桁 % 7) + 1 とする
  - img_procにipBitMask関数として実装すること
  - プロトタイプ宣言も忘れずに追加
  - main.cでは，ipCopy関数の代わりにipBitMask関数を呼び出す

演習課題の目安: 10分

最終レポートについて

課題4の最終レポート

- ユーザが指定した方法で入力画像を処理し，結果を別ファイルとして出力するプログラムを作成
  - 減色処理と上下反転処理は必須
  - 他の画像処理はオプション

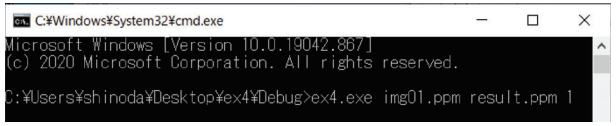


コマンドライン引数について

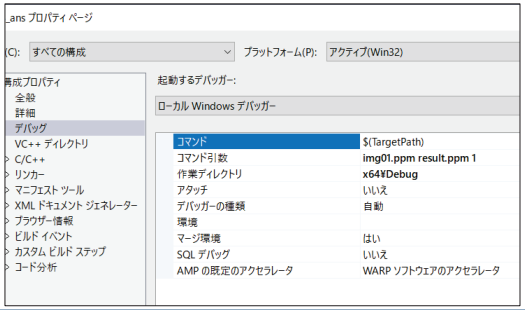
- ユーザのコマンドライン引数指定により，入力ファイル名，出力ファイル名，画像処理を選択できるようにすること
  - ex4.exe [入力ファイル名] [出力ファイル名] [モード]
- モードの仕様
  - 0：減色処理，1：上下反転，それ以外：コピー
  - オプション課題を実装したら2, 3, 4…で指定する
- 減色処理における下位nビットマスクは，n = (自身の学籍番号の下一桁 % 7) + 1 とする

コマンドライン引数の指定例

- コマンドラインの場合



- Visual Studioのコマンド引数の場合



実装の方針(1)

- main.cで，コマンドライン引数によってユーザが画像処理を選べるように改良する
  - main.cでのコマンドライン引数は”文字列”なので注意

```
int main(int argc, char* argv[])
{
  ...
  int mode;
  mode ← コマンドライン引数によるmode指定;

  ... /* 画像のロード */

  /* modeが0であれば減色処理 */
  /* modeが1であれば上下反転 */
  /* それ以外であればコピー */

  ... /* 画像のセーブ */
```

※文字列から数値への変換はatoi関数が便利.
[例]
```
  int b;
  b = atoi(″3″);
```
とすると，bには数値として3が代入される.

実装の方針(2)

- img_proc内，ipUDMirror関数を新たに作成
  - 上下を反転する方法は各自で考えること

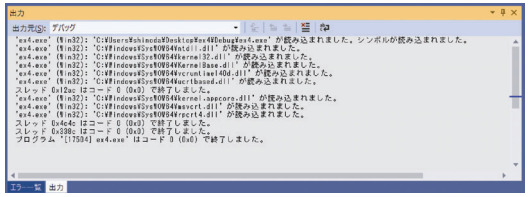
```
void ipUDMirror(IMAGE *p_in_image, IMAGE* p_out_image)
{
  int i, j;

  ...

}
```


レポートに載せる実行結果について

- 全てのモードの出力画像をのせ，結果が正しいか各画像ごとに説明し，必要に応じて考察をする
- デバッグ実行後の出力ウィンドウ(下記)を先頭行から全てのせ，メモリリークがないかを説明
  - モードに1を指定したときのみでかまわない
  - 確認ができればVisual Studio以外でもよい



最終レポートについての補足

- ソースコードと実行結果とも全員異なる内容になるのでレポートのコピペは不可能
  - 他人のソースコードやレポートをコピーした場合，オリジナルの作成者を含め単位は保証しない
  - これが原因で毎年数名が不可となっている
- 画像処理の実装種類数が多いほど評価UP
  - 例年は8割以上の学生がオプション課題を実装し，中には10種類以上の処理を実装する人も数名

レポートのテンプレートファイル

- レポート作成にあたっては，講義webページにあるレポートテンプレートファイルをダウンロードして利用すること
- 提出時はPDFに変換すること

終わった人は

- 最終週も出席を取るため，全員ネットワーク実験室に集まること

補足

より良い例外処理

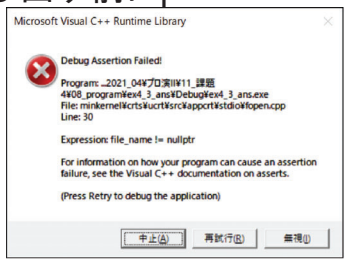
- ユーザが不適切な方法でプログラムを実行した場合でも安全に終了させるための工夫
  - argcをチェックし，こちらが意図した分だけコマンドライン引数が与えられているか
  - 入力/出力ファイルは正しくopenできているか
  - モードの書式は正しいか
- 不適切な場合，エラーメッセージを出力するとともにmainで1をreturnして終了するとよい

簡単なデバッグテクニック

- 怪しい処理行の前にブレークポイントを置き，ステップ実行（1行ずつ実行）
  - どの処理に分岐するか一目でわかる
  - 止まらない場合は，処理がそこまで到達していない
- （全環境共通）適当なprintf文を大量に設置する
  - コマンドプロンプトに表示される文字から，どこまで処理されたか一目でわかる
  - かなりいいかげんな方法であり，ステップ実行が難しい環境などで有用

実行時エラー

- エラー文を読むことで解決することが多い
- よくあるケース
  - コマンドライン引数を与えずに実行
  - 動的確保/解放のコードに間違い
  - iioMallocImageBuffer関数を呼び出す前にpBufferに画素値を書き込み/読み込み
  - 画像処理関数において，配列の範囲外にアクセス



画像処理テクニックの例（1）

- 回転，反転
  - コピー先のインデクスを変える（回転時はサイズ注意）
- モノクロ化
  - 各画素でRGB値を同じ値にする
- 拡大/縮小
  - 拡大：ある画素の値を複数の画素にコピー
  - 縮小：特定の位置の画素はコピーし，他は切り捨て
- モザイク化
  - 出力画像のn×n画素を入力画像の同範囲の平均値で置き換え

画像処理テクニックの例（2）

- 特定の色の物体を検出
  - RGB値がある範囲内の画素値はそのまま，他は0で置き換える
- ノイズ付加
  - ある確率分布に従う乱数を生成し，画素値に付加
- ノイズ除去
  - 付加されたノイズによって対処方法が異なる
  - 平均値フィルタやメディアンフィルタなど
- ぼかし処理，輪郭強調，エッジ検出
  - 講義資料webページの補足説明を参照

バイナリエディタ(1/2)

- 画像ファイルやテキストファイルのバイナリ情報が閲覧できる
- これを利用すれば減色処理の正否がわかる
- Visual Studioの場合, ファイル→開く→ファイル . . . →対象のファイルを選択する→「開く」の下矢印→「ファイルを開くアプリケーションを選択」→「バイナリエディター」を選択

バイナリエディタ(2/2)

- バイナリデータと位置は16進数表記
- 右側にASCIIコードに従ったテキストも表示